# .NET FRAMEWORK.

01/07/24.

Q) What is .Net Represent?

* .Net stands for "Network Enabled technology".
* In .Net (.) refers to object oriented & Net refers to the Internet.
* So the complete .Net means through object-oriented, we can implement internet based applications.

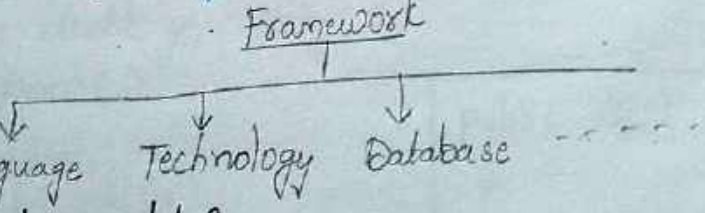
* It is a Software technology which is introduced by Microsoft in 2002.

What is Framework?

A framework is a Software (or)
A framework is a collection of many small technologies integrated together to develop applications i.e., executed anywhere.

* .Net frame is a framework technology which is integrated with multiple technologies.

ASP → Server side.
Javascript → Client side.




What is Not .Net?

1. Not a Operating System.
2. Not a application or package.
3. Not a database.
4. Not a ERP application.
5. Not a testing tool.
6. Not a programming language.

Then what Exactly .Net ?

•Net is a framework tool that supports many programming languages & many technologies.

* .Net supports 16+ programming languages.

In 60+ languages — 9 languages designed by Microsoft & remaining designed by non-microsoft.

Languages:

* Microsoft designed programming languages as follows.
  1) VB.Net [Visual basic 6.0] → Windows based applications.
  2) C#.Net [Web applications, mobile, ---]
  3) VC++.Net [Visual C++] → Gaming applications.
  4) J#.Net (Java)
  5) F#.Net.
  6) JScript.Net.
  7) Windows power shell
  8) Iron python.
  9) Iron Ruby.

Technologies:

The .Net supporting technologies are:

1) ASP.Net (Active Server Pages.Net)
2) ADO.Net (Active Data object.Net) → Connecting database.
3) WCF (Windows communication foundation)
4) WPF (Windows presentation foundation)
5) AJAX (Asynchronous Javascript and XML)
6) LINQ (Language Integrated Query)

etc.

(8) Web service.

WebAPI [Ex: Google pay]

↳ Angular, C#

* To develop any type of application using .Net we require One .Net technology & one .Net language.

Ex: ① If you want to develop windows application we have use Window Forms & C#.Net.

② If you want to develop web application, we have to use ASP.Net & C#.Net.

③ If you want to develop web services, we have to use WCF & C#.Net.

.Net Software:

.Net Software is avaliable as 2 products.

1. Visual Studio.Net.
2. .Net Framework.

1) Purpose of Visual Studio.Net is to make application development easy and fast by providing GUI (Graphical user interface).

Note: We can develop .Net application using text editors like Notepad, editplus ---- (For this in our system, we require framework of .Net).

2) .Net framework is different from Visual Studio.

* .Net framework is the "Engine" on which the .Net applications Run.
* .Net application cannot run on any machine unless it has .Net framework installed.

Version:

1st Version: C# 1.0 (language)

.Net framework : 1.0/1.1

Visual Studio : .Net 2002

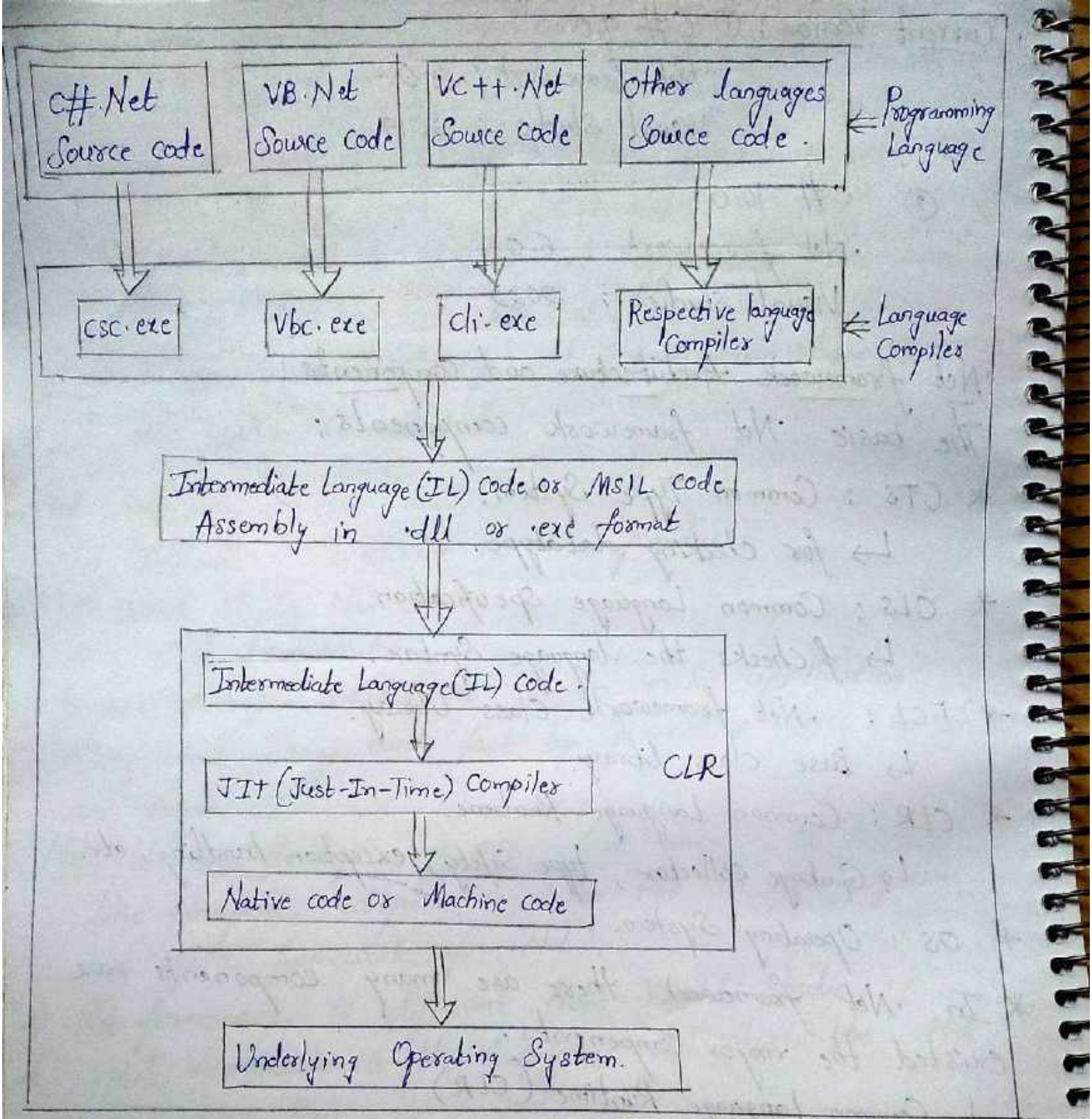
Current Version : ① C# 9.0
.Net framework : 5.0
Visual Studio : 2019

② C# 10.0
.Net framework : 6.0
Visual Studio : 2022.

.Net framework Architecture and Components :
The basic .Net framework components :

* CTS : Common Type System.
  ↳ for checking datatype.
* CLS : Common Language Specification.
  ↳ f checks the language syntax.
* FCL : .Net framework Class Library.
  ↳ Base class library.
* CLR : Common Language Runtime.
  ↳ Garbage collector, type safety, exception handling, etc.
* OS : Operating System.
* In .Net framework these are many components are existed the major components.
  1. Common Language Runtime (CLR).
  2. Class Library (Base class library).

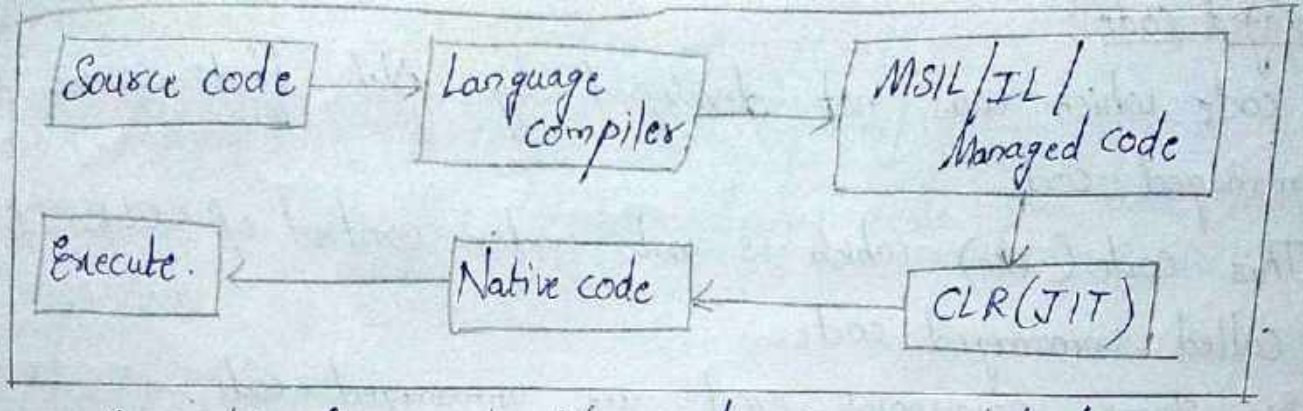
1. Common Language Runtime is the "Execution Engine" that handles running application.
* It provides services like garbage collections, type safety exception handling, etc.




2. Base class Library provides set of application interface & types of common functionality it provides strings, dates, numbers, etc.

* This are avaliable

C:\Windows\assembly\

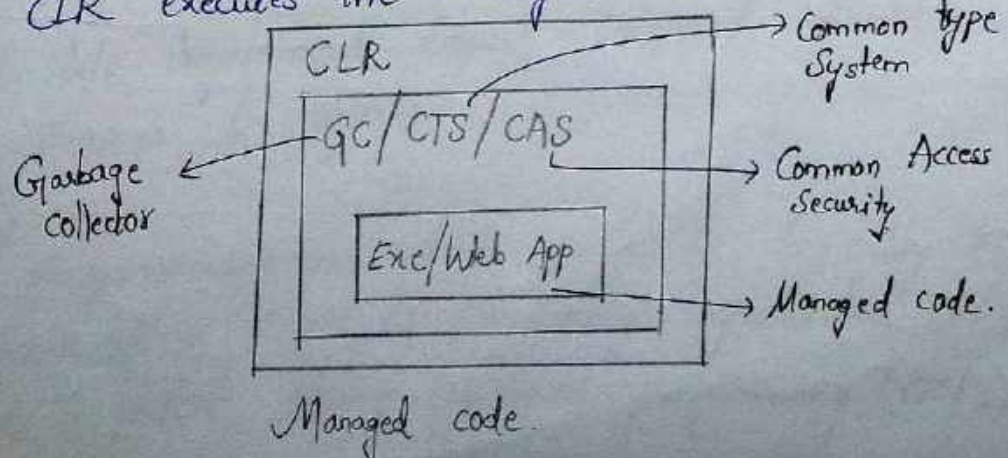



* In the .Net framework, the code is compiled twice.
  1. In the 1st compilation, the source code is compiled by the respective language compilers & generates the intermediate code which is known as MSIL (Microsoft Intermediate Language) or IL (Intermediate Language) or Managed code.
  2. In the 2nd compilation, MSIL is converted into Native code (Native code means code specific to the operating System so that the code is executed by the operating System) and this is done by the CLR.
* Always the 1st compilation is slow & 2nd compilation is fast.

Managed code:

The code (exe file) which runs under the control of CLR is called Managed code.

* CLR consists Garbage collector (GC), code Access Security (CAS), Common type System (CTS).
* CLR executes the managed code.



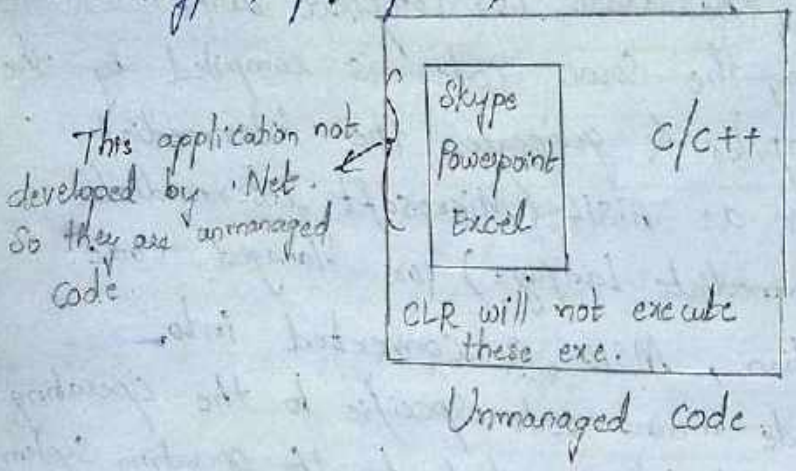

Managed code.

Unmanaged code:

The code which was not developed by .Net that comes to unmanaged code.

* This code (.exe) which is not under control of CLR is called unmanaged code.

Ex: Skype, powerpoint, excel are unmanaged code.



# OOPs

02/07/24

OOPs [Object Oriented programming System]

* OOPs is a concept (Concept always tells you rules and regulations). by Grady Boach.

* Paradigm (Pattern):
  1. Monolithical paradigm Ex: basics.
  2. Procedural Ex: C, cobol, Fortran.
  3. Modular Ex: C
  4. Object-Oriented Ex: C++, java, C#.

