C sharp Book for Beginners:-

1. Sample Program

```
//Namespace Declaration
using System;
class Program;
{
public static void Main()
{
//write to console
Console.WriteLine("Welcome to Pragim Technologies!");
}
}
```

Using namespace declaration

The namespace declaration ,using system, indicates that you are using the system namespace. A namespace is used to organize your code and is collection of classes, interfaces, structs, enums and delegates.Main method is the entry point into your applications

Eg:-

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace _1_SampleProgram
{
    class Program
    {
        static void Main1()
        {
            Console.WriteLine("Welcome to C# classes training part2!");
            Console.ReadLine();
        }
        static void Main()
        {
            Console.WriteLine("Welcome to C# Training Classed");
            Main1();
            Console.ReadLine();

        }
    }
}
```

2. Reading and writing to console

```
using system;
calss Program
{
 static void Main()
  {
  //Prompt the user for his name
  console.WriteLine("Please enter your name");
  //Read the name from console
  string UserName=Console.Readline();
  //concatenate name with hello world and print
  Console.WriteLine("Hello"+UserName);
  //peaceholder syntax to print name with hello world
  //conole.WriteLine("Hello{0}",UserName);
  }
  }
```

Note:-C sharp is case sensitive

Eg:-

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace _2_WriteLineandReadline
{
    class Program
    {
        static void Main()
        {
            Console.WriteLine("Please enter your FirstName");
            string FirstName = Console.ReadLine();

            Console.WriteLine("Please enter your LastName");
            string LastName = Console.ReadLine();

            Console.WriteLine("Hello {0},{1}", FirstName,LastName);
            //Console.WriteLine("Hello" + UserName);
            Console.ReadLine();
        }
    }
}
```

3. Built-in type in C#
#Bolean type==>Only true or false
#Integral Type==> sbyte,byte,short,unshort,int,uint,long,ulong,char

#Floating Type==> float and double
#Decimal Type
#String Type

For Eg:-

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace _3Built_in_Types
{
    class Program
    {
        static void Main()
        {
            bool b = true;
            int i = 0;
            Console.WriteLine("Min={0}", int.MinValue);
            Console.WriteLine("Max={0}", int.MaxValue);


            double d= 123.222334455;
            Console.WriteLine(d);
            Console.ReadLine();

        }
    }
}
```

4. String

string Name ="\"Ganesh\"";
string Name = "One\nTwo\nThree";
Q)What is Verbatim Literal?
Ans==>verbatim literal,is a string with an @ symbol perfix, as it @"Helo"
verbatim literals make escape sequences translate as normal printable characters to enhance readbility
for eg:- without verbatim literal:-"c:\\Ganeh\\DotNet\\Training\\Csharp"
With verbatim literal:-@"c:\\Ganeh\\DotNet\\Training\\Csharp"

Eg:-

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace _4_StringDataType
{
    class Program
    {
        static void Main()
        {

            string Name = @"c:\\Ganeh\\DotNet\\Training\\Csharp";
            Console.WriteLine(Name);
            Console.ReadLine();
        }
    }
}
```

5. Common operator in c

#Assignement Operator like =

Arithmatic Operators like +,-*,/,%

Comparison Operators like ==,!=,>,>=,<,<=

Conditional Operator like &&, ||

Ternary Operator like ?:

Null Coalescing Operator ??

Eg:-

```
sing System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace _5_CommonOPerator_In_C_Sharp
{
    class Program
    {
        static void Main(string[] args)
        {
            //Assignment Operator
            int i = 10;
            bool b = true;


            //Aerithmatic Operator
            int numerator = 10;
            int Denominator = 2;
            int result = numerator / Denominator;
```

```
        Console.WriteLine("Result={0}", result);



        //Comparison Operator
        int Number = 10;
        if (Number==10)
        {


        }
        if (Number!=10)
        {

        }

        //Conditional Operator
        int N1 = 10;
        int N2 = 20;

       // if (N1==10 && N2==20)//both the condition should be satisfy
        if (N1 == 10 || N2 == 30) //one condition should be satisfy
        {
            Console.WriteLine("Hello");
            Console.ReadLine();
        }


        //Ternary Operator


        /*int num = 15;
        bool IsNumber10;
        if (num==10)
        {
            IsNumber10 = true;

        }
        else
        {
            IsNumber10 = false;
        }
        Console.WriteLine("Number==10 is {0}", IsNumber10);
        Console.ReadLine();
        */
        int num = 15;
        bool isNumber10 = Number == 10 ? true : false;
        Console.WriteLine("Number==10 is {0}", isNumber10);
        Console.ReadLine();
    }
```

```
    }
}
```

6.Nullable Type in C sharp

In C sharp typsses are divided into 2 broad categories
#Value types==>int, float, double,structs,enum etc
#References Types==> interface,class,delegates,arrays etc

#By default values types are non vullable to make them nullable use?
int i=0(i is non nullable, so i can not be set to null, i=null will generate compile error)

int?j=(j is nullable int,so j=null is legal)

Nullable types bridge the differences between c # types and Database types

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace _6_Nullable_Types_of_C_sharp
{
    class Program
    {
        static void Main(string[] args)
        {
            /*
            bool? areyouMajor = null;
            if (areyouMajor==true)
            {
                Console.WriteLine("User is Major");
                Console.ReadLine();
            }
            else if(areyouMajor==false)
            {

                Console.WriteLine("User is not Major");
                Console.ReadLine();
            }

            else
```

```
            {
                Console.WriteLine("User did not answer the question");
                Console.ReadLine();
            }*/
            int? ticketonsales = 5;


            int availabletickets=ticketonsales??0;//null colleciong operator



           /* if (ticketonsales==null)
            {
                availabletickets = 0;
            }
            else
            {
                availabletickets =(int) ticketonsales;
            }
            * */
            Console.WriteLine("AvailableTickets={0}", availabletickets);
            Console.ReadLine();
           }

        }
    }
```

7. Data Type Conversion in C Sharp:-

Data type conversion in C sharp. There are two types of conversion i.e #implicit and Explicit
#Implicit conversion:-
Implicit conversion done by the compiler
1.When there is no loss of data in conversion
2. If there is no possibility of throwing the exception during the conversion
Example:-Converting an int to float will not lose any data and no exception will be thrown hence an implicit conversion can be done

Whereas when converting a float to an int , we lose the fractional part
so conversion is required. For explicit conversion we can use cast operator or the convert class in c sharp

Difference between Try and TryParse
If the number is in a string format you have two option-parse() and TryParse()

Parse() method throws and exception if it cannot parse the value,whereas TryParse() returns a bool whatever it suceceeded or failed.

Use parse() if you are sure the value will be valid, otherwise use TryParse();

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace _7_DataTypeConversion
{
    class Program
    {
        static void Main(string[] args)
        {
            //not possible
            /*float f = 100.25;

            int i = f;
            Console.WriteLine(i);
            Console.ReadLine();
             */

            //possible to do

           /* int i = 100;

            float f = i;
            Console.WriteLine(f);
            Console.ReadLine();
            */



           /* float f = 123.60F;
            int i = Convert.ToInt32(f);
            Console.WriteLine(i);
            Console.ReadLine();
            */



            string strNumber = "100TG";
            int Result=0;
         bool IsConversionIsSucesseful=   int.TryParse(strNumber,out Result);
          //  int i = int.Parse(strNumber);
```

```
            if (IsConversionIsSucesseful)
            {
                Console.WriteLine(Result);
                Console.ReadLine();

            }
            else
            {
                Console.WriteLine("Enter valid NUmber");
                Console.ReadLine();

            }

        }
    }
}
```

8.Array

An Array is a collection of similar data types
Examples:
int[] EvenNumbers= new int[3];
EvenNumbers[0]=0;
EvenNumbers[1]=2;
EvenNumbers[2]=4;
//Initilize and Assign Values in the same line
int[] OddNumbers={1,2,3};
Advantages:-Arrays are strongly typed.
Disadvantages=Arrays cannot grow in size once initialized.Have to rely on integral indices to store or retrive items from the array.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace _8_Array
{
    class Program
    {
        static void Main(string[] args)
        {
            int[] EvenNumbers = new int[3];

            EvenNumbers[0] = 0;
            EvenNumbers[1] = 2;
            EvenNumbers[2] = 4;
```

```
            Console.WriteLine(EvenNumbers[1]);
            Console.ReadKey();

        }
    }
}
```

9. Comments in C sharp

#single lIne comments -//
#multiline Comments (-/* * /)
#xml Documentation Comments -///

Comments are used to do what the program does and what specific blocks or lines of code do. C# compiler ignores comments

To comment and Uncomment, there are two ways
* 1. Use Designer
* 2. Keyboard Shutcut: ctrl+K,ctrl+c and CTRL+K,Ctrl+U

Note:-Do not try to comment every line of code .Use comment only for blocks of lines of code that is difficult to understand

10. If statement in C sharp

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace _10_IFStatement
{
    class Program
    {
        static void Main(string[] args)
        {

            Console.WriteLine("please enter a number");
           int UserNumber=int.Parse( Console.ReadLine());
            /*
           if (UserNumber == 1)
           {
               Console.WriteLine("your number is one");
               Console.ReadLine();
           }
          else if (UserNumber==2)
           {
               Console.WriteLine("your number is 2");
               Console.ReadLine();
           }
```

```
            else
             {
                 Console.WriteLine("your number is not between 1 and 3");
                 Console.ReadLine();
              }
              * */


              if (UserNumber==10|| UserNumber ==20)
        {
          Console.WriteLine("your number is 10 or 20");
        }

          }
    }
}
```

