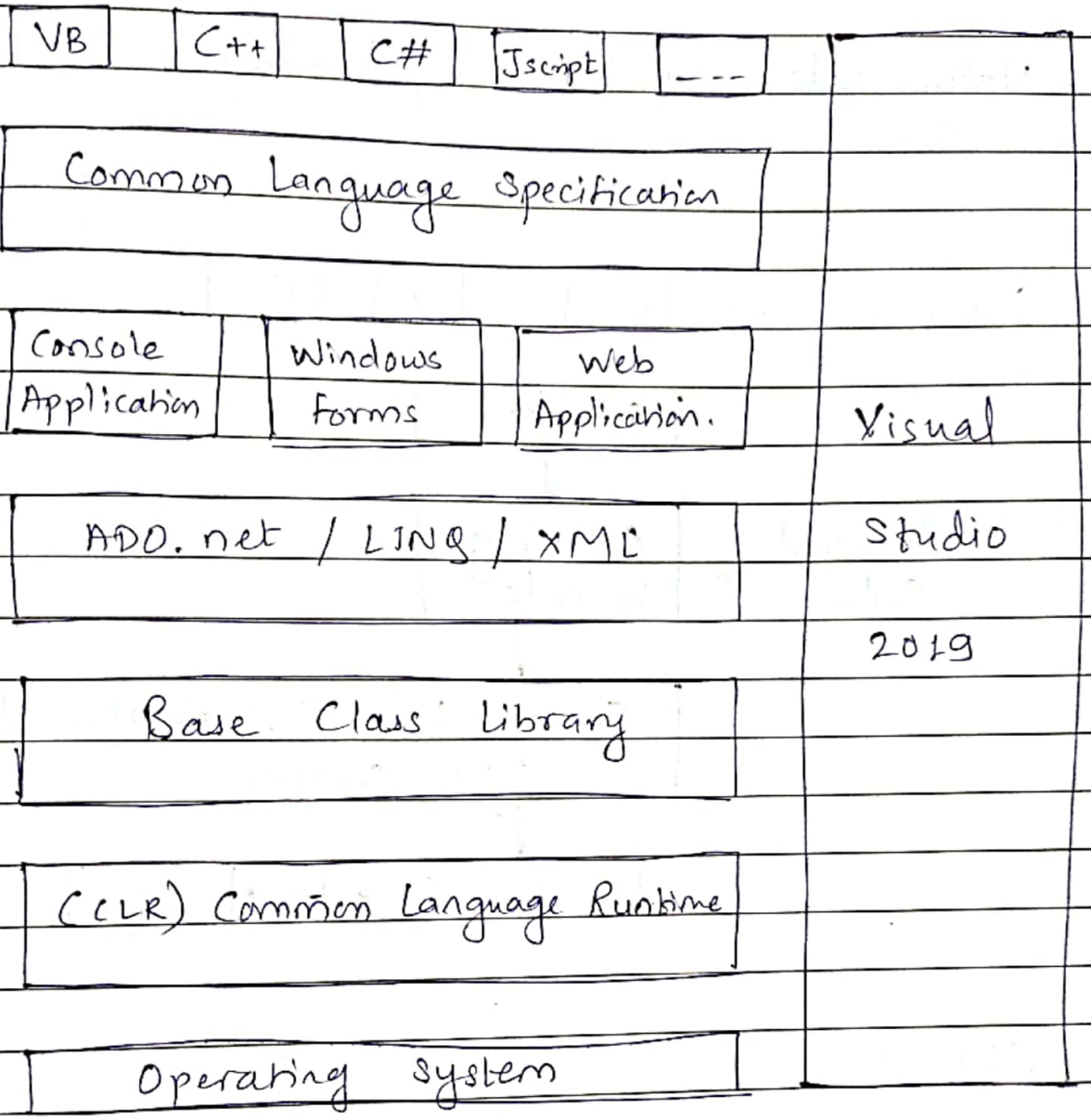
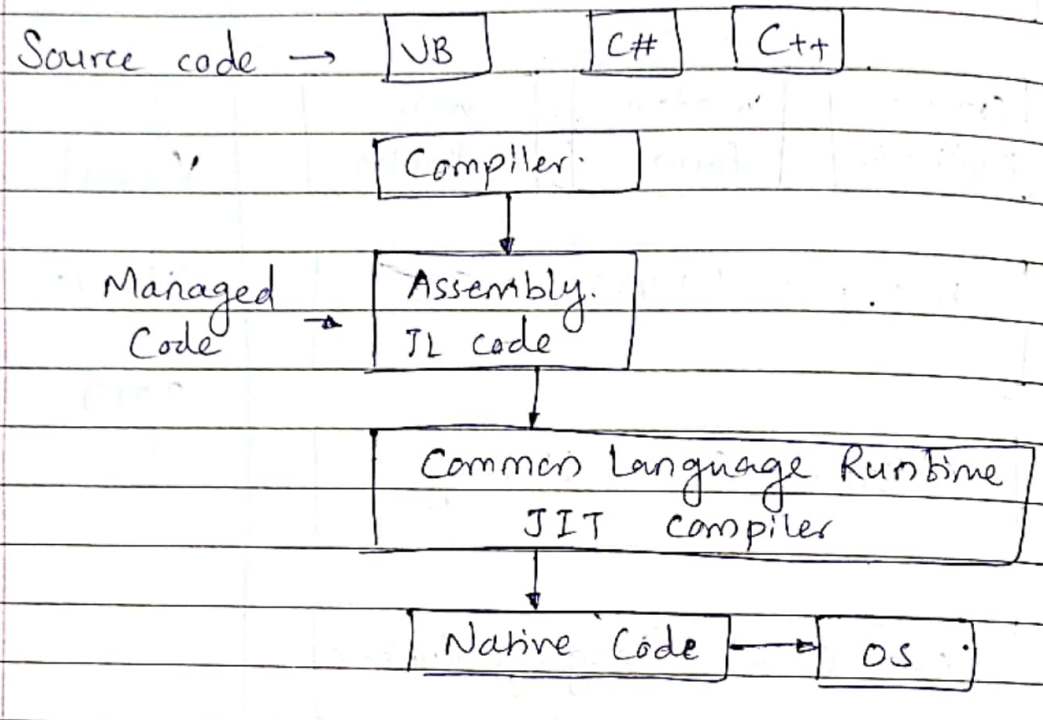
.NET FRAMEWORK