What is the need of OOPs:

1. Reusability → Data Re-usability [saves memory] → reuse source code
2. Extensibility → For reusability purpose
3. Modularity → Source code can be divided into diff modules (functionality)
4. Security → we require security for the sake of code reusability.
5. Simplicity } → achieved by polymorphism.
6. Efficiency }

* The above 6 reasons can be achieved by features are
  1. Encapsulation.
  2. Abstraction.
  3. Inheritance.
  4. Polymorphism.

Note: We implement Object-Oriented functionality using classes & objects.

OOPs:

The programming in which data is logically represented in the form of a class & physically represented in the form of Object is called Object-Oriented programming (OOPs).

Application:

* Programming languages & technologies are used to develop application.
* Application is a collection of programs.
* We need to design and undesstand a single program before developing an application.

Program Elements: Program is a set of Instructions, Every program consist of Identity, Variables, methods.

* "Identity" of a program is Unique.
* Programs, Classes, Variables & methods having identity.

2. Variable is an identity given to memory location.
3. Method is a block of instructions with an identity.
   * Method performs operations on data (variables)
   * Method takes input data, perform operations on data & returns result.

Syntax of method:

```
return_type method_name(arguements)
{
  body;
}
```

Ex:

```
int add(int a, int b)
{
  int c=a+b;
  return c;
}
```

* C#.Net is Object oriented programming language.
* OOPs is a concept of defining objects & establish communication between them.

**Object:** Physical & entity.

Object is everywhere and everything.

**Nature of objects:** An object must be **self initialize** & **self executable**. i.e., no one not to instruct an object.

* An object must have a conceptual boundary (i.e., it nature is not to be fixed) has to changeable depends on usage.

**Ex:** Myself in classroom as a student,
Hotel — Customer,
Home — Family member.

* In Real-life object has entities

1. State → properties or variables
2. Behaviour → Methods
3. Identity → Name of object.

if satisfy that entity is called Object..

* In real life "**state**" means properties or attributes or characteristics are nothing but physical variables.
These are 2 types 1. Visible
2. Unvisible.

* By using properties we can able to provide the actions, these actions are known as "**Behaviour**". [Methods]
→ Depending on the **state corresponding behavior** is existing where there is no state, there is no behavior.
→ State and behavior both are **interdependent**.

* "**Identity**" means a name is given in order to identify the object.

## Class: Logical Entity

→ Class is a logical entity or declarative entity.
(or)

→ Class is a model from which we can define multiple objects of same type.
(or)

→ Class is a blueprint
(or)

→ Class is a combination of variables & methods.

Syntax of class:

```
class class_name
{
    Variables ; (properties)
    methods ;
}
```

Syntax of object:

```
class_name reference = new class_name();
```

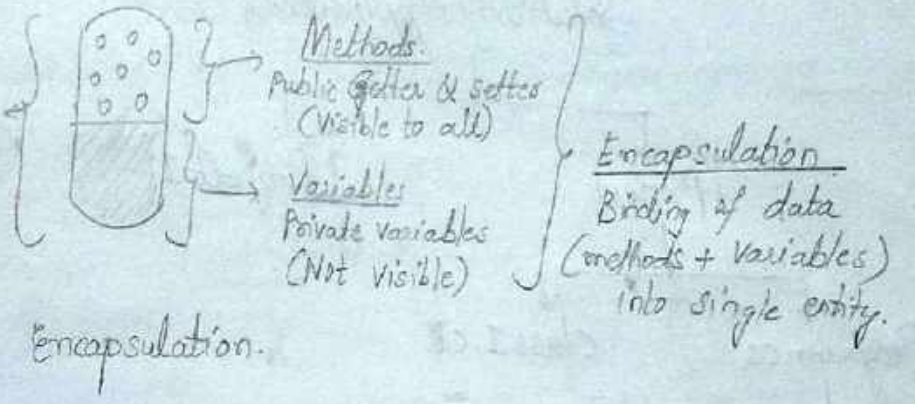
reference → object name

Ex: Account acc = new Account(); //instance created (memory allocated)

## Encapsulation:

Wrapping of variables (properties) and methods in a single unit is called Encapsulation.

* Designing a class itself is an Encapsulation.
* The advantage of Encapsulation:
  1) Data abstraction.
  2) Data binding.
     * By default class – internal.
     * Inside class → variables – private.
     * By methods → communicate




Encapsulation.

Data Abstraction:

Showing the essential details without showing the background details is called as Data abstraction.

* This is practically can be implemented in class by using Access Modifiers or specifiers.

* By default class properties (variables & methods) are private i.e., they are abstracted and not accessible.

Access Modifiers/Access specifiers:

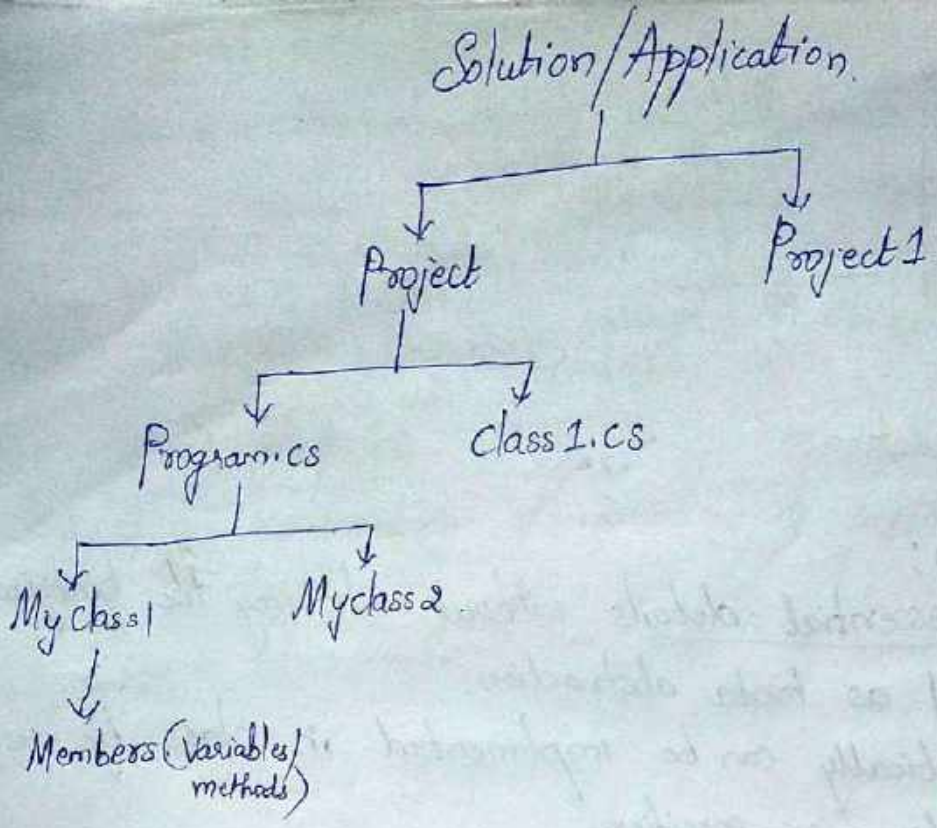
They are specifying accessibility or access level of a class or class members.

* Using this specifiers, a class or members we can define accessibility by the specifiers. they are

1. Private
2. Public
3. protected
4. Internal.
5. protected internal.
6. private protected

* Generally, variables are private (other objects cannot access the data directly).

Methods are public (to send and receive the data).




① Public :

* If we declare a class or class member as public we can access by all the classes of current project and all other projects of application.
* To access a public member there is no restriction ~~within~~ the to the application.
* Generally, we donot define public data since data requires ~~Segui~~ security. If we define public methods.

Ex:

```
using System;
namespace Example
{
 class Student
   {
public int sid;
    public void method1()
      {
        Console.Write(a);
      }
}
class Myclass
  {
  public static void Main()
    {
      Student obj = new Student();
      obj.Sid = 10;  → we can access variable as it is public
      obj.method1();  → we can access the method as we gave access modifier as public.
      Console.Read();
    }
}
```

② private:

* private members of a class properties can access only within the class.

Ex:

```
using System;
namespace MySpace
{
 class Test
 {
    int sid; // by default private to access within class
    public void method1()
    {
      Console.Write(a);
    }
 }
```

```
class Prime
{
 public static void Main()
 {
   Test obj = new Test();
 // obj.a = 10 //error.
   obj.method1(); // as it is public can access
   Console.Read();
 }
}
```

③ protected:

protected is similar to private when we defining in a single class.

* The importance of protected will takes place if you derive the class
* The protected data can be accessed by current class members as well as derived class members.

Ex:

usi

Ex:

```
using System;
namespace MySpace
{
① class Test Class1 // base class
  {
    protected int a=10; // variable declared & initialized
    public void method1()
    {
      Console.WriteLine("Class1 value of a:" + a);
    }
  }

② class Class2 : Class1 // derived class
  {
    a=23;
    public void method2()
    {
      a=23; // declared directly (memory re-used)
      Console.WriteLine(Class Method2 Value of a:" + a);
    }
  }

③ class Prime
  {
    public static void Main()
    {
      class1 obj1 = new class1();
      obj1.method1(); //10
      class2.obj2 = new class2();
      obj2.method2(); //23 → Here, the a=10 in class 1 is not replaced by a=23 in derived class. only using data memory [4 bytes] int
      obj2.method1(); //23.

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

* In derived class, we can use the services of base class but not to disturb the services of base class.

④ internal: If we declare class or class members access modify as Internal which can be accessed by all the classes of current project.

Ex:

```
using System;
namespace MySpace
{
① class class1
  {
    internal int a=10;
    public void method2()
    {
    Console.WriteLine("Method1 values
      is :" + a);
    }
  }

② class class2
  {
  public void method2()
  {
  Con
  class1 c1 = new class1();
  Console.WriteLine("Method2
   Value is :" + c1.a);
  }
}

③ class prime
  {
   public static void Main()
    {
    class1 obj1 = new class1();
    Console.WriteLine("Main method :" + obj1.a);

    class2 obj2= new class2();
     obj2.method2();

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

| Specifier | Same Assembly/Project | | | Other Assembly/Project | |
|---|---|---|---|---|---|
| | Declared class | Other classes | Derived classes | Other classes | Derived classes |
| private | Yes | No | No | No | No. |
| public | Yes | Yes | Yes | Yes | Yes. |
| protected | Yes | No | Yes | No | ~~No~~ Yes |
| internal | Yes | Yes | Yes | No | No |

Points to remember:

* default access modifier of class is internal.
* default access modifier of method is private.
* default access modifier of variables/properties is private.
* default access modifier of constructor is private.
* Finally default access modifier of class member ~~is~~ will be private.

Why static constructor cannot contain access specifier?

* We don't require to access static constructor they are automatic invoke, so we don't require any access specifier or object.

To add one project inside the other project:

* To work in other assemblies
  → Right on solution explorer
  ↳ click on add
  → select new project → select Class library.

In class library

```
using System;
namespace AccessSpecifier
  {
    public class class1
      {
        private int a;
        public int b;
        protected int c;
        internal int d;
        private protected int e;
        protected internal int d;
      }
}
```

* After writing the access specifier,
  ↳ right click on project name → select Build.
* To access that data (variables) [a, b, c, d, e, f) in your Sample project add the '.dll file' in your project.
  ↳ dynamic link library
# Application: new Console Application [sample]
*
* Right click on Console... 'Sample' in solution Explorer
Add ~~right click~~ Project reference Right click → Add references
  ⇒ ↳ Click on Browse → Select DLL file.

In Sample project [Console Application]

```
using System;
using AccessSpecifier;
namespace MySpace
  {
```

```
class Program : class 1
{
    static void Main(string[] args)
    {
        Program obj = new Program();
        // Console.WriteLine(obj.a); private
        Console.WriteLine(obj.b); public
        Console.WriteLine(obj.c); // protected [Program is a derived class of class 1]
        // Console.WriteLine(obj.d); internal
        // Console.WriteLine(obj.e); private protected
        Console.WriteLine(obj.f); // protected internal.
                                  ↳ accessible in other assemblies.
        Console.Read();
    }
}
}
```

Instance methods:

Syntax:

```
return_type access_modifier Method_name (arg1, arg2, ..)
{
    logic;
}
```

* Methods can be in 4 ways:
1. Method with no arguement and no return value.
2. Method with arguement and no return value.
3. Method with arguement and return value.
4. Method with no arguements and return value.

* Generally, class we use instance members and instance methods.
* Instance members (variables) related to object.
* We invoke instance members using object - address.
* Return Statement can return only 1 value.

Ex:

```
using System;
namespace Myspace
{
  class Test
  {
    internal int Sum(int x, int y)
    {
      return x+y; //returns only single value
    }
    internal void print()
    {
      for(int i=1 ; i<=40 ; i++)
        Console.Write("-");
      Console.WriteLine();
    }
    internal void swap(int x, int y)
    {
      int t;
      t=x;
      x=y;
      y=t;
      Console.WriteLine("After swapping of x:"+x+
        "and y:"+y);
    }
  class Program
  {
```

return type

2 integer i/p parameters.

```
public static void Main(String[] args)
{
    Test obj = new Test(); //object instance creation
    obj.print(); //invoke print method
    obj.Sum(23, 45); //invoke Sum method with arguements
    obj.print(); //invoke print method
    obj.swap(5, 10); //invoke swap method with arguements
    Console.ReadLine();
}
}
}
```

Pre-defined instance methods:

1. ToString() → to convert to string type.
2. GetHashCode() → to get address (reference) of object.
3. GetType() → to get the datatype of variable.
4. Equals() → to check whether to User data is equal to actual data.