11. Switch Statement:-

Multiple if else statements can be replaced with a switch statement

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace _11_SwitchStatment
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("please enter a number");
            int UserNumber = int.Parse(Console.ReadLine());
            /*
            if(UserNumber==10)
            {
                Console.WriteLine("Your number is 10");
            }
            else if(UserNumber==20)
            {
                Console.WriteLine("Your number is 20");
            }
            else if(UserNumber==30)
            {
                Console.WriteLine("Your number is 30");
            }
```

```
            else
            {
                Console.WriteLine("your number is not 10,20 & 30");
            }
             */

            switch(UserNumber)
            {
                    /*
                case 10:
                    Console.WriteLine("YOur number is 10");
                    break;
                case 20:
                    Console.WriteLine("your number is 20");
                    break;
                case 30:
                    Console.WriteLine("Your number is 30");
                    break;
                default:
                    Console.WriteLine("your number is not 10,20 and 30");
                    break;
                     */



                case 10:
                case 20:
                case 30:
                    Console.WriteLine("your number is {0}", UserNumber);
                    break;
                default:
                    Console.WriteLine("your number is not 10,20 and 30");
                    break;

            }
            Console.Read();
        }
    }
}
```

12. Switch Statement Continued.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
```

```
namespace _12_Switch_Statment_Continued
{
    class Program
    {
        static void Main(string[] args)
        {


          int totalCoffeCost = 0;
            start:
            Console.WriteLine("Please select your coffe size:-1 -small, 2-
Medium, 3-large");
            int UserChoice  = int.Parse(Console.ReadLine());
            switch (UserChoice)
            {
                case 1:
                    totalCoffeCost += 1;
                    break;
                case 2:
                    totalCoffeCost += 2;
                    break;
                case 3:
                    totalCoffeCost += 3;
                    break;
                default:
                    Console.WriteLine("your choice{0} is invalid",
UserChoice);
                    goto start;
            }
        Decide:
            Console.WriteLine("Do you want to by another Coffe-Yes or NO?");
            string UserDecision = Console.ReadLine();
            switch (UserDecision.ToUpper())
            {
                case "YES":
                    goto start;
                case "NO":
                    break;
                default:
                    Console.WriteLine("Your choice {0} is invalid.please try
again",UserDecision);
                    goto Decide;
            }

            Console.WriteLine("thank you for shopping with us");
            Console.WriteLine("bill amount={0}", totalCoffeCost);

                    Console.Read();
        }
```

```
    }
}
```

13. While Loop
 * While loop check the conditional first
 * If the condition is true, statement with in the lop are executed
 * This process is repeated as long as the condition evaluates

NOTE:-Don't Forget to update the variable participating int he condition, so the loop can end ,same point

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace _13_WhileLoop
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Plese enter your target?");
            int usertarget = int.Parse(Console.ReadLine());
            int start = 0;
            while (start<=usertarget)
            {
                Console.WriteLine(start);
                start = start + 2;

            }
            Console.ReadLine();
        }
    }
}
```

14. Do While Loop

A do while loop checks its condition at the end of the loop. This means that the do loop is guaranteed to execute at least one minute. Do loops are used to present a menu to the user

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

```
using System.Threading.Tasks;

namespace _14_Do_Whileloop
{
    class Program
    {
        static void Main(string[] args)
        {
              string UserChoice = "";
              do
              {
                  Console.WriteLine("please enter your target?");
                  int UserTarget = int.Parse(Console.ReadLine());
                  int start = 0;
                  while (start < UserTarget)
                  {
                      Console.WriteLine(start + "");
                      start = start + 2;
                  }
                  Console.ReadLine();

                  do
                  {
                      Console.WriteLine("do you want to continue-Yes or NO");
                      UserChoice = Console.ReadLine().ToUpper();
                      if (UserChoice != "Yes" && UserChoice != "NO")
                      {
                          Console.WriteLine("Invalid choice Please select yes
or NO");
                          Console.ReadLine();
                      }
                  } while (UserChoice != "YES" && UserChoice != "NO");
              } while (UserChoice == "YES");
        }

    }
}
```

15. for Loop and For Each Loop

For Loop:-A for loop is very similar to while loop we do the initialization at the one place, condition check at another place and modifying the variable at another place, whereas for loop has all of these at one place.

Foreach Loop:-A foreach loop is used to iterate through the items in a collection. For Example, foreach loop can be used with arrays or collections such as Array List, HashTable and Gererics. We will cover collections and generics in a later session.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace _15_For_and_ForEach
{
    class Program
    {
        static void Main(string[] args)
        {


            /*
            int[] Number = new int[3];
            Number[0] = 101;
            Number[1] = 102;
            Number[2] = 103;
            int i = 0;
            for (int j = 0; j <3; j++)
            {
                Console.WriteLine(Number[j]);
            }




            foreach (int k in Number)
            {
                Console.WriteLine(k);
            }
            Console.ReadLine();




            while (i < Number.Length)
            {
                Console.WriteLine(Number[i]);
```

```
            i++;
        }
        Console.ReadKey();
        */


    /*  for (int i = 1; i <= 20; i++)
        {
            Console.WriteLine(i);
            if (i==10)
            {
                break;
            }
        }
        Console.ReadLine();
     */




        for (int i = 0; i <= 20; i++)
        {
            if (i%2==1)
            {
                continue;
            }
            Console.WriteLine(i);
        }
        Console.ReadLine();




    }
}
}
```

16. Methods

Q)WHY METHODS?
Methods are also called as functions. These terms are used interchangeably. Methods are steamily useful because they allow you to define your logic once, and use it, at many places. Methods make the maintenance of your applications easier

Methods

*[attributes]
 * access-modifiers return-type method-name(parameters)
 * {
 * Method Body
 * }

1. We will talk about attributes and access modifiers in later Session
2. Return type is any valid data types of void
3. Method name can be any meaningful Name
4. Parameters are optional

STATIC VS INSTANCE METHOD
#when the method declaration includes static modifiers, that method is said to be a static method
When the static modifier is present, the method is said to be as instance method. #Static method is invoked using the class name, where as an instance methods is that multiple instance of a class can be created(or instaiated) and each instance has its own separate method. However, when a method is static, there are no instances of that
Method and you can invoke only that one definition of the static method.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace _16Methods
{
    class Program
    {
        static void Main(string[] args)
        {
            Program p = new Program();
            p.EvenNumber(30);
          int sum=  p.Add(10, 20);
          Console.WriteLine("Sum={0}", sum);
          Console.ReadLine();
        }

        public int Add(int FirstNumber, int secondNumber)
        {
            return FirstNumber + secondNumber;
        }
        public void EvenNumber(int Target)
        {
            int start = 0;
            while (start<=Target)
            {
                Console.WriteLine(start);
```

```
                start = start + 2;

            }
            Console.ReadLine();
        }
    }
}
```

17. METHOD PARAMETER

There are 4 different types of parameters a method can have
1. Value Parameters:- Create a copy of parameters passed, so modifications does not affect each other's
2. Reference Parameters:- The reference method parameter keyword on a method parameter causes a method to refer to the same variable that
was passed into the method. Any changes made to the parameter in the method will be reflected on that variable when control passed back
to the calling method.
3. Out parameter==> Use when you want a method to return more than one value.
4. Parameter Arrays:-The parms Keywords letes you specify a method parameter that takes a variable number of arguments. you can send comma-separated list of arguments, or an array, or no arguments.
b. parameter keyword should be the last one in a method declaration, and only one param keyword is permitted in a method declaration
NOTE:-Method Parameter VS Method Arguments

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace _17_MethodParameter
{
    class Program
    {
        static void Main(string[] args)
        {
            //value and Reference parameter
            /*int i = 0;
            SimpleMethod(ref i);
            Console.WriteLine(i);
            Console.Read();
             */



            //output Parameter
```

```
            /*
            int total=0;
            int Product=0;
            Calculate(10, 20, out total, out Product);
            Console.WriteLine("Sum={0} && Product={1}",total,Product);
            Console.Read();
             */



            int[] Numbers = new int[3];
            Numbers[0] = 101;
            Numbers[1] = 102;
            Numbers[2] = 103;
           // ParamsMethod();
            ParamsMethod(Numbers);
            ParamsMethod(1, 2, 3, 4, 5);
            Console.Read();


        }

       /* public static void SimpleMethod(ref int j)
        {
            j=101;
        }
        */


        /*//output parameter
        public  static void Calculate(int FN,int SN,out int Sum,out int
Product)
        {
            Sum= FN+SN;
            Product = FN + SN;
        }
         */



        public static void ParamsMethod(params int[] Numbers)
        {
            Console.WriteLine(" there are {0}", Numbers.Length);
            foreach(int i in Numbers)
            {
                Console.WriteLine(i);
            }
        }
    }
}
```

18. NAMESPACES

Q. WHY Namespaces?
Namespaces are used to organize your programs. They also provide assistance in avoiding name clashes

NAMESPACES
Namespaces don't correspond to file, directory or assembly names. They could be written in separate files and /or separate assemblies and still belong to the same namespaces.
Namespaces can be nested in 2 ways.
Namespaces alias directives. Sometimes you may encounter a long namespace and wish to have it shorter this could be
#this could improve readability and still avoid name clashes with similarity named methods.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
//using PATA= ProjectA.TeamA;
//using PATB = ProjectA.TeamB;
using PATA=PROJECTAA.TEAMA;
using PATB= PROJECTA.TEAMB;

namespace _18_NameSpaces
{
    class Program
    {
        static void Main(string[] args)
        {
            PATA.ClASSA.print();
            PATB.CLASSA.print();

            /*
            ProjectA.TeamA.ClassA.Print();
            ProjectA.TeamB.ClassA.Print();
             */

/*
            PATA.ClassA.Print();
            PATB.ClassA.Print();
 */
```

```
            Console.Read();