JIT- Just in time

ADO- Active Data Object

Native code- machine code/ language.



CLR :-

- It perform memory management, exception handling, debugging, security checking, thread execution, code execution, code safety, verification & compilation.
- The code that is directly managed by CLR is called managed code.
- When managed code is compiled the compiler convert source code into a CPU independent intermediate language code.
- A JIT compiler compile IL code into native code which CPU specific.

C# (C sharp):-

- Microsoft developed
- runs on .Net framework.
- simple, modern, object oriented
- combined feature VB, C++, java

Feature :-

+ simple
+ Object oriented
+ Structure programming language.
+ Fast speed.

Program:-

```
System.Console.WriteLine("Hello World");
```

Data Types:-

* Object :-

```
int id = 10
object obj = id  ← boxing
float Employeeid = float (obj)  ← Unboxing

Console.Writeline (obj)
Console.Writeline (Employeeid)

Object obj-str = "Trupti";
Object float   = 6.00f ;
```

* Value data type → Predefined → Integer / Boolean / float
  → User defined → Enum / structure

* Pointer data

* Referee Data → Predefined → object / string / Dynamic
  → User defined → class / Interface / Delegates

→ Boxing = When variable of value type get converted into object type is called Boxing.

→ Unboxing = When variable of type object get converted into value type then it is called as unboxing.

Value type :-

The value types directly contain data. Some examples are int, char and float, which stores numbers ~~op alphi~~ alphabets and floating point numbers, respectively

- User Defined value type :-

① Structure = A struct type is a value type that is typically used to encapsulate small groups of related ~~var~~ variables contain constructors, constants, fields, methods, properties, operators, events and nested types

② Enumeration = The enum keyword is used to declare an enumeration, a distinct type that consists of a set of named constants called the enumerator list.

- Reference type :-

Variables of reference types stores references to their data while variables of value types directly contain their data.

* Predefined Reference type :-

- Dynamic type
- Object type
- String type
- Array type.

* User defined Reference type :-
  - class type
  - Interface type.
  - Delegate type.