Ex: if(pin == 4324) (or) if(pin.Equals(4324))

Pre-defined static methods of Static class [Console]:

1. WriteLine()
2. ReadLine()
3. Read()
4. Write()
5. ReadKey().

4/06/24

Ex:

```
using System;
namespace ConsoleAppl
{
  class Student
  {
    uint sid;
    String sname;
    double fees;
    internal void accept()
    {
      Console.Write("enter sid: ");
      this.sid = uint.Parse(Console.Readline());
      Console.Write("Enter sname: ");
      this.sname = Console.Readline();
      Console.Write("Enter fees: ");
      this.fees = Convert.ToDouble(Console.Readline());
    }
    internal void display()
    {
      Console.Write("object reference: " + this.GetHashCode());
      Console.WriteLine("\n sid: " + this.sid);
      Console.WriteLine("sname: " + sname);
      Console.WriteLine("Fees: " + fees);
    }
  }
  Class Program
  {
```

when object is created i.e., the reference of obj will be read by "this"

to get the address of the object

```
public static void Main(String[] s)
  {
    Student sathya = new Student(); //instance Creation
    Console.WriteLine(sathya.GetHashCode()); //display the reference(address) of object sathya.
    sathya.accept(); //invoke
    sathya.display(); //invoke

    Student Hema = new Student(); //instance created
    Console.WriteLine(Hema.GetHashCode());
    Hema.accept(); //invoke accept method.
    Hema.display(); //invoke display method.
    Console.ReadLine();
  }
 }
}
```

* "this" is a keyword which is representing the current class instance (object).
  ↳ memory.

Q) When we will use 'this' keyword ?
Whenever we want to access the current class instance members within the class by using the object, we can use 'this' keyword.

this
↳ nothing but pointer

Syntax: `this.<instance member> ;`

Ex: `this.sid ;`

* It is also called default object reference variable.
* it holds ↓ Object Address.
current
* this doesnot work in static ~~member~~ method [becoz obj not required], only works in instance method (obj required).

Ex:
```
class Sample
{
int a,b;
internal void ~~test~~ set (int a, int b)
{
this.a = a;
this.b = b; // b=b; Confusion which is parameter
}
internal void print()
{
Console.WriteLine("Object reference : " + obj.GetHashCode());
Console.WriteLine("A : " +this.a);
Console.WriteLine("B : " + this.b);
}
}
class Program
{
public static void Main()
{
Sample obj = new Sample();
obj.set(10,20);
obj.print();
}
}
}
```

formal parameters
parameter
Class members (instance variable)
Parameter
'b' is assigned to the instance variable 'b'
to overcome this use 'this' for instance variable [this.a]
actual parameter
call by value

C#.Net will support 3 types of possibilities of "Passing parameters".

1) call-by-value. (Pass by value) //default.
2) call-by-reference. (call by ref)
3) call-by-out (pass by out).

① Call by value:

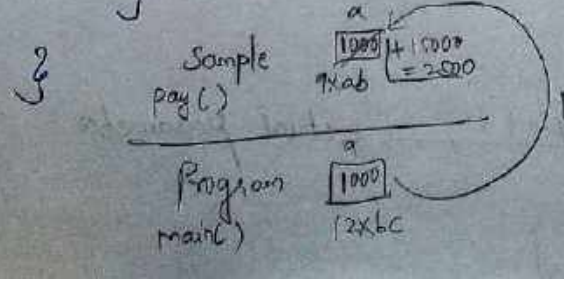
* In call by value, when formal parameter are modified that modification will not be reflected to the actual parameter.
* We are just passing the values from actual arguement.
  * receiving values - formal parameters.
  * Sending values - Original parameter
    ↓
    constants, expressions, variable.

Ex:

```
using System;
namespace MySpace
{
  Class Sample
  {
    internal void fees(int a)
    {
      a=a+1500;
      Console.WriteLine("A value in Sample class: "+ a);//2500
    }
  }
  class Program
  {
    public static void Main()
    {
      Sample s = new Sample();
      int a = 1000;
      s.fees(a);
      Console.Write("A value in Program class: "+ a);//1000
      Console.Read();
    }
  }
}
```

formal parameter

actual parameter



2) Call by ref:

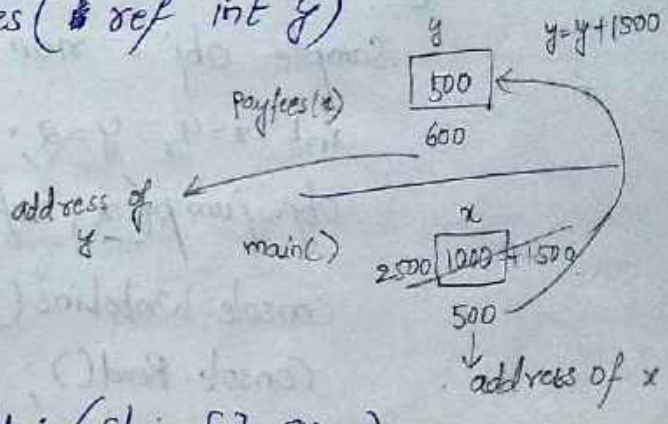
In call by ref, reference of the variable will pass from actual parameters to formal parameters and changes made in formal parameter it reflects to the actual parameters.

* At the time of actual parameter & actu formal parameter use 'ref' keyword.

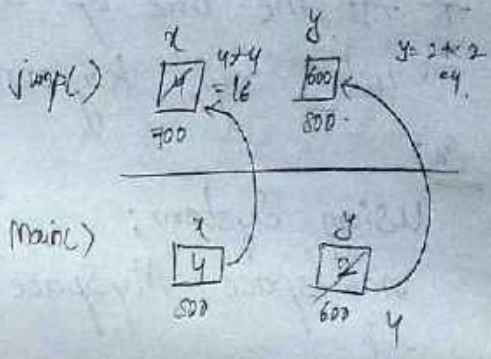
Ex:

```
using System;
namespace MySpace
{
  class Sample
  {
    internal void payfees(ref int y)
    {
      y = y + 1500;
    }
  }
  class Program
  {
    public static void Main(String[] args)
    {
      Sample s = new Sample();
      int x = 1000;
      Console.WriteLine("x Value in Program class before
                         calling: " + x); //1000
      s.payfees(ref x); actual parameter
      Console.WriteLine("X value in Program class after
                         Calling: " + y); //2500.
    }
  }
}
```

formal parameter.



② using System;
namespace MySpace
{
class Sample
{
Internal void jump(int x, ref int y)
{
x = x * x;
y = y * y;
Console.WriteLine(x + " " + y); // 16 16
}
}
class Program
{
public static void Main(string[] args)
{
Sample obj = new Sample();
int x=4, y=2;
obj.jump(x, ref y);
Console.WriteLine(x + " " + y); // 4 4
Console.Read();
}
}
}




③ Call by out:
Call by out is same as call by references or in call by out also when formal parameters changes is modified these modifications will be reflected back to actual parameters.
* while passing the parameter, while catching the parameter

we have to use out keyword.

* In call by out also with the help of parameters we will pass by the address.

Note: out parameter is not required to initialize, if we initialize the CLR will ignore that value.

Q) When to go call by out:

Whenever we don't want to pass some value to function but we want to get back the modified value we will go for call by out.

Ex:

```
using System;
namespace MySpace
{
  Class OutParameters
  {
    public static int Calculate (int m1, int m2, int m3,
                    out double avg, out string Grade)
    {
      int T = m1+m2+m3;
      avg = T/3.0;
      if(m1<35 || m2<35 || m3<35)
        {
          Grade = "Fail";
        }
      else
        {
          if(avg>=90)
            Console.Write(
              Grade = "Distinction";
```

Formal parameters (value type)

Formal parameters (out type = ref type)

As avg & Grade will return value rather than accept input.

```
        else if (avg >= 70)
            Grade = "First class";
        else if (avg >= 60)
            Grade = "Second class";
        else
            Grade = "Third class";
    }
    return T;
}
public static void Main(String[] args)
{
    int CSharp, UI, ASP, total;
double avg;
    Console.Write("Enter CSharp marks: ");
    CSharp = int.Parse(Console.ReadLine());
    Console.Write("Enter UI marks: ");
    UI = int.Parse(Console.ReadLine());
    Console.Write("Enter ASP marks: ");
    ASP = int.Parse(Console.ReadLine());
    total = Calculate(cSharp, UI, ASP, out avg, out grade);
                                    (B)
    total = OutParameters.Calculate(cSharp, UI, ASP,
                                    out avg, out grade);
    Console.WriteLine("Total: " + total);
    Console.WriteLine("Average: " + avg);
    Console.WriteLine("Grade: " + grade);
    Console.Read();
}
}
```

return value integer

actual parameters.

class name — methodName

Within same class, object creation not required, to call method directly call method name with parameters.

05/06/2

Constructor :

As per Object-oriented, every class should require atlea one constructor.

* If it is not available, as C# an object oriented langu it will create "default" constructor.
* Constructor provides dynamic behavior for an object whereas method cannot.
* Constructor is a special method.
* When object created (instance) automatically it invokes the constructor but not a method.
  ↳ has call explicitly by obj name.

Purpose:

Constructor provides dynamic behavior for an object.

Rules:

* Constructor name & className must be same.
* Constructor not allow any return type.
* Constructor can contain arguements.

* Constructors can be 1. Default constructor.
  2. Parameterized Constructor
  3. Copy Constructor.

Variable: Variable Stores the information of class (object).

As in OOPs we define variables different types.

1. Static Variables: Store common information of all objects. These variables access by class_name. These are common for every object. Ex: ClassName.Static_Variable.
2. Instance variables: Store specific information of specific object, Access instance using object reference.

   Ex: ObjectName.instance_variable.

3. Method parameters: Takes input in a method. Access method parameter directly and only inside the method.

4. Local variables: Store processed information inside the method. Access directly and only inside method.

Ex:
```
Class Employee
{
    Static String company = "My Company";   } Static
    Static String address = "Hyderabad";    } Variable.

    int empid;
    String empname;     } instance
    double empsalary;   } Variable

    Void totalSalary(double basic) // basic is a
                                      method parameter
    {
    double hra = 10*basic/100; //local variable
    double da = 15*basic/100; //da is a local
                                 variable.
    }
```

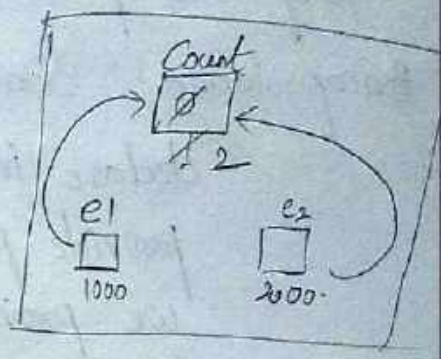
Q) When to move for static variables: When the data is common for all objects we use static variables

→ Programming wise examples:
Static String institute_name, address, ph_no etc

→ Real time of static variables: Subscriber in Youtube, likes, status in whatsapp, count of customer accessing the web page, etc.

Ex: using System;
namespace MySpace
{
class Employee
{
internal static int count = 0; //static variable.
internal Employee() //constructor
{
count++;
}
internal void print() //method
{

}
}
class Program
{
public static void Main(string[] args)
{
Employee e1 = new Employee(); //instance created invokes the constructor.
Employee e2 = new Employee();
e2.print(); // invokes the print method.
Console.Write("The total count : " + Employee.count);
Console.ReadLine();
}
}
}




Classname.Count
↓
to call static variable

# Properties in C#.Net (Property):

By using this property concepts we can perform better way encapsulation.

## C# properties (Get and set):

They are pre-defined methods.

Encapsulation: Sensitive data (hide)

declare instance variables as private.
provide public methods as of now.
we provide some predefined methods.
get and set --- this are called as C# properties.

* Provide public Get and set methods through properties to access and update the values of a private instance variables.
* As we know that private instance variables can only be accessed within the same class (an outside class has no access to it).
  However, sometimes we need to access them - this can be done with 'properties' concept.

## Properties (Get & set):

Encapsulation: Combining of properties & methods in a single unit is called Encapsulation.

* Generally, we declare fields/variables as private.
* Generally, in public or internal methods we define (user define methods) but now use get & set methods (pre-defined methods).
* Through, this methods (Get & set) we can access properties (private properties and update the value of a private field/variable)

get and set property methods:

property is a special type of method which will contain two accessors.

1. set.
2. get.

Rules: Property name should be same as variable name but property name start with capital letter.

ex: variable name is age.
property name is Age.

Syntax to define property:

```
<access_modifier> <return_type> <property_name>
{
    get
    {
        return <variable_name>;
    }
    set
    {
        <variable_name> = value;
    }
}
```

value → keyword. [which stores the data passed from the Main method.]

1) set:

Whenever we assign a value to property concern property set accessor will execute, within the set accessor we will have predefined variable ist the assigned value will be assigned into "value" variable then the value will be assign into given variable.

Note: Within the set accessor before accessing we can implement validations.

get :

Whenever we are retrieving the value from property, concern property get accessor will execute.

Ex:

```
using System;
namespace MySpace
  {
    Class Person
      {
        private string name; //instance variable
        internal string Name //property.
          {
            set
              {
                name = value;
              }
            get
              {
                return name;
              }
          }
      }
    Class Program
      {
        public static void Main (string [] args)
          {
            Person myobj = new Person();
            myobj.Name = "Hema";
            Console.WriteLine (myobj.Name);
          }
      }
  }
```

get; set; (or) set; get;

value is a keyword [the parameter which is passed will be stored in value].

returns the value stored in name.