        }
    }
}
```

```
/*
namespace  ProjectA
{
    namespace TeamA
    {
        class ClassA
        {
            public static void Print()
            {
                Console.WriteLine("tea A print MEthod");
            }
        }
    }
}
 */
```

```
/*
namespace ProjectA
{
    namespace TeamB
    {
        class ClassA
        {
            public static void Print()
            {
                Console.WriteLine("tea A print MEthod");
            }
        }
    }
}*/    }
}
```

19. Classes

Q. What is a Class?

So far in this video tutorial we have seen simple data types like int, float ,double etc. If you want to create complex custom types, then we can make use of classes

A class consist of data and behavior. Class data is represented by its fields and behavior is represented by its method

Purpose of a class constructor:-
The purpose of the class constructor is to initialize class fields' class constructor is automatically called when instance of the class is created. Constructors do not have return value and always have the same name as the class Constructors do not have mandatory. If we do not provide a constructor, a default Parameter less constructor is automatically provided. Constructor can be overloaded by the number and types of parameters.

#Destructors
Destructors have the same name as the class with ~symbol in front of them
They do not take any parameter and do not return value
Destructor are places where you could put code to release any resources your class was holding during its lifetime.
We will cover this in details in a later session.
They are normally called when the c# garbage collector decide to clean your object from Memory.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace _19_Classes
{
    class Customer
    {
        string _FirstName;
        string _LastName;

        public Customer():this("No FirstNameProvided","No last Name
provided")
        {

        }
        public Customer(string FrinstName, String Lastname)
        {
            this._FirstName = FrinstName;
            this._LastName = Lastname;
        }

        public void PrintFullName()
        {
            Console.WriteLine("Full Name is: " + _FirstName + " " +
_LastName);
        }
```

```
      /* ~Customer()
        {
            //clean up code
        }
       */

    }
    class Program
    {
        static void Main(string[] args)
        {
            //Customer c1 = new Customer("Ganesh", "Chauhan");
            Customer c1 = new Customer();
            c1.PrintFullName();
            Customer c2 = new Customer("Ganesh", "Chauhan");
            c2.PrintFullName();
            Console.Read();
        }
    }
}
```

20. Static and Instance class Members

When a class member includes a static modifier, the member is called as static member. When no static modifier is present the member is called as non-static member or instance member

#Static member are invoked using class name, whereas instance member are invoked using instances (objects) of the class.

#An instance member belongs to specific instance (object) of class. If i create 3 objects of a class, i will have 3 sets of instance member in the memory, whereas there will ever be only one copy of a static member, no matter how many instance of a classes are created

NOTE:-Class members=fields, methods, properties, events, indexers, constructors.

Static Constructor

Static constructor is used to initialize static fields in a class .You declare a static constructor by using the static keywords in front of the constructor name. Static constructor is called only once, no matter how many instance you create. Static constructor are called before instance constructors

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

```
using System.Threading.Tasks;

namespace _20_Static_and_InstanceClassMember
{
    class Program
    {
        class circle
        {
            static float PI; //= 3.141F;
            int _Radius;
            static circle()
            {
                Console.WriteLine("Static Constructor");
                circle.PI=3.141F;
            }

            public circle (int Radius)
            {
                Console.WriteLine("Instance constructor");
                this._Radius = Radius;
            }
            public static void Print()
            {

            }
            public float CalculateArea()
            {
                return circle.PI * this._Radius * this._Radius;
            }
        }
        static void Main(string[] args)
        {
            circle c1 = new circle(5);
            float Areaa1 = c1.CalculateArea();
            circle.Print();
            Console.WriteLine("Area={0}", Areaa1);

            circle c2 = new circle(6);
            float area2 = c2.CalculateArea();
            Console.WriteLine("Area={0}", area2);
            Console.Read();
        }
    }
}
```

21. Inheritance

Why Inheritance?

```
/*
 public class FullTimeEmployee()
 {
  string FirstName;
```

```
  string LastName;
  string Email;
  float yearlySalary;
  public void PrintFullName()
  {
  }

 }

 public class PartTimeEmployee()
 {
  string FirstName;
  string LastName;
  string Email;
  float HourlySalary;
  public void PrintFullName()
  {
  }
A lots of code between these two is duplicated
 */
```

We can minimize the code by using the base class like above all the common code into base Employee class

```
  public class Employee()
 {
  string FirstName;
  string LastName;
  string Email;
  public void PrintFullName()
  {
  }

//full time and part time specific code in the respective derived classes
 public class FulltimeEmployee
 {
  float YearlySalary;
 }



public clTass partTimeEmployee
 {
 float YearlySalary;
 }
*/
```

#Pillars of Object oriented Programming

1. Inheritance
2. Encapsulation

3. Abstraction
4. Polymorphism

1. Inheritance is one of the primary pillars of object oriented programming
2. It allows code reuse
3. Code reuse can reduce time and errors.

Note:-you will specify all the common fields, properties, method in the base class, , which allows reusability. In the derived class you will user only have fields, properties and methods that are specific to them.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace _21_Inheritance
{/*
    public class Employee
    {
       public string FirstName;
       public  string LastName;
        public string Email;

        public void printFullName()
        {
            Console.WriteLine(FirstName + "  " + LastName);
            Console.ReadKey();
        }
    }
    public class FullTimeEmployee : Employee
    {
        public  float YearlySalary;
    }
    public class PartTimeEmployee : Employee
    {
      public float HourlyRate;
    }
  */
    public class ParentClass
    {
        public ParentClass()
        {
            Console.WriteLine("Parent class constructor called");

        }
        public ParentClass(string Message)
```

```
        {
            Console.WriteLine(Message);
        }
        public class ChildClass : ParentClass
        {
            public ChildClass():base("Derived class controlling parent
class")
            {
                Console.WriteLine("Child class consturctor called");
            }
        }


        class Program
        {
            static void Main(string[] args)
            {
                /*
                FullTimeEmployee FTE = new FullTimeEmployee();
                FTE.FirstName = "Pragim";
                FTE.LastName = "Tech";
                FTE.YearlySalary = 50000;
                FTE.printFullName();


                PartTimeEmployee PTE = new PartTimeEmployee();
                PTE.FirstName = "part";
                PTE.LastName = "Time";
                PTE.printFullName();
                 */
                ChildClass CC = new ChildClass();
                Console.ReadKey();

            }
        }

    }

}
```

22. Method Hiding In C Sharp

While inherit new class from base class we may not need the base class method in Derived class, so in that case we can hide the base class method to the derived class by Method hiding process.

```
using System;
```

```
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace _22_MethodHiddingINCsharp
{
    public class Employee
    {
        public string FirstName;
        public string LastName;
        public void PrintFullName()
        {
            Console.WriteLine(FirstName + " " + LastName);
        }
    }
    public class PartTimeEmployee:Employee
    {

    }
    public class FullTimeEmployee:Employee
    {
        public new void PrintFullName()
        {
           // base.PrintFullName();
            Console.WriteLine(FirstName + " " + LastName+ "-Contractor");
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            FullTimeEmployee Fte = new FullTimeEmployee();
            Fte.FirstName = "FullTime";
            Fte.LastName = "Employee";
            Fte.PrintFullName();


            PartTimeEmployee PTE = new PartTimeEmployee();
            PTE.FirstName = "PartTime";
            PTE.LastName = "Employee";
           // PTE.PrintFullName();
            ((Employee)PTE).PrintFullName();
            PTE.PrintFullName();
            Console.Read();
        }
    }
}
```

23. Polymorphism

Polymorphism
Polymorphism is one of the primary pillars of object-oriented programming.
Polymorphism allows you to invoke derived class methods through a base class reference during runtime
In base class the method is declared virtual, and in derived class we override the same method. The virtual keyword indicates, the method can be overridden in any derived class

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace _23_Polymorphism
{
    public class Employee
    {
        public string FirstName = "FN";
        public string LastName = "LN";

        public virtual void PrintFullName()
        {
            Console.WriteLine(FirstName + "" + LastName);
        }
    }
    public class PartTimeEmployee:Employee
    {

        public override void PrintFullName()
        {
            Console.WriteLine(FirstName + "" + LastName+"-Part Time Employee-
--");
        }
    }
    public class FullTimeEmployee:Employee
    {
        public override  void PrintFullName()
        {
            Console.WriteLine(FirstName + "" + LastName + "-Full time
Employee---");
        }
    }
    public class TemporaryEmployee:Employee
    {
        public override  void PrintFullName()
        {
```

```
            Console.WriteLine(FirstName + "" + LastName + "Temporary
Employee---");
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Employee[] employees = new Employee[4];
            employees[0] = new Employee();
            employees[1] = new PartTimeEmployee();
            employees[2] = new FullTimeEmployee();
            employees[3] = new TemporaryEmployee();


            foreach(Employee e in employees)
            {
                e.PrintFullName();
            }
            Console.Read();
        }
    }
}
```

24. Method Overriding vs. Method Hiding

Method Overriding
In method overriding a base class references variable pointing to a child class object, will invoke the overridden method int the child class
Method Hiding:-
In method hiding a base class reference variable pointing to a child class object, will invoke the hidden method in the base class.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace _24_Method_Overriding_vs_MethodHiding
{
   public class BaseClass
   {
      public virtual void print()
```

```
        {
            Console.WriteLine("I am Base class print Method");
        }
    }
    public class DerivedClass:BaseClass
    {
        //method ovveriding
        /*
        public override void print()
        {
            Console.WriteLine("I am Derived class method");
        }*/


        //method hidding
        public new void print()
        {
            Console.WriteLine("I am derived class method.");
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            BaseClass b = new DerivedClass();
            b.print();