Variable :-
- A variable is a name of memory ~~location.~~ location.
- It is used to store data.
- It's value can be changed and it can be reused many times.

→ Rules for defining variable :-

- A variable can have alphabets, digits and underscore.
- A variable name can start with alphabet and underscore only. It can't start with digit.
- No white space is allowed within variable name.
- A variable name must not be any reserved word or keyword e.g. char, float, etc.

Arrays :-

- Array is a group of similar types of elements that have contiguous memory location.
- Array is an object of base type System.Array
- Array index starts from 0. (zero).
- We can store only fixed set of elements in array.

→ Ex

```
using System;
public class ArrExample
{
  public static void Main (string[] args)
  {
    int[] arr = new int [5]; // Creating Array.
    arr [0] = 10;
    arr [2] = 20;
    arr [4] = 40;

    // traversing array.

    for (int i=0; i<arr.length ; i+1)
    {
      Console.WriteLine (arr [i]);
    }
  }
}
```

→ Ex

```
class arrExample
{
  static void Main (string[] args)
  {
    int[] arr = { 10, 20, 30, 40, 50 };
    int[] arrr = new int [5];
    int [] arrr = new int [5] { 20, 30, 50, 60, 70 };
```

```
int[] arrr = new int[] {20,30,50,60,70,70,100,22};

double[] arrdo = new double[] {20.30, 30.26, 50.25, 70.56};
decimal[] arrde = Array.ConvertAll(arrdo, x =>
                                   (decimal)x);

arr[2] = 100;
Console.WriteLine(arrr[4]); //access only one element
                              in array;


for (int i=0; i<arr.length; i++)
{
    Console.WriteLine(arr[i]);
}
```

- Example :-

① string userName = Console.ReadLine();
Console.Writeline("User name is = " + userName);

② double Emp1d = 20;
Decimal emp1d = Convert.ToDecimal(Emp1d);

Structs :-

- Structure is defined using struct keyword
- Using struct keyword ~~one~~ one can define the structure consisting of different data types in it.
* Hold a bunch of different data types.
- A structure can also contain constructors, constants, fields, methods, properties, indexers and events etc.

ReadLine → input

WriteLine → output

PAGE No.
DATE / /

→ Ex

```
Using System;
namespace Console Application.
{
    // Defining structure.
    (public) struct Person.
    {
     // Declaring different data types
     (public) string Name;
     (public) int Age;
     (public) int Weight;
    }

    class Struct Example
    {
      //Main method
      Static void Main (string [] args)
      {
       // Declare P1 of type Person
       Person P1;
       //P1's data
       P1.Name = " Keshav Gupta";
       P1.Age = 21;
       P1.Weight = 80;

       // Displaying the values.
       Console.WriteLine ("Data Stored in P1 is" +
                          P1.Name + ", age is " +
                          P1.Age + " and weight is"
                          + P1.Weight );
```

Enum:-

The main objective of enum is to define our own data types (Enumerated Data types).

Enumeration is declared using enum keyword directly inside a namespace, class or structure.

- Ex

```
using System;
namespace ConsoleApplication8
{
  //Making an enumerator 'Month'
  enum month
  {
    // following are the data members
    jan,
    feb,
    Mar,
    apr,
    may
  }
  class Program
  {
    // Main method.
    static void Main (string[] args)
    {
      // getting integer values of all data members.
      Console.WriteLine ("The value of jan in
                          month " + " enum is " +
                          (int) month.jan);
```

```
        Console.WriteLine("The value of feb in month" +
                          "enum is " + (int)month.feb);
        Console.WriteLine("The value of Mar in month" + "enum
                          is" + (int)month.mar);
        Console.WriteLine("The value of Apr in month" + "enum
                          is" + (int)month.apr);
        Console.WriteLine("The value of May in month" +
                          "enum is" + (int)month.may);
    }
  }
}
```

Control Statement :-

* Decision Making :-

1) if statement
2) else - if statement
3) nested if statement
4) if - else - if ladder.