Property

Ex:

```
using System;
namespace MySpace
{
  class Employee
    {
      ulong eno;
      String ename;
      byte eage;
      ulong ephno;
      double esal;
      Static ulong compphno = 78464321;

    internal Employee(ulong eno, String ename, byte eage,
                      ulong ephno)
      {
        this.eno = eno;
        this.ename = ename;
        this.eage = eage;
        this.ephno = ephno;
      } this.e
    internal byte Eage //property.
      {
        get              //(08)  get;
          {
            return eage;
          }
        set
          {
            while (Value <= 20 || Value >= 60)
              {
                Console.WriteLine("Enter age between 21 to 60");
```

```
            value = byte.Parse(Console.ReadLine());
            } eage = value;

        }
    }
    /* OR in real time
        set
        {
            if(value >= 0)
            {
                eage = value;
            }
            else
            {
                throw new ArguementException("Age cannot
                                be negative");
            }
        }  */
    internal ulong Ephno
    {
        get
        { return ephno;
        }
    }
    internal static ulong compphno
    {
        get
        {
            return compphno;
        }
    }
```

```
    internal void Display()
    {
        Console.Write("Employee No: " + eno);
        Console.Write("Employee Name: " + ename);
        Console.Write("Employee Salary: " + esal);
    }
}//end of Employee class

class Program
{
    public static void Main(string[] args)
    {
        Employee emp = new Employee(121, "Rao", 67432,
                                    9987654321);
        //set Property of Eage;
        emp.Eage = 90;

        //Display
        emp.Display();
        Console.WriteLine("Age of employee: "+emp.Eage);
        Console.WriteLine("Phone of employee: " + emp.Ephno);
        Console.WriteLine("Company phone number: " +
                                    Employee.Compphno);

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

```
//Bank application using property.
using System;
namespace MySpace Bank
{
  class BankAccount
  {
    private decimal balance; //instance variable.
    internal decimal Balance //property.
    {
      get { return balance; }
      set
      {
        if(value>=0)
        {
          balance = value;
        }
        else
        {
          throw new ArguementException("Balance cannot
                                        be negative");
        }
      }
    }
    public void Deposit(decimal amount)
    {
      if(amount > 0)
        Balance += amount;
    }
    public void withdraw(decimal amount)
    {
      if(amount>0 && amount <= Balance)
      {
```

```
            Balance -= amount;
        }
    }
}
Class Program
{
public static void Main(string[] args)
    {
      BankAccount account = new BankAccount();
      account.Deposit(500);
      account.withdraw(200);
      Console.WriteLine("Current Balance" + account.Balance);
      ~~console.Read();~~
      } try
      {
          account.Balance = 450; //valid.
          Console.WriteLine("Balance " + account.Balance);
          account.Balance = -100; //invalid.
      }
      catch (ArguementException ex)
      {
        Console.WriteLine(ex.Message);
      }

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

Q) You are developing a program to save the features of different cars in a car dealership and be able to show when required. The dealership sells only two types of cars, Maruthi and Hyundai. Both types of cars have following properties:

1. length
2. Breadth.
3. Ground clearance.
4. color.

Only Hyundai cars have airbags. Therefore, for only Hyundai car we need to gather 'Number of airbags.'
Only Maruthi cars have ~~operate~~ battery operated type and may have 2 to 8 batteries. Therefore, for maruthi cars only we need to gather 'Number of batteries.'
With above information please design a class hierarchy with get and set property methods.

Questions

① What are main features of OOPs concept
② What is an object & class ?
③ What is Encapsulation ?
④ What is abstraction ?
⑤ Write about on access modifiers ?
⑥ What is constructor ?
⑦ What is static constructor & static method ?
⑧ What is the need of property concept ?
⑨

Static Constructor:

While defining a constructor, if we have static keyword which is called as static constructor.

" Static constructor cannot contain parameter, accessmodifier. "

Syntax:

```
static <class_name()>
  {
  }
```

Q) When static constructor will call ?

When CLR requires class.

Points to consider for static constructor:

* A class can contain only one static constructor.
* If a class having one instance constructor and one static constructor, first static constructor will execute then instance constructor will execute.

When Static method ?

Whenever a method is not accessing any instance variable we will go for Static method.

(or)

Whenever we want perform operation on only Static variables or local variables or no variables then declare that method static methods.

Ex: Advantage of Static method: Object Not required

List out predefined Static methods in .Net ?

WriteLine(), ReadLine(), ToInt32(); Equals();
ReferenceEquals().---

* If we don't want to create obj, then inorder to enter in class, we define Static method [having Static variables only].

Constants:

* to declare a constant we have to use Const keyword.
* Constant is a member of a class.
* Constant represents a fixed value which is common for all objects, but value will never changes.
* By default constant is Static (will get the memory when the class is loading).
* Constant should initialize at the time of declaration.
* Predefined constant in C#.Net.
  Min Value and Max value.

Note: To Represent Universal Fixed Values we will go for constants.

Ex: PI, No. of days per week, No of Hours, Speed of light etc.

Real time Examples: Comp Establishment, Company founder name, etc.

Difference b/w Constants and Static variables.

| Constant variable | Static variable |
| --- | --- |
| → const keyword. | → Static keyword. |
| → Constant value can't be changed | → Static value can change. |
| → should be initialized at the time of declaration. | → Can be initialized at any time & declaration as well as in static methods. |
| → Whenever we want to have the same value to all objects but not required to change forever. | → Whenever we want to represent common value for all objects but required to change in feature. |
| → Declared during design time. | → Declared during runtime. |
| → By default is static | → Static keyword should be declared explicitly. |

readonly:

* readonly field should be declare by using readonly keyword.
* readonly is by default instance.
* readonly field will be also creating for every object with different value but value can't be changed.
* readonly can be initialize at the time of declaration as well as within the instance constructor but can't be initialize within the method.

Realtime Examples: aadhar_no, PAN_No, etc.
Employee examples: eno, ename, dob etc.

When we will go for readonly?
Whenever we want to have a field for multiple objects with different values but values should not be changed then we will go for readonly.

Difference b/w constant and readonly.

| readonly | constant. |
|---|---|
| → readonly keyword | → const keyword. |
| → by default instance | → by default static. |
| → Initialization can be done during declaration and also runtime only in constructor. | → Constant can be initialized only at the time of declaration i.e., during design time (compile time), can't be initialized in runtime. |
| → Values will differ from one object to another object | → Value will be same for all objects. |
| When? Whenever we require a field for all objects with different values but value doesn't require to change then we go for readonly. | When? Whenever we want same value for all objects and not required to change during entire p time (forever). |

Q) Create a class with the following like instance variables, static variables, constants, readonly, static constructor, static method, property concept (if possible)

```
using System;
namespace MySpace
{
  class Employee
    {
readonly  readonly ulong eno;
          readonly string ename;   } //instance variable
          double salary;
          static string compname;
          static string complac;   } static variable
          static ulong compphno;
```

```
const string compfounder = "xyz"; // constant
const uint estbyear = 1990;

static Employee()  //Static constructor
{
  Console.WriteLine("Hello Employee...");
  compname = "AisIndia";
  comploc = "Hyderabad";
  compphno = 9342433344;
}

internal Employee() //constructor
{
  Console.WriteLine("Enter eNo: ");  → Ulong
  eno = Convert.ToUInt64(Console.ReadLine());
  Console.WriteLine("Enter eName: ");
  eName = Console.ReadLine();
  Console.WriteLine("Enter Salary: ");
  salary = Convert.ToDouble(Console.ReadLine());
}

internal void Display() //instance method.
{
  Console.WriteLine("Employee ID: " + eno);
  Console.WriteLine("Employee Name: " + ename);
  Console.WriteLine("Salary: " + salary);
}

internal static void Print() //Static method:
{
  Console.WriteLine("Company name: " + compname);
  Console.WriteLine("Company established year: " + estbyear);
  Console.WriteLine("Company phone number: " + compphno);
```

```
        Console.WriteLine("Company location: " + Comploc);
    }
}

class Program
{
    public static void Main(string[] args)
    {
        Employee eObj = new Employee(); //invokes the constructor.
        Employee.Print(); //invoke the static method
        Console.WriteLine("Employee Details: ");
        Employee.Display();
        eObj.Display(); //invoke the instance method.
        Console.ReadLine();
    }
}
```

Develop a OOPs Solution for the following Bank Requirement.

1. Create a bank account with a/c No, Name, balance, a/c holder phoneno, bankname, bank loc, bank phno, bank estb year, bank establish person name.
2. Define an initialization mechanism for a/c holder age but the age should be min 18yrs to 100 yrs.
3. Define a retrieval and initialization mechanism for a/c a/c holder phone no, but this initialization mechanism for to change the phoneno of the a/c holder.
4. Define an retrieval mechanism for bank ph No.
5. Define a mechanism to withdraw the amount and display current balance after withdrawl, Min bal a/c should be 500.

6. Define a mechanism to withdraw the amount and after deposit current balance.

7. Define a mechanism to display the bank details i.e, bankname, bankloc, bank establish year, ph No. bank person name, etc.

8. Define a mechanism to display bank a/c details ie; a/c No, a/c Name, a/c ph No, a/c holder age etc.

Note: After creating account display a/c created successfully msg with following a/c details.

Fields:

```
readonly ulong acct_no
readonly string acct_name
readonly ulong acct_phno
double acct_bal
byte acct_age
string acct_loc
readonly ulong acct_aadhar
readonly string acct_PAN
Static string bank_name
static string bank_loc
static ulong bank_ph
Const short bank_yr = 1990
Const string bank_person = "OMG"
```

```
internal Account { }
internal byte CustAge { set; }
internal ulong CustPhno. { get; set; }
internal ulong BankPhno { get; }
internal void withdraw (double amt);
internal void deposit (double amt);
Display AccountDetails()
Display BankDetails()
TransferAmt (Object, amt) etc.
```

## Inheritance:

The advantage of Inheritance is Extensibility and object code Reusability.

What is Inheritance?

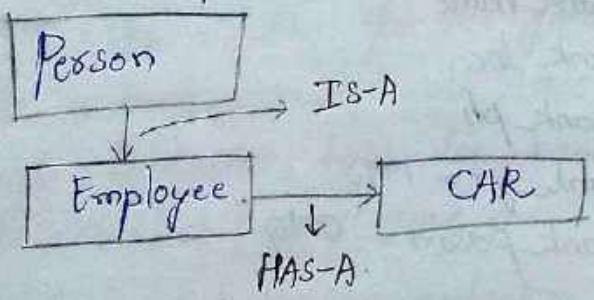
It is one kind of intelligence how to extend the current Software.

* Defining a new class by reusing the members of other class.

For reusing the members we will maintain a relationship b/w the class of 2 way relationship, i.e;

1) Is – A
2) HAS – A

1) Is-A: If we want to extend existing functionality with some extra functionality then we go for Is-A relationship.
2) HAS-A: If we don't want to extend and just we have to use existing functionality then we go for HAS-A relationship.




Terminologies:

Parent / Base / Super Class (having unique (common) features): The class from which members are reused.
(Python, C++, C#)

Child / Derived / Subclass: The class which is using members.

Note: Every class is not a superclass until or unless the class is inherited.

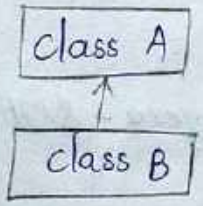
Type


Types of Inheritance:

1. Single Inheritance.
2. Multilevel Inheritance.
3. Hierarchical "

* Multiple Inheritance, Hybrid Inheritance.
  exist in C# but doesn't support & support by interfaces

1) Single Inheritance:




```
class A
{
...
}
class B:A
{
...
}
```

The inheritance mechanism in which the single subclass is inherited from the single superclass.

Note:

* Basically we create an object from the derived class since it has the properties of base class also.

Ex:

```
using System;
namespace MySpace
{
  class Guru
    {
      internal void Call()  // Superclass, base class.
        {
          console.WriteLine("Guru-call");
        }
```

```
    internal void Camera()
    {
        Console.WriteLine("Guru - Camera - 2MP");
    }
}
class Program Galaxy : Guru.  // Derived class & Subclass
{
    internal void call VideoCall()
    {
        Console.WriteLine("Galaxy - VideoCall");
    }
    internal void Camera()
    {
        Console.WriteLine("Galaxy - camera - 8MP");
    }
}

class Program
{
    public static void Main(string[] args)
    {
        Galaxy g1 = new Galaxy();
        g1.call();  // Access existing.
        g1.VideoCall();  // New feature.
        g1.Camera();  // updated feature.

        Guru g2 = new Guru();
        g2.call();
        g2.Camera();
        //g2.VideoCall();  error.
        Console.ReadLine();
    }
}
```

**this** : Reads current object address.

this(): It is used to invoke the constructor of same class.
It must be used inside the constructor.

base : A reference variable used to invoke instance membess of parent class from child class.
* It must be used inside instance method or instance block or constructor of child class.

base(): It is used to invoke the constructor of same class.
It must be used inside child class constructor.