            DerivedClass d = new DerivedClass();
            d.print();
            Console.Read();
        }
    }
}
```

25. Method Overloading

Method Overloading:-
function overloading and Method overloading terms are used interchangeably.
Method overloading allows classes to have multiple methods with the same name with a different signature. So, in C# functions can be overloaded based on the number, type(int, float etc.) and kind(value, Ref or Out) of parameters
#the signature of a method consists of the name of the method and the type, kind (value, reference, or output) and the number of its formal parameters. The signature of a method does not include the return type and the prams modifiers. so, it is not possible to
Overload a function, just based on the return type or prams modifiers.
NOTE:- If you want to know about different kinds of methods parameters, please watch part 17- Method Parameters, in this video series.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace _25_MethodOverLoading
{
    class Program
    {
        static void Main(string[] args)
        {

        }
        public static void Add(int FN, int SN)
        {
            Console.WriteLine("Sum=", FN + SN);
        }
       /* public static void Add(int FN,int SN, int TN)
        {
            Console.WriteLine("Sum={0}", FN + SN);
        }
        */
        public static void Add(float FN,float SN)
        {
            Console.WriteLine("Sum= ", FN + SN);
        }
        public static void Add(int FN,float SN)
        {
            Console.WriteLine("Sum: ", FN + SN);
        }
        public static void Add(int FN,int SN,int Tn,int Fourth)
        {
            Console.WriteLine("Sum= ", FN + SN);
        }
        public static void Add(int FN,int SN, out int TN)
        {
            Console.WriteLine("Sum={0}", FN + SN);
            TN = FN + SN;
        }
        public static void Add(int FN, int SN,  int TN)
        {
            Console.WriteLine("SUm={0}",FN+SN);
        }


        //will get error at the time of compilation
       /* public static int Add(int FN,int SN,int TN)
        {
            Console.WriteLine("Sum={0}", FN + SN + TN);
```

```
            return FN + SN + TN;
        }*/

        public static void add(int FN, int SN, int[] TN)//not differnet if we
create another method and pass the parameter params int[] TN
        {
            Console.WriteLine("Sum={0}", FN + SN);
        }



    }
}
```

26. Why Properties?

Why Properties?
Marking the class fields public and exposing to the external word is bad, as you will not have control over what gets assigned and returned.

```
public class Student
{
public int Id;
public string Name;
public int PassMark;
}
public class Program
{
public static void Main()
{
 student C1= new Student();
C1.Id=-101;
C1.Name=null;
C1.PassMark=-100;
Console.Writeline("Id={0}&Name={1} & PassMark={2}",C1.Id,C1.Name,C1.PassMark);
}
}
```

Program with public fields
1 Id should always be non-negative number
2. Name cannot be set to be null
3. If student name is missing "no Name" should be returned
4. Pass Mark should be read only
In below example we use setId(int ID) GetId() method to encapsulate _id class field.
As a result, we have better control on what gets assigned and returned from the _id field
Note:-Encapsulation is one of the primary pillars of Object oriented programming.

```
using System;
using System.Collections.Generic;
```

```
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace _26_WhyProperties
{
    public class Student
    {
        private int _id;
        private string _Name;
        private int _PassMark = 35;

        public int GetPassMark()
        {
            return this._PassMark;
        }
        public void SetId(int Id)
        {
            if(Id<=0)
            {
                throw new Exception("Student id can not be negative");
            }
            this._id = Id;
        }
        public void SetName(string Name)
        {
            if(string.IsNullOrEmpty(Name))
            {
                throw new Exception("Name cannot be null or empty");
            }
            this._Name = Name;
        }
        public string Getname()
        {
            return string.IsNullOrEmpty(this._Name) ? "NO Name" : this._Name;
          //Same as terniary operator
            /* if(string.IsNullOrEmpty(this._Name))
            {
                return "NO Name";
            }
            else
            {
                return this._Name;
            }*/
        }

        public int GetId()
        {
            return this._id;
        }
```

```
    }
    class Program
    {
        static void Main(string[] args)
        {/*
            Student C1 = new Student();
            C1.ID = -101;
            C1.Name = null;
            C1.PassMark = 0;

            Console.WriteLine("ID={0} && Name={1}&& PassMark={2}", C1.ID,
C1.Name, C1.PassMark);
            Console.Read();
          */



            Student C1 = new Student();
            C1.SetId(101);
            C1.SetName("Ganesh");

            Console.WriteLine("Student Id={0}", C1.GetId());
            Console.WriteLine("Student Name={0}", C1.Getname());
            Console.WriteLine("Student PassMark={0}", C1.GetPassMark());
            Console.Read();
        }
    }
}
```

27. Properties in C Sharp

In C sharp to encapsulate and protect fields we use properties
1. We use get and Set accessory to implement property
2. A property with both get and set accessor is a Read/Write property
3. A property with only get accessor is a Readonly Property
4. A property with only set accessor is a write only property

NOte:-The advantage of properties over traditional Get() and Set() method is that, you canaccess them as if they were public fields.

Auto Implemented Properties:-

#If there is no additional logic in the property accossors, then we can make use of auto implemented properties intruduce in C# 3.0

#Auto-Implmented properties reduce the amount of code that we have to write

#when we use Auto-Implemented properties, the compiler creates private, anonymous fields that can only be accessed through the property's get and set accessors.

```
using System;
using System.Collections.Generic;
using System.Linq;
```

```
using System.Text;
using System.Threading.Tasks;

namespace _26_WhyProperties
{
    public class Student
    {
        private int _id;
        private string _Name;
        private int _passmark = 35;
        public string Email { get; set; }
        public string City
        {
            get;
            set;
        }
        public int PassMark
        {
            get
            {
                return this._passmark;
            }
        }
        public string Name
        {
            set
            {
                if (string.IsNullOrEmpty(value))
                {
                    throw new Exception("Name field can not be null or
empty");
                }
                this._Name = value;
            }
            get
            {
                return string.IsNullOrEmpty(this._Name) ? "No name" :
this._Name;
            }
        }
        public string GetName()
        {
            return string.IsNullOrEmpty(this._Name) ? "No Name" : this._Name;
        }
        public int Id
        {
            set
            {
                if (value <= 0)
                {
```

```
                    throw new Exception("Student id can not be negative");
                }
                this._id = value;
            }
            get
            {
                return this._id;
            }

        }


    }
    class Program
    {
        static void Main(string[] args)
        {
            Student C1 = new Student();
            C1.Id = 101;
            C1.Name = "Ganesh";
            Console.WriteLine("Student Id={0}", C1.Id);
            Console.WriteLine("Student Name={0}", C1.Name);
            Console.WriteLine("Student PassMark={0}", C1.PassMark);
            Console.ReadKey();

        }
    }
}
```

28. Struct

Struct:-
Just like classes structs can have
1. Private fields
2. Public fields
3. Constructor
4. Method

Object initializer syntax, introduced in C# 3.0 can be used to initialize either a struct or a class.
Note: There are several differences between classes and structures which we will be looking at in a later session.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace _28_Structs_in_C_sharp
{
```

```
    public struct Customer
    {
        private int _id;
        private string _Name;

        public string Name
        {
            get { return _Name; }
            set { _Name = value; }
        }
        public int ID
        {
            get {return this._id; }
            set { this._id=value; }
        }
        public Customer(int id, string Name)
        {
            this._id = id;
            this._Name = Name;
        }
        public void PrintDetails()
        {
            Console.WriteLine("Id={0} && Name={1}", this._id, this._Name);
            Console.ReadKey();
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Customer C1 = new Customer(101, "Ganesh");
            C1.PrintDetails();
            Customer C2 = new Customer();
            C2.ID = 102;
            C2.Name = "Radha";
            C2.PrintDetails();

            Customer c3 = new Customer
            {
                ID = 103,
                Name = "Rob"
            };
            c3.PrintDetails ();

        }
    }
}
```

29. Differences between Classes and Structs

Classes Vs Structs:-
#A struct a value type where as a class is a reference type
All the differences that are applicable to value type and references type are also applicable to classes and structs.
Structs are stored in a stack, Where as classes are stored in a heap.
#Value type hold their value in memory where they are decleared, but references type hold a references to a object memory
#Vlaue types distroyed immediately after the scope is lost, whereas for references type only the references variable is destroyed after the scope is lost.The object is later destroyed by garbage collector.(We will talk about this in the garbage collection session)
When you copy a struct into another struct, a new copyu of that struct get created and modifications on one struct will not affect the value contained by the other struct.
when you copy class into another class, we only get a bopy of the reference valiable. Both the reference variable point ot the same object on the heap, so ,operations on one variable will affect the value contained by the other reference bariable.
#Structs can't have destructors, but classes can have destructors.
#Struct cannot have explicit parameter less constructor where as class can.
#Struct can't inherit from another clas where as a class can, Bothe structs and classes can inherit from an interface.
Examples of structs int he .NET Framework-int(System.Int32),double(System.Double)etc.
NOTE1:-A class or a struct cannot inherit from another struct.Struct are sealed types.
NOTE2:-How do you prevent a class from being inherited?What is the signaficance of sealed keywords?

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace _29_DifferenceBetweenClassesAndStructs
{
    public class Customer
    {
        public int ID { get; set; }
        public string Name { get; set; }
    }
    public sealed class Customer2///if we  use keyword sealed then we can not inherit from this class.
    {
    }

    //struct can not have parameter less constructor

   /* public struct Customer1
```

```
    {
        public Customer1()
        {

        }
    }
    */
    class Program
    {
        static void Main(string[] args)
        {
            //a struct is a value type where as a class is reference type
            //value type store the value in stack but reference type store a
value in heap of the memory.
            /*
            int i = 10;
            if(i==10)
            {
                int j = 20;
                Customer C1 = new Customer();
                C1.ID = 101;
                C1.Name = "Ganesh";
              }
             */

            //when you copy struct into another struct a new copy of struct
gets created and modification of struct will not affect
            //and value contain by other struct


            //when you copy class into another class we only got the copy of
the reference variable both the reference variable point the same
            //object on the heap. so operation on one  variable  will affect
the  value contain bye the other reference variable

            int i = 10;
            int j = i;
            j = j + 1;
            Console.WriteLine("i={0} && j={1}", i, j);
            Console.Read();

            Customer c1 = new Customer();
            c1.ID = 101;
            c1.Name = "Ganesh";


            Customer C2 = c1;
            C2.Name = "Marry";
            Console.WriteLine("C1.Name={0}", c1.Name);
            Console.Read();
```

```
        }
    }
}
```

30. Introduction of Interfaces.

Interface:-
We create interface using interface keyword. Just like classes interfaces also contains properties, methods, delegates or events, but only declarations and no implementations.