Example:

```
using System;
namespace MySpace
{
  class Parent
    {
     internal int a, b;
     internal parent (int a, int b);
       {
        this. a = a;
        this. b = b;
       }
    }
     class child : Parent
       {
        int c, d;
        internal child (int a, int b, int c, int d) : base(a,b)
          {
            this. c = c;
            this. d = d;
          }
       }
```

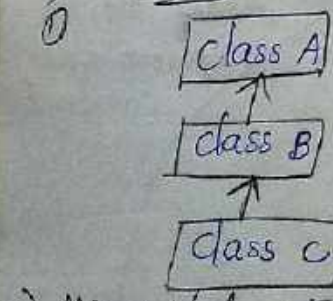
invokes the parent class constructor

```
internal void Display()
{
Console.WriteLine("Parent a: " + base.a);
Console.WriteLine("Parent b: " + base.b);
Console.WriteLine("child c: " + this.c);
Console.WriteLine("child d: " + this.d);
}
}
class inherit
{
public static void Main(string[] args)
{
child c = new child(10, 20, 30, 40);
c.Display();
}
}
}
```

invokes the child constructor

~~2) Multi~~

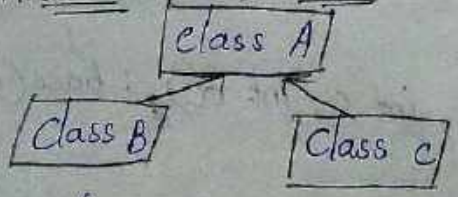
2) Multilevel Inheritance:




* The Inheritance Mechanism in which every intermediate class ~~acces~~ acts as Superclass.

→ Multilevel Inheritance.

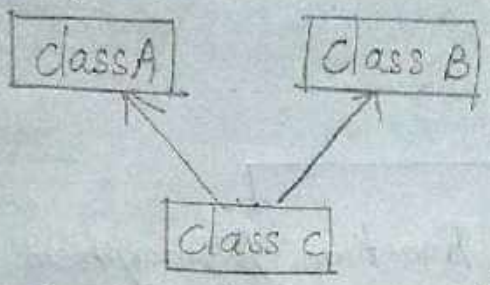
3) Hierarchical Inheritance:




→ Hierarchal Inheritance.

* The inheritance Mechanism in which multiple subclass will be inherited from a single base class (Superclass).

4) Multiple Inheritance:




The Inheritance Mechanism in which single subclass will be inherited from multiple base classes [Super classes].

* This type of inheritance leads to ambiguity problem. This is a reason in C#.Net directly multiple inheritance not supported.

→ Why Multiple inheritance is not possible in C#.Net ?

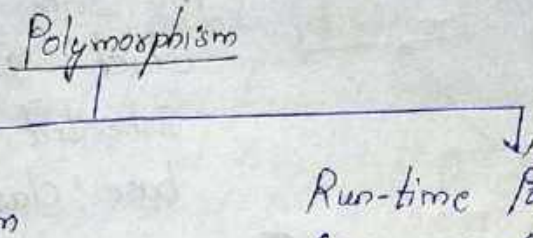
* Since the derived class will be inherited from a multiple base classes, the derived class get the properties from both the classes.

→ If there is same properties are existed then there will be ambiguity problems occurs and also code complexity will takes place, so to avoid all these we implement Multiple inheritance through "interface."

Q) What is ultimate Super class for all the ~~objects~~ .Net class Object class.

Note: * In "ASP.Net Page" is the Super class for all webForms classes i.e., webform1, webform2, ---. webformn.

4) Polymorphism:




Compile time Polymorphism
(Static Binding)
(Early Binding)
Ex: Achieved by Method Overloading
Operator "

Run-time Polymorphism.
(Dynamic Binding)
(Late Binding)
Ex: Achieved by "Virtual" Method Overriding.

1. Static Polymorphism:
In Static polymorphism a method which will bind at compile time will execute at runtime.
* Static polymorphism we can achieve with the help of "method Overloading.

Method Overloading:
Method Overloading means implementing multiple methods with the same method name but different Signature is called the Method Overloading.
(arguements)

Example: In C#.Net Write(), WriteLine(), Read(), ReadLine() are "Overloaded methods."
* In database "Select" command is "Overloaded."
* Keyboard is Overloaded, Mouse Click event is Overloaded.
* Due to overloaded the advantage is users convience as well as in device Oriented Machinary size will decrease.

WriteLine() --- Overloaded 19 times
Write() --- Overloaded 18 times.

ToInt32() --- Overloaded 19 times.
ToString() --- 36 times , etc.

Example:

```
using System;
namespace Myspace
{
  class Calculate
  {
  internal void Add(int a, int b)
    {
     Console.WriteLine("Addition of 2 numbers: " + (a+b));
    }
   internal void Add(int a, int b, int c)
    {
     Console.WriteLine("Addition of 3 numbers: " + (a+b+c));
    }
    internal void Add(~~int a,~~ double a, double b)
     {
     Console.WriteLine("Addition of 2 double numbers: " + (a+b));
     }
  }
   class Inherit
    {
     public static void Main()
      {
        Calculate obj = new Calculate();
        obj.Add(10, 20);
        obj.Add(20, 30, 40);
        obj.Add(40.23, 50.23);
        Console.Read();
      }
}
```

* In the above program, at the time of compile the following 2 statements are:

```
Calculate obj = new Calculate();
Obj.Add(10,20);
```

Here compiler will consider, reference variable type to bind the method among 3 methods, Add methods() within the calculate class.

Here, the reference variable type is

```
Calculate Obj.
```

Here type is Calculate due to that reason it will go to calculate class will bind Add(), which contain 2 integer parameters.

* When we execute the statements CLR will consider Object type (RHS) ie., new Calculate().

```
Obj.Add(10,20);
```

it will execute based on Calculate class of Add method with 2 parameters.

So, in above statements, it is executing a method which is binding at compile time due to that reason we can say function overloading is Static Polymorphism.

Advantage of Overloading: is fastness.

How to prove function overloading is a static polymorphism?

a) How compiler bind method:
Based on reference variable type (LHS).

a) How CLR will create the method:
Based on object type (RHS);

* Overloading depends on only parameters but not return type.

Function Overriding [Method Overriding]:

* It means implementing multiple methods with the same name and same Signature in a combination of base and derived class including return type.
* For base class method, we have to use virtual keyword and the derived class method, we have to use override keyword.
* While implementing function overriding both methods access modifiers, return type, method name, Signature are same.

Ex:

```
using System;
namespace MySpace
  {
  class BC [Base class]
    {
      internal virtual void Display()
        {
          Console.WriteLine("BC Base class Display");
        }
    }
  class DC : BC [Derived class]
    {
      internal override void Display()
        {
          Console.WriteLine("DC Derived class Display");
        }
    }
/*class DC1 : DC
    {internal override void Display()
```

```
{
    Console.WriteLine("DC 1 Derived class Display");
}
}*/
class Program
{
public static void Main(string[] s)
{
    BC bobj = new BC();
    bobj.Display();

// DC dobj = new BC();   Error (we cannot call base class by the derived class object)
// dobj.Display();

BC bobj1 = new DC();
    bobj1.Display();

}
}
}
```

o/p:

BC Base class Display.
DC Derived class Display.

* In the above example when we compile the below 2 statements,

```
BC bobj1 = new DC();
bobj1.Display();
```

Control will give priority for reference variable type i.e. Base obj due to that reason, control will

bind base class method.

```
class BC
{
    internal virtual void Display()
    {
    }
}
```

bobj1.Display()

bobj1.Display(); This statements when we compile the above 2 statements control will give priority for obj type i.e, new Der() due to that reason control will execute.

Runtime it will check the type of object what we assigning in base class reference.

```
class Der : Base
{
    internal override void Display()
    {
    }
}
```

bobj1.Display()

bobj1.Display() : with this we can say function overriding is a Dynamic Polymorphism. Compile time it is binding base class Display method and run time it is executing Derived class Display method.

Q) Can we say following is Dynamic polymorphism.

1) Der obj = new Der();
   obj.Display();

No it is not Dynamic polymorphism.

2) Base obj = new Base().
   obj.Display();

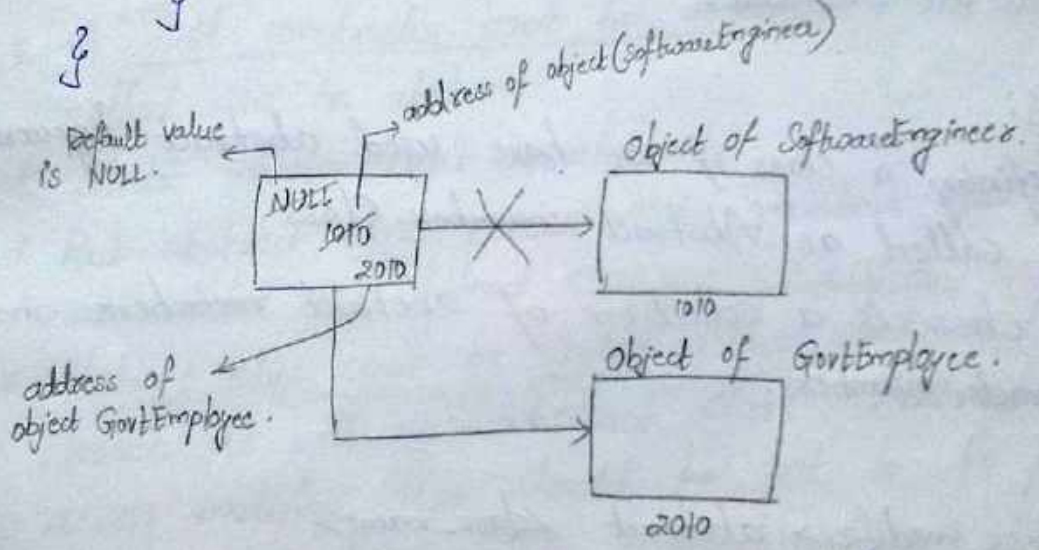
No it is not Dynamic polymorphism.

When we will go for Overriding and in Overriding why Super class reference variable & Sub class object?
Providing Solution by example:

```
using System;
namespace MySpace
{
  class Employee                          → Superclass
  {
    internal virtual void CalSal(double basicSal)
    {
      double grossSal = basicSal;
      Console.WriteLine("Employee Salary: " + grossSal);
    }
  }
  class SoftwareEngineer : Employee       → Subclass, → Superclass
  {
    internal override void CalSal(double basicSal)
    {
      double hra = basicSal * 4/100;
      double grossSal = basicSal + hra;
      Console.WriteLine("Employee Salary: " + grossSal);
    }
  }
  Class GovtEmployee : Employee           → Subclass, → Superclass
  {
    internal override void CalSal(double basicSal)
    {
      double hra = basicSal * 4/100;
      double da = basicSal * 2/100;
      double grossSal = basicSal + hra + da;
  } } Console.WriteLine("Employee Salary: " + grossSal);
```

```
Class
{
  public static void Main (String[] args)
  {
    Employee e1 = new SoftwareEngineer();  // [Reference of base class is used to create the object of derived class
    e1.CalSal(1000);   // Invoking the method of SoftwareEngineer.
    e1 = new GovtEmployee();  // Using the reference already created from Employee base class is creating the new obj of the GovtEmployee.
    e1.CalSal(2000);   // Previous object instance will be destroyed as it replaced by new object.
    Console.Read();
  }
}
```

reference — object creation




* Whenever we want to implement same behavior with different functionalities in the combination of base and derived class we will go for Method Overriding
* While implementing "Method Overriding" will go for Superclass reference variable and Subclass object because to make previous object as Unreffered object and current object be in live.

With this we can save the memory.

# Difference b/w Overloading & Overriding.

| Overloading | Overriding. |
|---|---|
| → Multiple methods with same name but with different Signature. | → Can have multiple methods with same name and same Signature. |
| → To implement for Overloading we don't require any keywords. | → To ~~not~~ implement this we use virtual & override keywords. |
| → It is a static polymorphism | → It is a Dynamic polymorphism. |
| → This can be implemented in single class and also in combination of base & derived class. | → It is not possible in single class we have to go base & derived class. |
| → Static methods are Overloaded. | |

## Abstract Class:

* While defining a class if we have used abstract keyword which is called as Abstract ~~member~~. Class.
* Abstract class is a collection of abstract members and non abstract members.

Syntax:

```
<access_modifier> abstract <class_name>
{
    // abstract members.
    // non-abstract members.
}
```

* Abstract class by default uses 'Virtual' keyword.

Q) What do you mean by abstract member?

While declaring a class member if we have used abstract keyword which is called as abstract member.

* Abstract member only contain declaration part within the abstract class, we have to define/implementation

abstract member within the derived class by using override keyword.

Q) What do you mean by declaration & definition?

```
internal void Display() //declaration part
{
    Console.Write("Hello"); // definition part.
}
```

* By default abstract member is a virtual due to that reason while declaring abstract class we don't requires to use virtual keyword.
* A field constructor can't be abstract but property and method can be abstract.
* Static member can't be abstract.
* But abstract class contain static members but that should be ~~non-static~~ non-abstract class; i.e. non-inheritable class.
* Abstract class can't be sealed class, static and private classes as well as abstract class. Abstract members - access modifiers also should ~~be~~ not to be private because abstract class is depending on a concept called Inheritance.

Q) What do you mean by non-Abstract member?

Implementing member is non abstract member, means this members will contain declaration & definition.

Q) What is the difference b/w Normal class & Abstract class?

→ Normal class contain only normal members these are ~~etc~~ called as non-abstract members or "Concrete members" due to that these are called as ~~call~~ class as Fully Implemented class. Finally it is called as "Concrete

class or Non-abstract class.

→ Abstract class is a collection of abstract members and non-abstract members due to that reason which is called as Partially implemented class.

→ We can't create object for abstract class but we can create reference variable for abstract class.

Q) When we will go for abstract class?

Whenever we want to implement some members in current class some other members in feature classes (Derived class).

Example:

```
using System;
namespace MySpace
{
  clas abstract class Customer
      {
        internal void AboutInfo() //non-abstract member.
          {
           Console.WriteLine("About Info is calling ");
          }
          internal abstract void CalBill(int totunits); // declaration
                                  ↳ abstract member.
      }
      class IndustryCustomer : Customer // def
        {
        internal override void CalBill(int totunits) // definition by its
          {                                  own class.
           double totbill = totunits * 10;
           Console.WriteLine("Industry Customer Bill : " + totbill);
          }
}
```

```
class ResidentialCustomer : Customer
{
  internal override void CalBill(int totunits)
  {
    double totbill = totunits * 6;
    Console.WriteLine("Residential Customer bill : " + totbill);
  }
}
class IndAgriculturalCustomer : Customer
{
  internal override void CalBill(int totunits)
  {
    double totbill = totunits * 2;
    Console.WriteLine("Agricultural customer bill : " + totbill);
  }
}
class Program
{
  public static void Main(string[] args)
  {
    //Customer cust = new Customer(); error.

    Customer cust = new IndustrialCustomer();
      cust.CalBill(200);
      cust.AboutInfo();
      cust = new ResidentialCustomer();
       cust.CalBill(100);
      cust = new AgriculturalCustomer();
       cust.CalBill(300);

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

List of the predefined Abstract classes in .Net ?

1. DbConnection
2. DbCommand.
3. Db DataAdapter.
4. DbDataReader and so on.

# OBJECT ORIENTED PROGRAMMING (OOPs)

10/7/24

## Interfaces:

* To define interface we have use the 'interface' keyword.
* interface is a collection of abstract members.
* By default interface members are public & abstract.
* Finally, we can say interface will not have any implementation.
* interface members we should implement within the derived class without using 'override' keyword.

Syntax:

```
interface <interface_name>
  {
    //abstract members;
  }
```

Note: Interface name start with capital 'I' followed by its name.

Ex:
```
interface IMyinterface
  {
  }
```

* An interface cannot contain fields & constructors.
* We can declare a property & method within interface but we can't implement a property or method with the interface.
* Interface supports inheritance including Multiple inheritance.
* We can't create object for interface as well as but we can create reference variable for interface.
* Within the interface we can't define static members.

Ex:
```
using System;
namespace MySpace
  {
  interface First //interface
    {
```

→ Contains abstract, virtual & interface.

```
        void m1(); //declaration
        void m2(); //declaration
    }  derived class  → base class
class Second : First //until or unless we provide the definition
                     for every method we get error.
 {
 public void m1()  → definition for m1 method.
  {
   Console.WriteLine("m1---"); //definition.
  }
 public void m2()
  {
   Console.WriteLine("m2---"); //definition.
  }
 }
class Program
 {
 public static void Main(string[] s)
  {
   First Obj = new Second();
   Obj.m1();
   Obj.m2();
   Console.Read();
  }
 }
}
```

Superclass reference → First Obj

→ Subclass object.

② Example

```
using System;
namespace Bank
 {
  interface IBankAccount
   {
    bool DepositAmount(decimal Amount);
    bool WithdrawAmount(decimal Amount);
    decimal checkBalance();
   }
```

```
public class SavingsAccount : IBankAccount
{
  private decimal Balance = 0;
  private readonly decimal PerDayWithdrawLimit = 10000;
  private decimal TodayWithdrawal = 0;
  public bool DepositAmount (decimal Amount)
  {
   Balance = Balance + Amount;
   Console.WriteLine ($" You have Deposited: {Amount} ");
   Console.WriteLine ($" Your Account Balance: {Balance} ");
   return true;
  }
  public bool WithdrawAmount (decimal Amount)
  {
    if (Balance < Amount)
    {
     Console.WriteLine("You have Insufficient Balance!");
     return false;
    }
    else if (TodayWithdrawal + Amount > PerDayWithdrawLimit)
    {
     console.WriteLine ("Withdrawal attempt failed");
     return false;
    }
   else
    {
     Balance = Balance - Amount;
   TodayWithdrawal = TodayWithdrawal + Amount;
   Console.WriteLine ($" You have Successfully withdraw:
                        {Amount} ");
   Console.WriteLine ($" Your account Balance: {Balance} ");
   return true;
  }
}
```

```
    public decimal CheckBalance()
    {
      return Balance;
    }
}
public class CurrentAccount : IBankAccount
{
  private decimal Balance = 0;
  pri public bool DepositAmount(decimal Amount)
    {
      Balance += Amount;
      Console.WriteLine($"You have Deposited: {Amount}");
      Console.WriteLine($"Your account Balance: {Balance}");
      return true;
    }
    public bool WithdrawAmount(decimal Amount)
    {
      if(Balance < Amount)
        {
          Console.WriteLine("You have Insufficient balance!");
          return false;
        }
        else
          {
            Balance -= Amount;
            Console.WriteLine($"You have Successfully
                              Withdraw: {Amount}");
            Console.WriteLine($"Your current Balance: {Balance}");
            return true;
          }
    }
    public decimal checkBalance() { return Balance; }
}
```

```
class Program
{
public static void Main(string[] s)
{
Console.WriteLine("Savings Account: ");
IBankAccount savingsAccount = new SavingsAccount();
savingsAccount.DepositAmount(2000);
savingsAccount.DepositAmount(1000);
savingsAccount.WithdrawAmount(1500);
savingsAccount.WithdrawAmount(5000);
Console.WriteLine($"A Savings Account Balance :
                  {savingsAccount.checkBalance()}");

Console.WriteLine("\n Current Account: ");
IBankAccount currentAccount = new CurrentAccount();
currentAccount.Deposit(500);
currentAccount.Deposit(2000);
currentAccount.WithdrawAmount(1000);
currentAccount.WithdrawAmount(5000);
Console.WriteLine($"Current Account Balance :
                  {currentAccount.checkBalance()}");

Console.ReadLine();

}
}
}
```

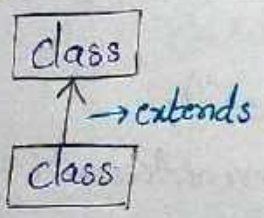
Q) Can we implement multiple inheritance in a combination of one class & multiple inheritances.

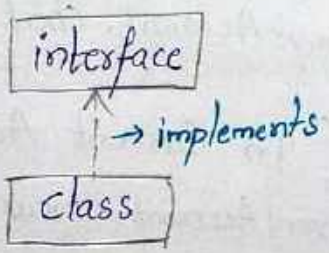
Yes, but first we have to inherit class then we have to inherit interfaces.

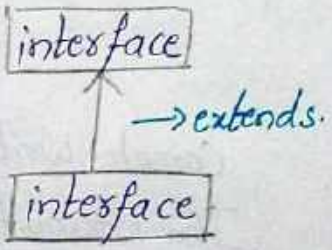
Ex:
```
interface I1
{
}
interface I2
{
}
```

```
class BC
{
}
class DC : I1, I2
{
}
```

Relations:



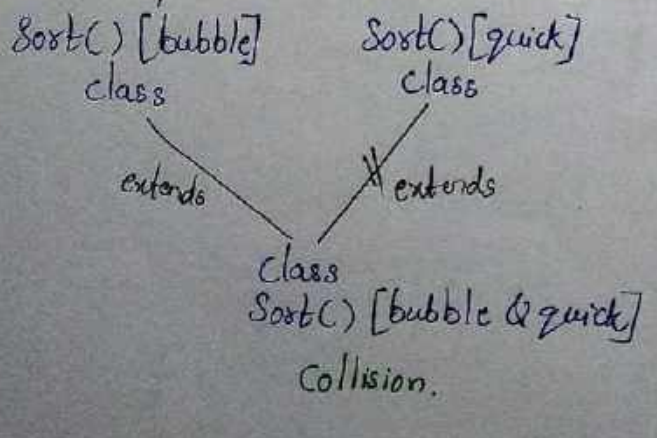

Multiple inheritance in C#:

* A class can extends only class.
* A class can extends class & implements any no. of interfaces.
* An interface can extend any number of interfaces called Multiple inheritance.

Predefined interfaces in .Net.

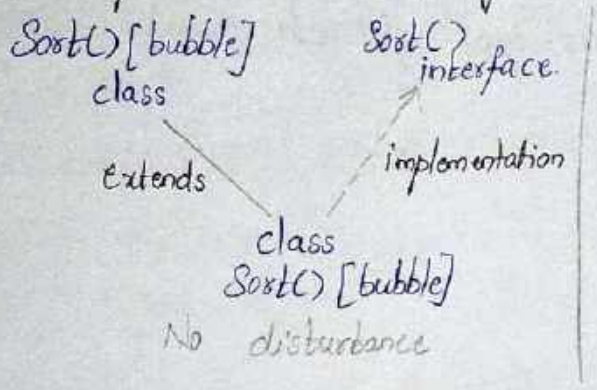
IDbConnection, IDbCommand, ITditableTextControl etc.

Ex: ① Multiple Inheritance (Multiple classes)



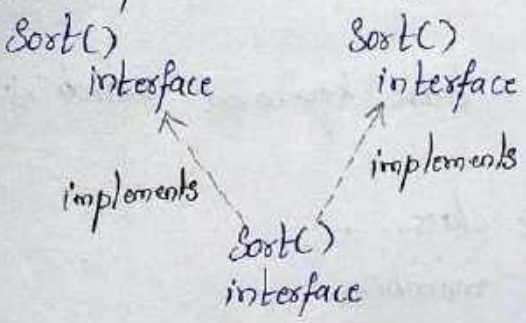

```
class A
{
}
class B
{
}
class C : A, B
{
}
```

2) Multiple Inheritance [Single class & interface]




```
class A {
  }
interface B {
  }
class C : A, B
  {                  ↳ multiple inheritance.
  }
```

3) Multiple inheritance [2 interfaces - interface]




```
interface A {
  }
interface B {
  }
class C : A, B
  {
  }
```

Object Up-Casting:

We can store the address of child class into parent type reference variable.

Ex: Parent addr = new child(); // upcast

(or)

```
child obj = new child();
Parent addr = obj ;    // upcast.
```

DownCasting:

The concept of collecting child object address back to child type reference variable from parent type.

Ex:

```
parent p = new parent();
child c = (child)p ; // it is not down casting.

parent p = new child();
child c = (child) p ; // it is down casting.
```

Different types of Classes ?-

In C#.Net we can define various type of classes.

1. Normal class (concrete class)
2. Static class.
3. Sealed class.
4. Partial class.
5. abstract class.
6. generic class.

Static class :

While defining a class if we have static keyword which is called a static class.

* We can't create object for static class.
* Static class can contain only static members.
* Static class can't be inherited.

* To define specific functionality
↓
Utility class.

Ex:

```
using System;
namespace ExampleStatic
{
 static class MyClass
  {
    // const int a=45; //valid const indirectly static member
    internal static void show()
     {
      Console.WriteLine("Show method is called");
     }
    internal static void Print()
     {
      Console.WriteLine("Print method is called");
     }
  }
```

```
/*
  class Class1 : Myclass
   {
     Cannot derive static class [Non-inheritable]
                                  ↓
                                static class
   }
*/
class Program
 {
   static void Main()
     {
       MyClass.Show();
       MyClass.Print();
     }
 }
}
```

Predefined Static class:

1. Console.
2. Convert & so on.

Where to use: Whenever you want to restrict the class with single instance we have to use static classes.

* The static classes are often used in situations where we don't need to create multiple instance and where you want to group related functionalities together, such type of classes Utility classes & helper classes.

Utility classes:

Static classes are often used to create utility classes that contain a collection of related methods that perform specific tasks. These methods can be called without the need to create the instance of a class.

Ex: ① Console → Static class specific for i/p & o/p purpose
② Convert → Static class specific for data conversion.

③
```
public static class MathUtils
{
    public static double Square(double num)
    {
        return num * num;
    }
}
```

In the above class, Static class MathUtil, still we can also define other Static methods related to Maths functions. Already we have some predefined static Utility classes especially for i/p, o/p – Console static utility class, for converting datatype – Convert static utility class, etc.

Usage:

```
double result = MathUtils.Square(5);
```
↑ static class name defined above.

By using static class & static methods we can also create Singleton classes i.e; only 1 instance of a class.

## Sealed class:

* Defining a class with sealed keyword.
* Sealed class can't be inherited.
* sealed class can contain instance & static members.
* We can create object for sealed class.
* A member become constant if we define it is sealed hence can't be modified.
* We can apply sealed modifier to class or method or variable.

Ex: If class is sealed it can't be inherited.

```
Sealed class A {
}
class B: A  //error.
{
}
```

* If the method is final, can't be overridden.
(When we don't want to use functionality or property further (extend) to others, we make that class or method as sealed).

Ex:

```
using System;
namespace MySpace
 {
   class X
   {
     protected virtual void F1()
       {
       }
     protected virtual void F2 () { }
   }
```

→ If we make class as sealed, we can't inherit it further implementation

```
sealed class Y : X
       {
```

→ it is better to make the method / functionality as Sealed.