It is a compile time error to provide implementations for any interface member.

Interface member are public by default, and they don't allow explicit modifiers.

Interface cannot contain fields.
If class or struct inherit from interface, it must provide implementation for all interface members. Otherwise we will get compile error.

A class or struct can inherit from more than one interface at the same time where as class can cannot inherit from more than once class at the same time.

Interfaces can inherit form other interface. A class that inherits this interface must provide implementation for all interface members in the entire interface inherit chain.

We cannot create instance of an interface but an interface reference variable can point to a derived class object.

Interface Naming Convention: Interface names are prefixed with capital I.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace _30_Introductonto_Interfaces
{
```

```
   /* interface ICustomer
    {
        //interface member can  have only decleration not implementation
        //interface member can not have access modifiers it it public by
default.
        void print();
        /*
    {
        Console.WriteLine("Hello");
    }
     */




   /* }
    interface I2
    {
        void I2Method();

       }

    class Customer : ICustomer,I2
    {
        public void print()
        {
            Console.WriteLine("Interface print Method");
        }
        public void  I2Method()
        {
            Console.WriteLine("I2 method Another method");
        }


    }
    */


    interface Icustomer1
    {
        void Print1();
    }

    interface Icustomer2
    {
        void print2();
    }
    public class Customer:Icustomer2
    {
        public void Print1()
```

```
        {
            Console.WriteLine("Interface print1 Method");
        }
        public void print2()
        {
            Console.WriteLine("I2 Method");
        }
    }


    class Program
    {
        static void Main(string[] args)
        {
            /*
            Customer c1 = new Customer();
            c1.print();
            c1.I2Method();
            Console.Read();
             */
            Customer c1 = new Customer();
            c1.Print1();
            c1.print2();


          //  Icustomer1 Cust = new Icustomer1();//we can not create like
that
            Console.Read();
        }
    }
}
```

31. Explicit Interface Implementations

Q). A class inherits from 2 interfaces and both the interfaces have the same method name.
How should the class implement the method for the both interfaces?

Example:-
Using system;

```
Class Program:i1,I2
{
  Static void Main()
   {
     Program p=new program();
     ((I1)p).InterfaceMethod();
     ((I2)p).InterfaceMethod();

   }
 Void I1.InterfaceMethod()
  {
    Console.WriteLine("I1 interface method implemented");
  }
 Void I2.InterfaceMethod()
  {
   Console.writeline("I2 interface method implementations");
  }

}
Interface I1
{
 Void InterfaceMethod();
}
Interface I2
{
Void InterfaceMethod();
}
```

#in above Program we are using explicit interface implementation technique to solve this problem
Note:- when a class explicitly implements, an interface member, the interface member can no longer be accessed thru class reference variable, but only thru the interface reference variable.

Access modifier are allowed on explicitly implemented interface members.

Default and Explicit Implementation:-
Note: if you want to make one of the interface method, the default, then implement teat method normally and the other interface method explicitly. This makes the default method
Explicitly. This makes the default method to be accessible thru class instance

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace _31_Explicit_Interface_Implementation
```

```
{
    interface I1
    {
        void InterfaceMethod();
    }
    interface I2
    {
        void InterfaceMethod();
    }
    class Program:I1,I2
    {
        public void InterfaceMethod()
        {
            Console.WriteLine("I1 interface Method");
        }
         void I1.InterfaceMethod()
        {
            Console.WriteLine("I1 interface Method");
        }
        static void Main(string[] args)
        {
            /*Program p = new Program();
            ((I1)p).InterfaceMethod();//explicit interface implementation
            ((I2)p).InterfaceMethod();
             */

            //we can also do like as below.
            I1 i1 = new Program();
            I2 i2 =new Program();

            i1.InterfaceMethod();
            i2.InterfaceMethod();

            Console.Read();
          //  p.InterfaceMethod();
            Console.Read();


            Program p = new Program();
            p.InterfaceMethod();
            Console.Read();
        }
    }
}
```

32.Abstract Classes

The abstract keyword is used to create abstract classes.

An abstract class is incomplete and hence cannot be instanited.
An abstract class can only be used a base class
An Abstract class cannot be sealed.
An Abstract class may contain abstract members(methods, properties, indexers and events) but not mandatory.
A non-abstract class derived form an abstract class must provide implementation for all inherited abstract members.
If a class inherits an abstract class, there are two options available for that classes
Option1:-provide implementation for all the abstract members inherited from the base class abstract class.
Option2:-If the class does not wish to provide the implementation for all the abstract member inherit form the abstract class, then the class has to be marked as abstract.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace _32_AbstractClasses
{
  public  abstract class Customer
   {
     //  public abstract void Print();
      public void Print()
      {
          Console.WriteLine("print");
      }