```
        sealed protected override void F1() //sealed method.
          {
          }
        protected override void F2()
          { }
       }
       class Z : Y
        {
         protected override void F1() //error as the previous method which method has to override is sealed
           {
           }
         protected override void F2 () { } //valid. as it is not sealed.
        }
        class Program
         {
          public static void Main()
           {
            Console.Read();
           }
   }  }
```

Sealed can apply for class, method or instance variable.

Static variable & Sealed variable:

Ex: Static variable:

```
class Test
{
static double PI = 3.14; //static variable.
public static void Main()
  {
    Console.WriteLine("PI value: " + PI); //3.14
    PI = 3.142; // its changeable as it is defined as static variable.
    Console.WriteLine("PI value: " + PI); //3.142
    Console.Read();
  }
}
```

* If you want to make that static variable not to inherit or not to change use sealead for that variable.

Ex: Sealed variable:

```
class Test
{
Sealed static double PI = 3.14;
public static void Main()
  {
    Console.WriteLine("PI value: " + PI); //3.
    PI = 3.142; //error
    Console.WriteLine("PI value:" + PI).
  }
}
```

(Static variable → PI; Here only it shows error as it is marked sealed, & in the code it is mentioned to modify the PI value. It is not changeable & inheritable.)

Variable:

Static → Changeable & inheritable.
Constant → Not-changeable & inheritable.
Sealed → Not-changeable & not inheritance.

* Better to not apply Sealed for variables, they cannot be modified.

Predefined Sealed class:

1. SqlCommand
2. SqlConnection.
3. SqlDataAdapter, etc.

* Private class:
  If you define a class as private, we cannot create an object & we can't inherit also.

Partial class:

* Partial class in C# do allows a class, interface or struct definition to be split into multiple files.
* Each file contains a part of the whole class & when the program is compiled, the compiler combines these parts into a single class definition.
* This can be useful in scenarios such as auto-generated code where 1 part is generated by a tool, and another part is manually written.
* Ex: Let's say you have a class representing a person & you want to split the implementation into two files: One for basic information & another for Contact details.

File 1 (PersonBasicInfo.cs)

```
public partial class Person
{
    public String FirstName { get; set; }
    public String LastName { get; set; }
    public void DisplayBasicInfo()
    {
        Console.WriteLine($"Name: {FirstName} {LastName}");
    }
}
```

File2 (PersonContactDetails.cs)

```
public partial class Person
{
    public string Email {get; set;}
    public string phonenumber {get; set;}
    public void DisplayContactDetails()
    {
        Console.WriteLine($"Your Email: {Email}");
        Console.WriteLine($"Your contact number: {phonenumber}");
    }
}
```

* In this example, we have defined a person class across 2 files using the partial keyword. The person class is split into basic information in PersonBasicInfo.cs and contact details in PersonContactDetails.cs.

Now, you can use the person class as if it were defined in a single file:

* namespace should be same for all this files.

```
class Program
{
    static void Main(string[] args)
    {
        Person person = new Person
        {
            FirstName = "Hemalatha",
            LastName = "Chollangi",
            Email = "hema@gmail.com",
            phonenumber = "9987654321"
        };
        person.DisplayBasicInfo();
        person.DisplayContactDetails();
    }
}
```

## Enumerations:

↳ User-defined Value datatypes.

* Enumerator, class & structure all are common create user defined datatypes.
* Enumeration is value type.
* We have to use enum keyword. Enumerator is a collection of String constants which are representing collection of integer constants.

Syntax:

```
<accessmodifier> enum <enum_name>
 {
  <String constant> = <integer constant> ;
  <String constant> = <integer constant> ;
 }
```

Ex: In Realtime:

Machinary, mostly embedded programs are widely used enumerations constant they are use easy to access & more readable.

① Tiffin center bill machine,
② Current bill machine, etc.

Program:

```
using System;
namespace STRINGS
 {
  internal enum Weekdays
   {
     Sunday = 1,
     Monday = 2,
     Tuesday = 3,
     Wednesday = 4,
     Thursday = 5,
     Friday = 6,
     Saturday = 7 }
```

```
class MyClass
{
internal static void Display(Weekdays d)
{
 Switch(d)
 {
  case Weekdays.Sunday:
       Console.WriteLine("Today is Sunday"); break;
  Case Weekdays.Monday:
       Console.WriteLine("Today is Monday"); break;
  case Weekdays.Tuesday:
       Conside.WriteLine("Today is Tuesday"); break;
  Case Weekdays.Wednesday:
       Console.WriteLine("Today is Wednesday"); break;
  case Weekdays.Thursday:
       Console.WriteLine("Today is Thursday"); break;
  Case Weekdays.Friday:
       Console.WriteLine("Today is Friday"); break;
  case Weekdays.Saturday:
       Console.WriteLine("Today is Saturday"); break;
 }
}
static void Main(string[] args)
 {
  Weekdays day = Weekdays.Friday;
  MyClass.Display(day);
 }
}
}
```

* enumeration elements are internally series of integer constants.
* enum doesn't have returntype X.
* default datatype of enum is int.
* It will create memory in stack since it is value type.
* Here enum will be loaded like a class

Q) Can we change the datatype of enum ?
Yes, but it should be any one of the numerical datatype like below:

```
enum <enumname> : long
  {
  }
```

Ex:

```
using System;
namespace enumerations
 {
  enum tiffins
    {
     DeadlyIdly = 5 ,
     Dosa   = -2 ,
     Poori , // -1
     Vada  // 0
    };
    class Mytiffins
     {
      internal static void Display (tiffins t)
       {
         switch(t)
         {
          case tiffins.DeadlyIdly :
               Console.WriteLine("Deadly Idly : Rs 25/- "); break;
          case tiffins.Dosa :
               Console.WriteLine ("Dosa : Rs 40/- "); break;
          Case tiffins.Poori :
               Console.WriteLine ("Poori : Rs 40/- "); break;
```

```
            case tiffins.Vada:
                Console.WriteLine("Vada: Rs 45/-"); break;
            }
        }
        class Program
        {
            static void Main(String[] k)
            {
                Mytiffins.Display(0);
            }
        }
    }
```

## Structures:

* It is also value type.
* To define structure we have to use Struct keyword.
* We can create an object for structure without using any keyword. If the structure doesn't have instance variable.
* Structure doesn't contain userdefined default constructor.
* We can't define static structure but structure members can be static.

Difference b/w class & structure.

| Class | Structure. |
|---|---|
| → class keyword. | → Struct keyword. |
| → It is a reference type. | → It is a value type. |
| → If we create object for class it will store in the heap memory. | → It will store in stack memory. |
| → Within the class we can initialize instance variable & static variable at the time of declaration. | → In this we cannot initialize instance variable but static is possible at declaration. |
| → We cannot create object for class without using new keyword. | → We can create object for structure without new keyword. If the structure doesn't have |

| | |
|---|---|
| → class will support inheritance. | → Structure no. |
| → Accessmodifiers all supports. | → only some |
| → class can be abstract, static & sealed. | → Structure is not. |
| → class support all types of polymorphism | → Structure not supports overriding no runtime polymorphism. |

instance variabe.

Ex: using System;
namespace MySpace
{
public struct Student
{
internal string name;
internal int age;
public Student (String name, int age) // method in Structure.
{
this.name = name;
this.age = age;
}
}
Struct Test
{
public static void Main (string[] k)
{
→ object created for structure without 'new' keyword.
Student S1;
S1.name = "John"; //passing values directly using obj name.
S1.age = 34;
Student S2 = new Student ("kir", 23); // calling method in structure Stud
}
}
}

Namespace:

* A namespace is a way to organize & group related classes, interfaces, struct & other types into a single unit.
* A namespace provides a scope for the types it contains allowing them to have a unique names without conflicting with types in other namespace.

What is the purpose of Namespace:

1. Organize code: Easier to find & for using them.
2. Avoid name conflicts: We can use same names by different namespaces.
3. Make code more readable: Make easier to understand and readable.

Defining a Namespace:

A Namespace definition begins with the keyword namespace followed by the namespace name as follows:

Syntax:

```
namespace namespace_name
{
  // code declaration.
}
```

* To call the namespace-enabled version either function or variable prepend the namespace name as follows.

```
namespace_name . item_name;
```

&

Example:

```
using System;
namespace First_space
  {
    class MyFirst
      {
        public void func()
          {
            Console.WriteLine("Inside first_space");
          }
      }
    namespace Second_space {
      class MyFirst
        {
          public void func()
            {
              Console.WriteLine("Inside Second_space");
            }
        }
      class TestClass
        {
          static void Main(string[] args) {
              First_space.MyFirst fc = new First_space.MyFirst();
              Second_space.MyFirst sc = new Second_space.MyFirst();
              fc.func();
              sc.func();
              Console.Read();
            }
        }
```

* When the above code is executed and compiled, it produces the following result.
* Inside First-space
* Inside Second_space

# DELEGATES.

* In C#.Net, a Delegate is a type that represents a reference to a method with a specific signature."

Purpose:

* Delegates are used to pass methods as arguements to other methods, or to return a method from a method.
* Delegate is similar to 'Pointer to Function' concept in C.
* A delegate is declared using the delegate keyword. followed by the return type & the name of the delegate, & then the parameters in parenthesis.
* Delegates are 2 types:
  1. Single Cast delegate: It can hold reference (address) of a single method.
  2. Multi cast delegate: It can hold refesence (address) of a multiple methods.
* Using multi cast delegates we can invoke multiple methods with a single call.

Implementation:

Implementation of delegate concept can be divided into following steps.

Step 1: Defining a class.
Step 2: Defining a method.
Step 3: Defining a delegate.

Syntax: (define delegate syntax)

<access_modifier> delegate <return_type> <delegate_name>(<type> <arg1>, <type> <arg2>);

Step 4: Creating object for delegate & initializing the method name.

<delegatename> <delegateObj_name> = new <delegate_name>(methodname);

Step 5: Invoking the method by using delegate

<delegateObjname>;;

→ Delegate are called as "type Function pointers" (in C++) because delegate declaration should follow the method declaration which is holding by the delegate by the delegate Object.

→ That means method return type and signature should be same for delegate.

Note: delegate can hold the address of Static methods as well as instance methods.

Example:

```
// Program to invoke instance methods by using delegate
   object (single cast delegate)
using System;
namespace Delegates
 {
   class Calculate
    {
     internal int Add(int a, int b) //method(instance)
       {
         return a+b;
       }
    }
     internal delegate int Mydelegate(int x, int y); //delegate declaration.
   class Single_CastDelegate
     {
      public static void Main(string[] p)
       {
```

Delegate

```
Calculate cobj = new Calculate(); //creating obj for method
Mydelegate delobj = new Mydelegate(cobj.Add);
//passing method name as parameter for with delegate obj.
int res = delobj(10,5); //caller to method.
Console.Write("Addition: " + res);
Console.Read();
}
}
}
```

* We can also create Single cast delegate to invoke the static method.

Purpose to delegate is to:

1. Encapsulation method calls: Not showing method but only invoking method by its reference.
2. Implement Event Handling: It handles events etc. [Ex: Button click event in Windows Form]
3. It will be implement in "LINQ" also for Queries etc.

Example (2) on Single Cast Delegate:

```
//Example to invoke static methods by using delegate object
    (Single class delegate)
using System;
namespace Delegate
{
  public class MyDelegateClass //class name.declaration.
  {
    public delegate String GreetingDelegate(String name);
    // delegate declaration.
    Public static string Greetings(string name) //Static method.
    {
```

```
    return "Hello @ " + name + " Welcome to Dotnet Delegates";
  }
  Static void Main (string[] args)
  {
  GreetingDelegate gd = new GreetingDelegate (MyDelegateClass.
                                    Greetings);
    string GM = gd ("Hema");
    Console.Write (GM);
    Console.Read();
  }
}
}
```

delegate obj — Delegate method — class name — Method name

In the above example, we create one delegate. Then we instantiate that delegate while we are instantinating the delegate we are passing the method name (Greetings) as a parameter to the constructor of delegate.

As of now, we are following to bind a method to a custom delegate & execute it.

An anonymous method in C#.Net is also related to a delegate. without taking a named block (function or method) to a delegate, we can also bind a code block (unnamed code block) to a delegate which is nothing but an anonymous method (nameless) in C#.

Ex: Example for Anonymous Method using Single delegat

```
using System;
namespace Delegates
  {
public class AnonymousMethods
```

```
{
  public delegate String GreetingsDelegate(String name);
  Static void Main(string[] args)
    { // without using method we are calling an anonymous method using
      GreetingsDelegate gd = delegate(String name)   delegate object.
        {
          return "Hello @" + name + "Welcome To Anonymous
                 method Delegate";
        }; // delegate object.
      String GM = gd("Hema");
      Console.Write(GM);
      Console.Read();
    }
}
}
```

In the above Example instead of a method we are binding the delegate to an unnamed code block which is also called as anonymous method.

& - in C# the anonymous method are created by using delegate keyword.

Advantage: Less in code in programming

Limitation: * Anonymous method cannot contain jump statements like goto, break & continue.

* cannot use ref & out parameters.

## Lambda Expressions:

Lambda Expressions in C# is the shorthand for writing the Anonymous functions.

* To simplify the anonymous functions.
* The previous examples of delegates and anonymous methods can be converted into lamda expressions.
* To create a lambda expression in C# we need to ~~simplify~~ specify the input parameter (if any) on the leftside of the lambda operator ⇒ and we need to put the expression or statement block within open & close "{ }" curly braces.
* We replace the delegate keyword with operator (⇒) and we also remove datatype of the parameters as the delegate knows the type of parameter.