   }
   public  class  Program:Customer
    {
       public override void Print()
       {
           Console.WriteLine("Print Method");
       }
        static void Main(string[] args)
        {
            Customer c = new Program();
            c.Print();
            Console.Read();
        }
    }
}
```

33. Abstract Classes VS Interfaces

Abstract classes can have implementations for some of its members(Methods) but the interfaces can't have implementation for any of its member.

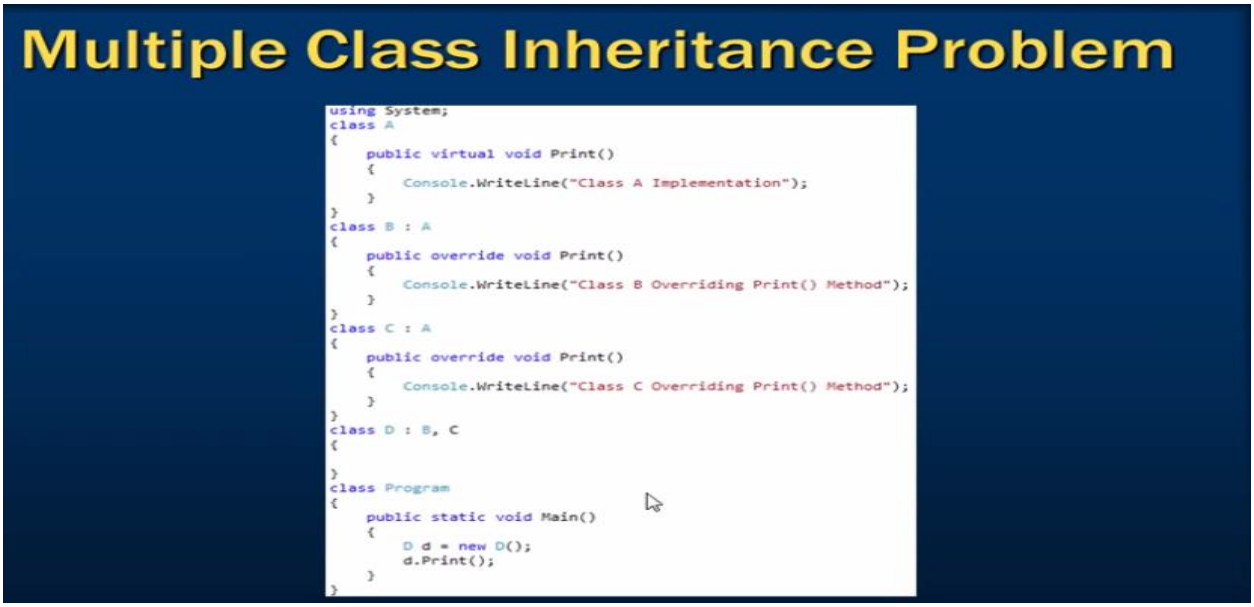
Interface cannot have fields but abstract can have fields

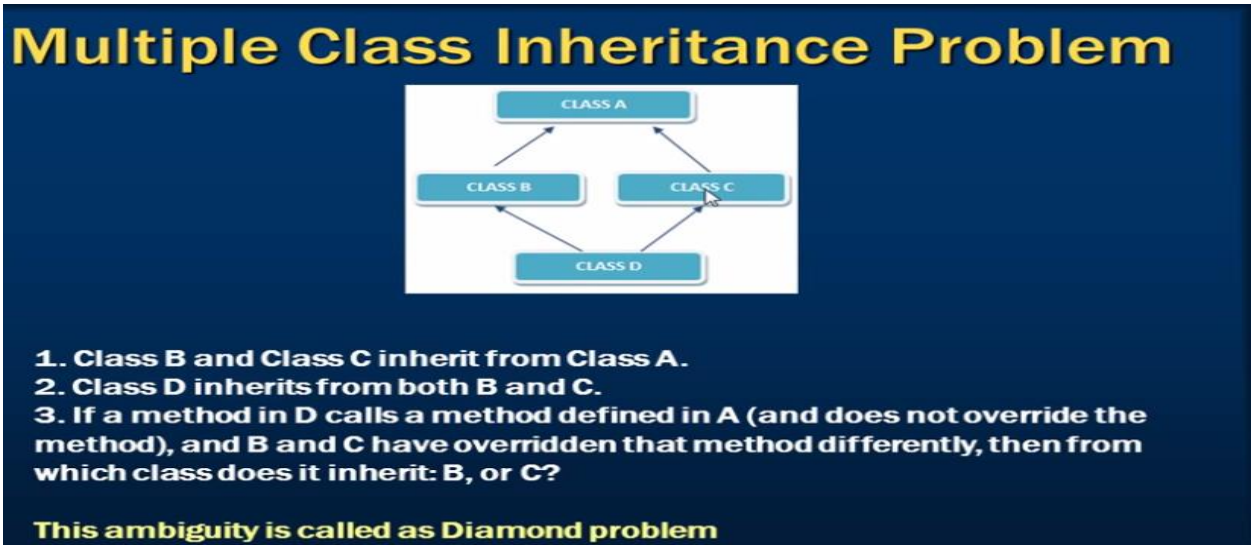
An interface can inherit from another interface only and cannot inherit form the abstract class; where as an abstract class can inherit from another abstract class and another interface.

A class can inherit form multiple interfaces at the same time, whereas class cannot inherit form multiple classes at the same time.

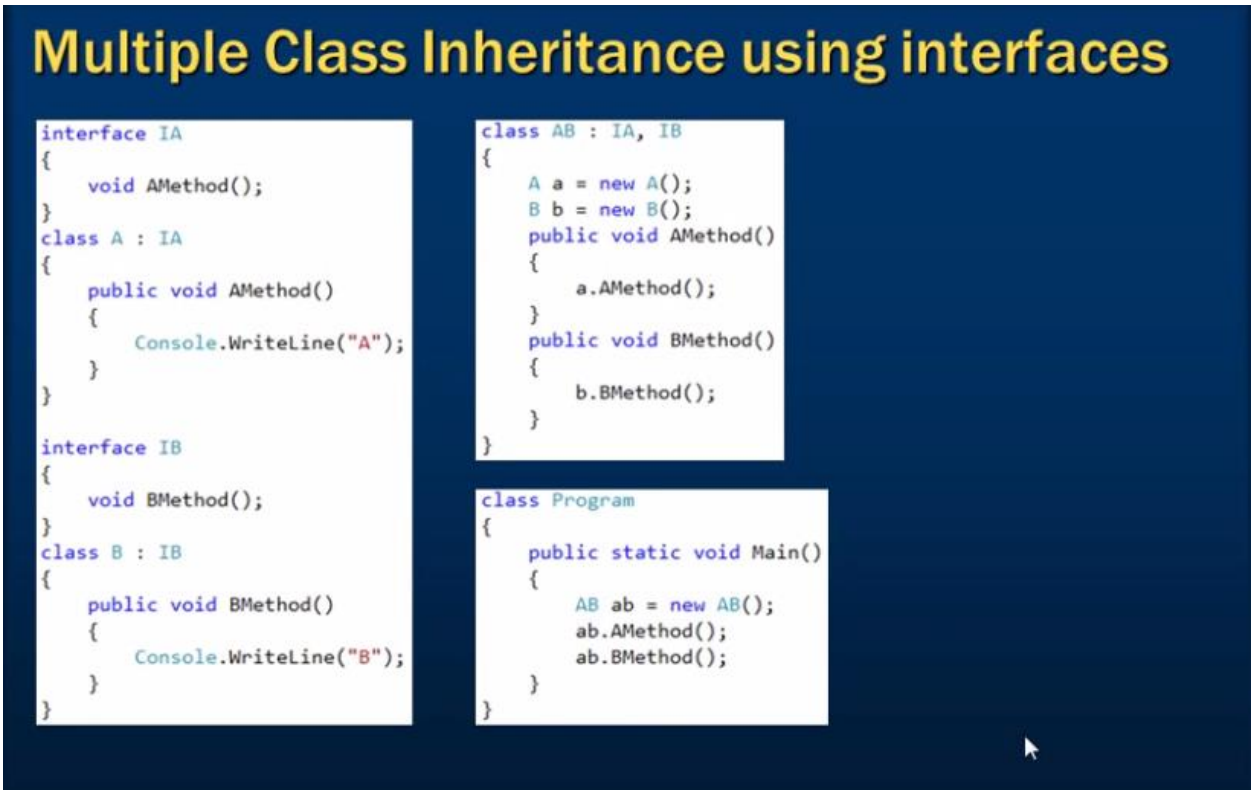
Abstract class member can have access modifiers whereas interface member cannot have access modifiers.

34. Problems of multiple classes Inheritance





35. Multiple Class Inheritance Using Interfaces




```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace _35_MultipleClass_Inheritance_using_Interfaces
{
    interface IA
    {
        void AMethod();
    }
    class A:IA
    {
        public void AMethod()
        {
            Console.WriteLine("A");
        }
    }
    interface IB
    {
        void BMethod();
    }
    class B : IB
    {
        public void BMethod()
```

```
        {
            Console.WriteLine("B");
        }
    }
    class AB:IA,IB
    {
        A a = new A();
        B b = new B();
        public void AMethod()
        {
            a.AMethod();
        }
        public void BMethod()
        {
            b.BMethod();
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            AB ab = new AB();
            ab.AMethod();
            ab.BMethod();
            Console.Read();
        }
    }
}
```

36. Delegates In C sharp

Q) What is Delegate?
A delegate is a type safe function pointer. That is, it holds holds a referemce (pointer) to a function.

The signature of the delegates must match the signature of the function, the delegate points to, otherwise you get the compiler error. This is the reason delegates are called as type safe function pointers.

A delegate is similar to a class. You can create an instance of it, and when you do so, you pass in the function name as a parameter to the delegate constructor, and it is to this function the delegate point to.

Tip to remember delegate syntax: Delegates syntax looks very much similar to a method with a delegate keyword.

```
using System;
using System.Collections.Generic;
using System.Linq;
```

```
using System.Text;
using System.Threading.Tasks;

namespace _36_Delegates_in_C_sharp
{
    public delegate void HelloFunctionDelegate(string Message);
    class Program
    {
        static void Main(string[] args)
        {
          //  HelloFunctionDelegate del = new HelloFunctionDelegate(Hello);
           // del("Hello From Delegates");

            //short form
            Hello("Hello from delegate");
            Console.Read();

            //A dekegate is a type safe function pointer
        }
        public static  void Hello(string strMessage)
        {
            Console.WriteLine(strMessage);

        }
    }
}
```

37.Delegates Uses in C sharp.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace _37_Delegates_Uses_in_C_sharp
{
    class Program
    {
        static void Main(string[] args)
        {
            List<Employee> emplist = new List<Employee>();
            emplist.Add(new Employee(){Id=101, Name="Mary",Salary=5000,
Experience=5});
            emplist.Add(new Employee(){ Id =101,Name = "Mike", Salary = 4000,
Experience = 4 });
            emplist.Add(new Employee() { Id = 101, Name = "John", Salary =
6000, Experience = 6 });
            emplist.Add(new Employee() { Id = 101, Name = "Tood", Salary =
5000, Experience = 3 });

            Employee.PromoteEmployee(emplist);
```

```
            Console.Read();
        }
    }
    class Employee
    {
        public int Id { get; set; }
        public string Name { get; set; }

        public int Salary { get; set; }
        public int Experience { get; set; }
       public static void PromoteEmployee(List<Employee>employeelist)
        {
           foreach(Employee employee in employeelist)
           {
               if (employee.Experience>=5)
               {
                   Console.WriteLine(employee.Name +  "Promoted");
               }

           }

        }
    }
}
```

Continue….

38. Delegates uses in C sharp continue……

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace _37_Delegates_Uses_in_C_sharp
{
    class Program
    {
        static void Main(string[] args)
        {
            List<Employee> emplist = new List<Employee>();
            emplist.Add(new Employee() { Id = 101, Name = "Mary", Salary =
5000, Experience = 5 });
            emplist.Add(new Employee() { Id = 101, Name = "Mike", Salary =
4000, Experience = 4 });
```

```
            emplist.Add(new Employee() { Id = 101, Name = "John", Salary =
6000, Experience = 6 });
            emplist.Add(new Employee() { Id = 101, Name = "Tood", Salary =
5000, Experience = 3 });
           // IsPromatable isPromotable = new IsPromatable(Promote);
            Employee.PromoteEmployee(emplist,emp=>emp.Experience>=5);
            Console.Read();
        }
       /* public  static bool Promote(Employee emp)
        {
            if (emp.Experience >= 5)
            {
                return true;
            }
            else
                return false;
        }
        */
    }
    delegate bool IsPromatable(Employee empl);
    class Employee
    {
        public int Id { get; set; }
        public string Name { get; set; }

        public int Salary { get; set; }
        public int Experience { get; set; }
        public static void PromoteEmployee(List<Employee> employeelist,
IsPromatable IsEligibletoPromote)
        {
            foreach (Employee employee in employeelist)
            {
               // if (employee.Experience >= 5)
                if(IsEligibletoPromote(employee))
                {
                    Console.WriteLine(employee.Name + "Promoted");
                }

            }

        }
    }
}
```

39. Multicast Delegates

Multicast Delegates:-

A multicast Delegate is a delegate that has reference to more than one function. When you invoke multicast delegates, all the functions the delegate is pointing to, are invoked.

There are 2 approaches to create a multicast delegate. Depending on the approach you use + or += to register a method with the delegate
- Or-= to un-register a method with the delegate

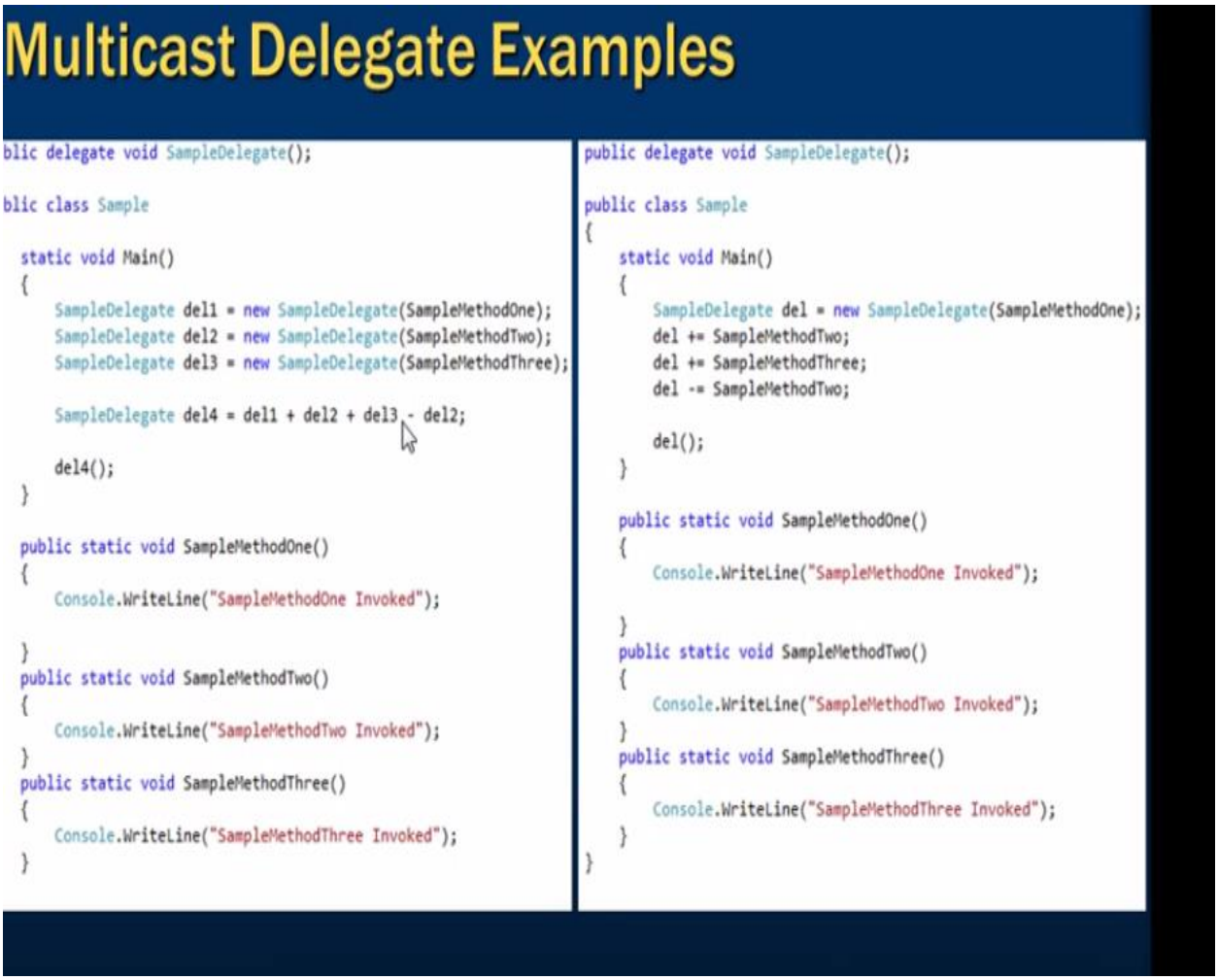
Note:-A multicast delegate, invokes the methods in the invocation list, in the same order in which they are added.

In the delegate has a return type other than void and if the delegate is a multicast delegate, only the value of the last invoked method will be returned. Along the same lines, if the delegate has an out parameter the value of the output parameter will be the value assigned by the last method

Common interview question- where do you multicast delegates?
Multicast delegate makes implementation of observer design pattern very simple. Observer pattern is also called as publish/subscribe pattern.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace _39_MultiCastDelegates_in_C_sharp
{
    //public delegate void sampleDelegate();
    public delegate void SampleDelegate(out int Integer);
    class Program
    {
        static void Main(string[] args)
        {

            //sampleDelegate del = new sampleDelegate(SampleMethodOne);
            //del();
           /* sampleDelegate del1, del2, del3, del4;
            del1 = new sampleDelegate(SampleMethodOne);
            del2 = new sampleDelegate(SampleMethodTwo);
            del3 = new sampleDelegate(SampleMethodThree);
            del4 = del1 + del2 + del3-del2;//multicast delegates
            del4();
            */



          /*  //short cut method
            sampleDelegate del = new sampleDelegate(SampleMethodOne);
            del += SampleMethodTwo;
```

```
            del += SampleMethodThree;
            del -= SampleMethodOne;
            del();
           * */


            SampleDelegate del = new SampleDelegate(SampleMthodOne);
            del += SampleMethodTwo;
            int OutpurtParameterVlaue = -1;
            del(out OutpurtParameterVlaue);
            Console.Write("Outputparametervalue={0}", OutpurtParameterVlaue);
            Console.Read();
        }
     /*   public static  void SampleMethodOne()
        {
            Console.WriteLine("Sample method One  invoked.");
        }
        public static void SampleMethodTwo()
        {
            Console.WriteLine("Sample method two invoked.");
        }
        public static void SampleMethodThree()
        {
            Console.WriteLine("Sample method three invoked.");
        }
      */
        public static void SampleMthodOne( out int Number)
        {
            Number = 1;

        }
        public static void SampleMethodTwo(out int Number)
        {
            Number = 2;
        }
    }
}
```



40. Exception Handling in C sharp

An exception is an unforeseen error that occurs when a program is running.
Examples:
Trying to read from a file that does not exist, throws FILENOTFOUNDEXCEPTION.
Trying to read fro a database table that does not exist, throws a SqlException.

Showing actual unhandled exceptions to the end user is bad for two reasons
1.User will be annoyed as they are cryptic and does not make much sense to the end users.
2. Exception contain information, that can be use for hacking into your application.

An exception is actually a class that derived from system. Exception class. They system.Exception class have several useful properties, that provide the valuable information about the exception.
Message: gets a message that describe the current exception

Stack Trace: provide the call stack to the line number in the method where the exception occurred.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.IO;

namespace _40_ExceptionHandling
{
    class Program
    {
        static void Main(string[] args)
        {
            StreamReader streamReadeer = null;
            try
            {
             streamReadeer = new StreamReader(@"C:\Sample
File\data.txt.txt");
            Console.Write(streamReadeer.ReadToEnd());
            Console.Read();

            }
            catch(FileNotFoundException ex)
            {
                /*
                Console.WriteLine(ex.Message);
                Console.WriteLine();
                Console.WriteLine();
                Console.WriteLine(ex.StackTrace);
                Console.Read();
                 */


                //log the details to the Db
                Console.WriteLine("Please chek if the file {0} exists or
not!", ex.FileName);
                Console.Read();

            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
                Console.Read();
            }
            finally
            {
```

```
                if(streamReadeer!=null)
                {
                streamReadeer.Close();
                }
                Console.WriteLine("Finally Block");
                Console.Read();
            }
        }
    }
}
```

41. Inner Exception

The Inner Exception property returns the exception instance that caused the current exception.

To retain the original exception pass it as a parameter to the constructor, of the current exception

Always chek if the inner exception is not null before accessing any property to the inner exception object, else you may get Null reference exception. To get the type of Inner Exception use GetType() method.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.IO;

namespace _41_Inner_Exception
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                try
                {
                    Console.WriteLine("Enter First Nuber");
                    int FN = Convert.ToInt32(Console.ReadLine());

                    Console.WriteLine("Enter Second Number");
                    int SN = Convert.ToInt32(Console.ReadLine());

                    int Result = FN / SN;
```

```
                    Console.WriteLine("Result={0}", Result);
                }
                catch (Exception ex)
                {
                    string filPath = @"C:\Sample File\log1.txt";
                    if (File.Exists(filPath))
                    {
                        StreamWriter sw = new StreamWriter(filPath);
                        sw.Write(ex.GetType().Name);
                        sw.WriteLine();
                        sw.Write(ex.Message);
                        sw.Close();
                        Console.WriteLine("There is problm, please try
later");

                    }
                    else
                    {
                        throw new FileNotFoundException(filPath + "ss not
presnet");
                    }

                }
            }
            catch(Exception exception)
            {
                Console.WriteLine("Current
Exception={0}",exception.GetType().Name);
               if(exception.InnerException !=null)
               {
                Console.WriteLine("curent Exception={0}",
exception.InnerException.GetType().Name);
               }
            }
            Console.Read();
        }
    }
}
```

42. Custom Exception in C sharp.

To understand the exceptions, you should have good understanding of

Part 21- Inheritance

Part40- Exception Handling Basics

Part 41:- Inner exception

When do you usually go for creation your own custom exceptions?

If none of the already existing detent exceptions adequately describes the problem.

Example:-

1. I have an asp.net web application
2. The application should allow the user to have only one logged in session.
3. If the user is already logged in, and if he opens another browser window and tries to login again, the application should throw an error stating he is already logged in another browser window.