```
//As of anonymous method we written like this
Greetingsdelegate gd= delegate (string name)
    {
    return "Hello@" + name + "Welcome To Anonymous Delegate";
    };
//with lambda.
Greetingsdelegate gd = (name) =>
    {
    return "Hello @"+ name+ "Welcome to Lambda Delegate";
    };
```

Anonymous Method ↑ ↓ Converted to

Lambda expression.

Ex:

```
using System;
namespace LambdaExpressionExample
{
  class LambdaExpression
    {
    public delegate string GreetingsDelegate (string name);
    static void Main (string[] args)
      {          // lambda expression with anonymous method.
        GreetingsDelegate gd = (name) =>
                {
                  return "Hello @" + name + " Welcome to
                                  Lambda expression";
                };
        string GM = gd("Hema");
        Console.Write(GM);
      }
    }
}
```

(2)

```
using System;
namespace LambdaExample
{
  class LambdaExpression
    {
    public delegate ~~string~~ double Square (double x);
    static void Main (string[] args)
      {
        Square Obj = (x) => // lambda expression with
                                anonymous method.
          {return x * x;
          };
```

```
            double res = obj(5);
            Console.Write(res);
        }
    }
}
```

③

```
public class LambdaExpression
{
    public delegate int Sum(int x, int y);
    Static void Main(string[] args)
    {
        Sum obj = (x,y) =>
        {
            return x+y;
        };
        Console.Write(obj(10,20));
    }
}
```

(Lambda ↑ ; parameters ↑ ; delegate object)

```
        double res = obj(5);
        Console.Write(res);
    }
  }
}
```

③

```
public class LambdaExpression
  {
    public delegate int Sum(int x, int y);
    Static void Main(string[] args)
    {                          ↑ lambda
      Sum obj = (x,y) =>
        {
          return x+y;
        };
        Console.Write(obj(10,20));
                      ↑ parameters
                      ↓ delegate object
    }
}
```

Multi-Cast Delegate:

* Multi-cast Delegate return type should be void because it can hold only "void" methods.

Ex:

```
using System;
namespace Delegate
 {
  class MultiCastDelegate
   {
    pub internal void Method1()
     {
      Console.WriteLine("Method1 is calling");
     }
```

```
internal void Method2()
{
  Console.WriteLine("Method2 is calling");
}

internal delegate void MultiDelegate();
public static void Main(String[] args)
{
  Multicast mobj = new Multicast();
  MultiDelegate delobj = new MultiDelegate(mobj.Method1);
  delobj += new MultiDelegate(mobj.Method2);
  delobj(); // Method 1 , Method 2
  delobj -= new MultiDelegate(mobj.Method1);
  delobj(); // Method 2
  delobj += new MultiDelegate(mobj.Method1);
  delobj(); // Method 2 Method 1
  delobj -= new MultiDelegate(mobj.Method2);
  delobj(); // Method 1
```

class object

class object Method1

delegate obj

o/p:

```
Method 1
Method 2
Method 2
Method 2
Method 1
Method 1
```

## Extension Methods:

Basically, extension is extending the functionality of a class. Before extension methods inheritance is an approach that is used for extending the functionality of a class i.e; if I want to add any new members to an existing class without making modifications to the class, we define child class to that existing class & then add new members in child class.

* In this case of an extension method, we will extend the functionality of an existing class.
* In this case we will create a new approach of creating a New class and by using that New class we will extend the functionality of an existing class.
* Both this approaches can be used for extending the functionalities of an existing class whereas in case of inheritance we call the methods defined in the old & new classes. by using Object of a new class whereas in case of ~~extend~~ extension method we call the old & new methods by using the object of a old class
* Generally Sealed, if we defined then we cannot derive members to the derived class. But we can extend the Functionality of ~~Seleo~~ Sealed by using Extension Methods.

### Steps to implement Extension methods:

→ Open an console application & write code in .cs file.

```
using System;
namespace ExtensionMethods
{
public class oldClass
    {
      public int x = 100;
      public void Test1()
        {
          Console.WriteLine("Method One : " + this.x);
        }
      public void Test2()
        {
          Console.WriteLine("Method two : " + this.x);
        }
    }
```

→ Now our requirement is to add 3 new methods to the old class.
As of now, if you dont have the source code, so we cannot change of oldclass. Then I need to add some methods in old class using the old class object, that is where the Extension method come into picture.
So extend the functionality with the help of Extension methods.

→ Create a Newclass.cs & write the code.

```
using System;
namespace ExtensionMethods
{
public static class NewClass
    {
```

```
public static void Test3(this OldClass o)
{
  Console.WriteLine("Method three");
}
public static void Test4(this OldClass o, int x);
{
  Console.WriteLine("Method Four" + x);
}
public static void Test5(this OldClass o)
{
  Console.WriteLine("Method Five" + o.x);
}
}
}
}
```

→ Create an objects of the class & access the methods

Same Namespace

```
using System;
namespace ExtensionMethods
{
public class Program
{
  static void Main(string[] args)
  {
    OldClass obj = new OldClass();
    obj.Test1(); // 100.
    obj.Test2(); // 100.
    //calling the Extension Methods.
    obj.Test3(); // Method three.
    obj.Test4(10); // 10
    obj.Test5(); // 100.
  }
}
}
```

## Exception Handling:

Basically when a program is executing there they are different types of Errors can exist like

1. Syntax error.
2. Logical error.
3. Runtime error.

Syntax error:

* This Syntax errors are nothing, Compilation errors when we not follow the rules of the programming languages this errors will occurs & the program doesn't execute until you clear the Errors. These errors are identified by the compiler.

* Logical Error:
* We also have Logical errors, compilers never identify this errors this are purely mistake by the user.

Runtime errors:

* Runtime errors (execution errors) also which are occurred at the time of program execution are called Runtime errors.
* These errors are occurs at runtime due to various reasons such as when we are entering the wrong data into a variable,
  trying to open a file for which there is no permission,
  Finding the square root for Negative numbers,
  giving the array elements out of the range, etc.
* these runtime errors are dangerous when they occur the program, Abnormally terminated on same line

where the error occured without executing remaining program.

* If you want to take the control into the user Hands rather defined predefined Exceptions try to write program code with the Exception Handling.

Q) What is an Exception in C#?

Exception is nothing but runtime error.

(or)

An exception in a class in C# which is responsible for abnormal terminations of the program when runtime errors occur while running the program.

(or)

Exception handling in C#.Net is a mechanism to handle runtime errors or exception that occur during the execution of a program.

Q) Who is responsible for the abnormal termination of the program whenever runtime error occur?

Objects of Exception classes are responsible for abnormal termination of the program whenever runtime error occurs. These Exceptions or classes are predefined under BCL where a seperate class is provided for each & every different types of exceptions like.

1. IndexOutOfRangeException.
2. FormatException.
3. NullReferenceException.
4. DivideByZeroException.
5. FileNotFoundException.
6. SQLException.
7. OverFlowException, etc.

* Each Exception class provides specific exception error message. So whenever a runtime error occurs in a program, first the Exception manager under the CLR identifies the type of Error that occurs in the program then creates an object of the Exception class related to that error & throws that object which will be immediately terminate the program abnormally.

Note: Exception class is the Superclass of all exception classes in C#.

Q) What happens if an Exception is raised in program?
Q) What CLR does when an exception occured in program?
Q) Purpose of Exception handling?

Ex:

```
// Program Execution without exception.
using System;
namespace Exceptions
{
class Program
  {
  public static void Main(string[] args)
    {
      int a=20;
      int b=10;
      int c;
      Console.WriteLine("A value: "+a);
      Console.WriteLine("B value: "+b);
      c=a/b;
      Console.WriteLine("C value: "+c);
      Console.Read();
    }
  }
}
```

o/p:

A Value : 20
B value : 10
C value : 2

② // program Execution with exception.

```
using System;
namespace Exceptions
{
  class MyClass
  {
    public static void Main(string[] args)
    {
      int a= 20;
      int b=0;
      int c;
      Console.WriteLine("A vallue : "+a);
      Console.WriteLine("B value : "+ b);
      c= a/b;  → Exception occured.
      Console.Write("C value : "+c);
    }
  }
}
```

o/p: A value : 20
B value : 0
Unhandled Exception : System.DivideByZeroException.
attempted to divide by zero.

* In the above program, it raises an error at the Statement c= a/b, pre-defined Exception is raised.
* The CLR terminates the program execution by throwing DivideByZeroException because the logical mistake we committed here dividing by zero,
Universally, it is not possible, so CLR first it will check what type of logical errors is this.
* It will find that, it will be a Divide by zero logical

error. So it CLR create an instance of DivideByZeroException class & then it will throw that instance by using throw statement. like
like throw new DivideByZeroException() by this predefined message will occur.

Q) Is it really the above Exception message User Understandable?
No, for this reason the solution to create User defined message for creating Exception handling by the User explicitly.

*How we can handle an Exception in .Net.
There are 2 methods to handle the exception in .Net.
1. Logical Implementation
2. try-catch Implementation.

# What are Logical Implementation to handle exception?
The first implementation is logical implementation only if it is not possible go for try-catch.

Ex:

```
using System;
namespace Myspace
{
  class Myclass
  {
    Static void Main(string[] args)
    {
      int n1, n2, result.
      Console.Write("Enter first number: ");
      n1 = int.Parse(Console.ReadLine());
      Console.Write("Enter second number: ");
      n2 = int.Passe(Console.ReadLine());
```

```
if(n2==0)
    Console.WriteLine("Second number should not be zero");
else
    result = n1/n2;
    Console.WriteLine($"Result= {result}");
Console.Read();
}
}
}
```

In the above program, if we pass '0' (zero) without predefined Exception, logical implementation takes place, Program it will be in your control.

But if the user pass second number as a String or out of Range value then once again predefined exception raised.

So for this to handle this type of exceptions in C# we need "try-catch" implementation. So unable to handle by logical implementation then go for "try-catch".

## Exception Handling using try-catch-finally blocks:

Syntax:

```
try
{
}
catch(<ExceptionClass> , <object_name>)
{
}
finally (optional)
{
}
```

Predefined class

User-defined

* Within the try block, we have to write the statements which may throw an error.
* Within the catch block, we have to write the code to handle the error by displaying the user friendly messages.
* Within the finally block, we have to write the error free code (or) the code which we want to execute irrespective of the error occurance.

# Execution flow of try, catch and finally.

* While executing try block when there is an error control will comeout from the try block & it will execute appropiate catch block then it will execute finally block.
* While executing try block if there is no error then it will execute full try block and it will skip catch block & it will execute finally block.
* catch block will execute only when there is an error, by finally block will execute always irrespective of error occurance.
* A try block can follow with 1 catch block or multiple catch blocks & with finally (or) without finally.

Example:

```
// Example using exception handled by user.
using System;
namespace Exception
 {
  class handling
   {
    public static void Main(string[] s)
     {
```

```
int n1, n2, result;
try
{
  Console.Write("Enter First number : ");
  n1 = int.Parse(Console.ReadLine());
  Console.Write("Enter Second number : ");
  n2 = int.Parse(Console.ReadLine());
  result = n1/n2;
  Console.WriteLine($"Result : {result}");
}
Catch
 {
  Console.WriteLine("Some error occured");
 }
Console.Read();
  }
 }
}
```

In the above program, we define some generic (common) catch block, for any type of error, same error message will occurs.

If you want specific error messages, use the Exception class. with some properties.

The Exception class properties are :

1) Message
2) Source
3) Help link
4) Stack Tree.

Example: // Exception Handling with exception class properties.

```
using System;
namespace Exception
{
 class Handling
  {
   public static void Main(string[] args)
    {
     int n1, n2, result;
     try
      {
       Console.Write("Enter first number : ");
       n1= int.Parse(Console.ReadLine());
       Console.Write("Enter Second number: ");
       n2 = int.Parse(Console.ReadLine());
       result = n1/n2;
       Console.Write($" Result : {result} ");
      }
     catch(Exception object_Ex)
       {
        Console.WriteLine("Message" + Object_Ex.Message);
       }
     Console.Read();
    }
  }
}
```

Exception class ↑ Superclass; object ↑ object; Property of ↑ Exception class.

The Exception class is a Super class which handle all exceptions from the corresponding try block, it is not giving change for d child class Exceptions for this reason, implement Multiple catch blocks

```
// Example on Multiple Catch blocks.
using System;
namespace Exception
{
 class MultipleCatch
  {
  public static void Main(string[] args)
   {
    int n1, n2, result;
    try
    {
     Console.Write("Enter n1 : ");
     n1 = int.Parse(Console.ReadLine());
     Console.Write("Enter n2 : ");
     n2 = int.Parse(Console.ReadLine());
     result = n1/n2;
     Console.Write($" Result : {result} ");
    }
    catch(DivideByZeroException)   // child class
      {
       Console.Write("Second number cannot be zero");
      }
     catch(FormatException)
      {
       Console.Write(" Number should be integer ");
      }
     catch(Exception obj)
        {//Console.Write("Some error occured");
         console.Write(obj.Message);
        }
    finally { console.Write("I am optional, I will execute any time "); }
   }
  }
}
```

* It is always recommend to write the Exception class Catch block at the end.

O/p:

① n1: 9989765432 1
Value was either too large or too small for an Int32.
I am optional, I will execute anytime.

* If it not satisfied with the predefined exceptions you can create the Customized Exception.

Q) Is it possible to catch all exceptions using a single catch block?
Yes, The Exception class is the superclass of all Child Exception classes & hence it can handle all types of Exceptions throw in the try block. We need to use this catch block only for stopping the abnormal termination irrespective of exceptions.

Q) When should we write Multiple catch blocks for a single try block?
1. To print message specific to an exception (OR)
2. To Execute some logic specific to an exception.

Combinations:

1) try

Combinations:

1) try - catch
2) try - catch - finally
3) try - finally:

Q) Why do we need finally block in Real-time project development?

As per Industry coding standard within finally block we need to write the resource releasing logic or clean up the code.

Ex: If we want to close ADO.NET objects such as Connection object, Command object, etc. we must call the close() method. in both the try as well as in the catch block so place only in finally block it compulsory execute.

Note:

If you want throw an exception expliatly use 'throw' keyword.