With the dot.net framework we don't have any exception that adequately describes this problem. So this scenario is one of the examples where you want to create custom exception.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.IO;
using System.Runtime.Serialization;

namespace _42_Custom_Exception_in_C_Sharp
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
            throw new UserAlreadyLoggedInException("User is Loged in--no
dubplicate session allowed");
            }
            catch(UserAlreadyLoggedInException ex)
            {
                Console.WriteLine(ex.Message);
            }
            Console.Read();
        }
    }
```

```
   public class UserAlreadyLoggedInException:Exception
   {
       public UserAlreadyLoggedInException():base()
       {

       }
       public UserAlreadyLoggedInException(string message):base(message)
       {

       }
       public UserAlreadyLoggedInException(string Message,Exception
innerException):base(Message,innerException)
       {

       }
       public UserAlreadyLoggedInException(SerializationInfo
info,StreamingContext context):base(info,context)
       {

       }
   }
}
```

43. Exception Handling Abuse

Exceptions are unforeseen errors that occur when a program is running. For Example, when an application is executing a query, the database connection is lost. Exception handling is generally used to handle these scenarios.

Using exception handling to implement program logical flow is bad and is termed as exception handling abuse.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace _43_Exception_Handling_Abuse
{
    class Program
    {
        static void Main(string[] args)
        {
            try {
            Console.WriteLine("Please enter  Numberator");
            int Number = Convert.ToInt32(Console.ReadLine());


            Console.WriteLine("Please enter Denominator");
            int Denominator = Convert.ToInt32(Console.ReadLine());
```

```
            int Result = Number / Denominator;

            Console.WriteLine("Result={0}", Result);
                }
            catch (FormatException)
            {
                Console.WriteLine("Please enter a number");
            }
            catch(OverflowException)
            {
                Console.WriteLine("Only numbers between{0} && {1} are
allowed", Int32.MinValue, Int32.MaxValue);
            }
            catch(Exception ex)
            {
                Console.WriteLine(ex.Message);
            }
            Console.Read();

        }
    }
}
```

44. Preventing Exception Handling

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace _43_Exception_Handling_Abuse
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                Console.WriteLine("Please enter  Numberator");
                int Numerator;
                bool IsNumberatorConverstionSucessful =
Int32.TryParse(Console.ReadLine(), out Numerator);
                if (IsNumberatorConverstionSucessful)
                {
                    Console.WriteLine("Please enter Denominator");
                    int Donominator;
                    bool IsDenominatorConversionSUcesseful =
Int32.TryParse(Console.ReadLine(), out Donominator);
```

```
                    if (IsDenominatorConversionSUcesseful & Donominator != 0)
                    {
                        int Result = Numerator / Donominator;
                        Console.WriteLine("Result {0}", Result);
                    }
                    else
                    {
                        if (Donominator == 0)
                        {
                            Console.WriteLine("Denominator can not be zero");
                        }
                        else
                        {
                            Console.WriteLine("Denominator should be a valid
number between {0} && {1}", Int32.MinValue, Int32.MaxValue);
                        }

                    }
                }
                else
                {
                    Console.WriteLine("Denominator should be a valid number
between {0} && {1}", Int32.MinValue, Int32.MaxValue);
                }
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);

            }
            Console.Read();
        }
    }
}
```

45. Why Enums

Enums are strongly typed constants.
If the program uses of integral numbers, consider replacing them with enums. Otherwise the program becomes less Readable, Maintainable
In next session we will replace this integral numbers with enums, which makes the program better readable and maintainable.

```
using System;
using System.Collections.Generic;
```

```
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace _45_Why_Enums
{
    class Program
    {
        static void Main(string[] args)
        {
            Customer[] Customer = new Customer[3];
            Customer[0] = new Customer
            {
                Name = "Mark",
                Gender = 1
            };
            Customer[1] = new Customer
            {
                Name = "Marry",
                Gender = 2
            };
            Customer[2] = new Customer
            {
                Name = "Sam",
                Gender = 0

            };
            foreach (Customer customer  in Customer)
            {
                Console.WriteLine("name={0} && Gender={1}",
customer.Name,GetGender (customer.Gender));
            }
            Console.Read();
        }
        public static string GetGender(int gender)
        {
            switch(gender)
            {
                case 0:
                    return "Unknown";
                case 1:
                    return "Male";
                case 2:
                    return "Female";
                default:
                    return"Invalid data Detected";
            }
        }
    }
    public class Customer
```

```
    {
        public string Name { get; set; }
        public int Gender { get;set; }
    }
}
```

46. Enums Examples

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace _45_Why_Enums
{
    class Program
    {
        static void Main(string[] args)
        {
            Customer[] Customer = new Customer[3];
            Customer[0] = new Customer
            {
                Name = "Mark",
                Gender = gender.Male
            };
            Customer[1] = new Customer
            {
                Name = "Marry",
                Gender =gender.Female
            };
            Customer[2] = new Customer
            {
                Name = "Sam",
                Gender = gender.Unknown
            };
            foreach (Customer customer in Customer)
            {
                Console.WriteLine("name={0} && Gender={1}", customer.Name,
GetGender(customer.Gender));
            }
            Console.Read();
        }
        public static string GetGender(gender gender)
        {
            switch (gender)
            {
                case gender.Unknown:
```

```
                    return "Unknown";
                case gender.Male:
                    return "Male";
                case gender.Female:
                    return "Female";
                default:
                    return "Invalid data Detected";
            }
        }
    }
    public enum gender
    {
        Unknown,
        Male,
        Female
    }
    public class Customer
    {
        public string Name { get; set; }
        public gender Gender { get; set; }
    }
}
```

47. Enums in C sharp

Enums:-
If the programme uses set of integral numbers, consider replacing them with enums, which makes the program more
Readable
Maintainable

1.Enums are enumerations.
2. Enums are strongly type constants. Hence an explicit cast is needed to convert from enum type to an integral type and vice versa. Also enum of one type cannot be implicitly assign to an enum of another type even through underline value of their member are the same.
3. The default underline type of enum is int
4. The default value of first element is zero and gets increased by 1.
5. It is possible to customize the underline type and values.
6. Enums are value types
7.enums keyword (all small letters) is used to create the enumerations, whereas Enum class contain static GetVlaues() and GetNames() method which can be used to list enum underlying type values and names.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

```
using System.Threading.Tasks;

namespace _47_Enums_In_C_Sharp
{
    class Program
    {
        static void Main(string[] args)
        {
            /*
            short[] values = (short[])Enum.GetValues(typeof(Gender));
            foreach(short value in values)
            {
                Console.WriteLine(value);
            }
            string[] Names=Enum.GetNames(typeof(Gender));
            foreach(string Name in Names)
            {
                Console.WriteLine(Name);
            }
            Console.Read();
             */

           /* Gender gender = (Gender)3;
            int Num =(int) Gender.Unknown;
            */


            Gender gender=(Gender) Season.winter;


        }
    }
    public enum Gender:short
    {
        Unknown,
        Male,
        Female
    }
    public enum Season
    {
        winter=1,
        sprint=2,
        summer=3
    }
}
```

48. Difference Between Types and type member

In this example Customer is the Type and Fields, Properties and method are type members.

So, In General classes, structs, enums, Interfaces, delegates are called as types and fields, properties, constructors, methods etc. , that normally reside in a type are called as type members.

In C # there are 5 different access modifiers:
1.Private
2. Protected.
3.Internal
4.Protected Internal
5. Public

Types members can have all the access modifiers where type can have only 2 (internal, public) of the 5 access modifiers.
Note:-Using reagions you can expand all collapse sections of your code either manually or susing visual studio Edit->Outling->Toggle All Outlinig

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace _48_Difference_Between_Types_and_Type_members
{
    class Program
    {
        # region Fields
        private int _id;
        private string  _firstName;
        private string  _lastName;
        #endregion

        #region Properties
        public int Id
        {
            get { return _id; }
            set { _id = value; }
        }
        public string LastName
        {
            get { return _firstName; }
            set { _firstName = value; }
```

```
        }
        #endregion

        #region MEthod
        public string GetFullName()
        {
            return this._firstName + "  " + this._lastName;
        }
        #endregion

        static void Main(string[] args)
        {

        }
    }
}
```

49. Access Modifiers in C sharp.

There are 5 different access modifiers in C sharp.

1. Private
2. Protected
3. Internal
4. Protected internal
5. Public

Private member are available only within in the containing type, whereas the public member available anywhere. There is no restriction.

Protected member are available, within the containing type and to the type that derived from the containing type.

Access Modifier	**Accessibility**
Private	Only with the containing Class
Public	Anywhere No restriction
Protected	With the containing type and Type Derived from containing type

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace _49_Access_Modifiers_in_C_sharp
{
```

```
    public class Customer
    {
        /*
        private int _id;
        public int ID
        {
            get
            {
                return _id;
            }
            set
            {
                _id = value;
            }
        }
         */
        protected int ID;

    }
    public class corporateCustomer:Customer
    {
        public void printId()
        {
            corporateCustomer CC = new corporateCustomer();
            CC.ID = 101;
            base.ID = 102;
            this.ID = 103;
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            /*
            Customer c1 = new Customer();
            Console.WriteLine(c1.ID);
             */
        }
    }
}
```

50. Internal and Protected Internal Access Modifiers.

A member with internal access modifier is available any where with the containing assembly. It's a compile time error to access, an internal member from outside the containing assembly.

Protected internal members can be accessed by any code in the assembly in which it is declared, or from with a derived class in another assembly. It is the combination of protected and internal. If you have understood protected an internal , this should be very very easy to follow.

Access Modifiers	**Accessibility**
Internal	Anywhere with the containing assembly
Protected internal	anywhere with the containing assembly And,from within the derived class in any other assembly.

```
using System;
namespace Assembley1
{
    public class AssemblyOneClasssI
    {
       protected internal int ID=101;
    }
    public class AssemblyOneClassII
    {
        public void SampleMethod()
        {
            AssemblyOneClasssI A1 = new AssemblyOneClasssI();
          Console.WriteLine(A1.ID);
          Console.Read();
        }
    }

}
```

```
using System;
using Assembley1;
namespace Assembly2
{
    public class AssemblyTwoClassI:AssemblyOneClasssI
    {
        public void print()
        {

            AssemblyOneClasssI A1 = new AssemblyOneClasssI();
            base.ID = 101;
            AssemblyTwoClassI A2 = new AssemblyTwoClassI();
            A2.ID = 101;

        }
```

```
    }
}
```