1) if statement =

```
if (condition)
{
    // Code to be executed
}
```

2) if - else statement

```
if (condition)
{
    // code if condition is true
}
else
{
    // code if condition is false.
}
```

3) if - else - if ladder

```
if (condition 1)
{
  // code to be executed if condition 1 is true.
}
else if (condition 2)
{
  // code to be executed if condition 2 is true
}
else if (condition 3)
{
  // code to be executed if condition 3 is true.
}
else
{
  // Code to be executed if all the conditions are false
}
```

Loop:-

1) For Loop

```
for (initialization ; condition ; incr/decr)
{
    // code to be executed
}
```

2) while loop

```
while (condition)
{
    // code to be executed.
}.
```

3) do -- while loop

```
do
{
    // code to be executed
}
while (condition)
```

4) Switch case

```
switch (expression)
{
    case Value1 : // code to be execued
                  break;
```

```
        case Value 2 : // Code to be executed
                       break;
        case Value 3 : // code to be executed
                       break;
        default : // code to be executed if all cases
                  are not matched;
                  break;
    }
```

Continue Statement:-

It continues the current flow of the program and skips the remaining code at specified condition.

```
public static void Main (string[] args)
{
    for (int i=1; i<=10; i++)
    {
        if (i==5)
        {
            Condition;
        }
        Console.WriteLine (i);
    }
}
```

go to statement :-

- go to statement is also known jump statement
- It is used to transfer control to the otherpart of the program.

- Ex

```
public class GoToExample
{
  public static void Main (string[] args)
  {
    ineligible :
    Console.WriteLine ("Enter your age: \n");
    int Age = Convert.ToInt32 (Console.ReadLine ());

    if (age < 18)
    {
      Console.WriteLine ("You are not eligible to vote!");
      go to ineligible;
    }
    else
    {
      Console.WriteLine ("You are eligible to vote!");
    }
  }
}
```

Loops :-

1) For Loop :- The for Loop is used to iterate a part of the program several times. If the number of iteration is fixed, it is recommended to use for loop than while or do-while loops.

2) While loop :- While loop is used to iterate a part of the program several times. If the number of iterations is not fixed, it is recommended to use while loop than for loop.

3) Do-while loop :- The C# do-while loop is used to iterate a part of the program several times. If the number of iteration is not fixed and you must have to execute the loop at least once, it is recommended to use do-while loop. The C# do-while loop is executed at least once because condition is checked after loop body.

Object & Class :-

- Object is an entity that has state and behaviour. Here, state means data and behaviour means functionality.
- Object is a runtime entity, it is created at runtime.
- Object is an instance of a class. All the members of the class can be accessed through object.

• Syntax :-

```
class_Name object_name = new class_name();
```

• Ex

```
Student S1 = new student();
```

(creating an object of student).

student is the type and S1 is the reference variable that refers to the instance of student class. The new keyword allocates memory at runtime.

Class :-

Class is a group of similar objects. It is a template from which objects are created. It can have fields, methods, constructors etc.

• Syntax :-

```
public class className.
{
  ..
}
```

• Ex :-

```
public class classObjDemo
{
  int id;    // instance variable
  string name;
  float weight;
```

```
    public static void Main(string[] args)
    {
        classObjDemo refvar = new classObjDemo();
        refvar.id = 1;
        refvar.name = "Rajeev"
        refvar.weight = (float) 22.02;

        Console.WriteLine (refvar.id);
        Console.WriteLine (refvar.name);
        Console.WriteLine (refvar.weight);
    }
  }
}
```

Access Modifier :-

Access Modifier	Applicable to the application (namespace)	Applicable to the current class	Applicable to the derived class	Applicable to outside the namespace / assembly	Applicable to outside the namespace but in derived class.
PUBLIC	Yes	Yes	Yes	Yes	Yes
PRIVATE	No	Yes	No	No	No
PROTECTED	No	Yes	Yes	No	Yes
INTERNAL	Yes	Yes	Yes	No	No
PROTECTED INTERNAL	Yes	Yes	Yes	No	No

Access Modifiers or specifiers are the keywords that are used to specify accessibility or scope of variables and functions in the application.

C# provides five types of access modifiers:-

1) Public
2) Private
3) Protected (not applicable for class) (Only member & member func)
4) Internal
5) Protected Internal (only member & member function)

1) Public:- It specifies that access is not restricted.

2) Protected:- It specifies that access is limited to the containing class or in derived class.

3) Internal:- It specifies that access is limited to the current assembly.

4) Protected Internal:- It specifies that access is limited to the current assembly or types derived from the containing class.

5) Private:- It specifies that access is limited to the containing type.

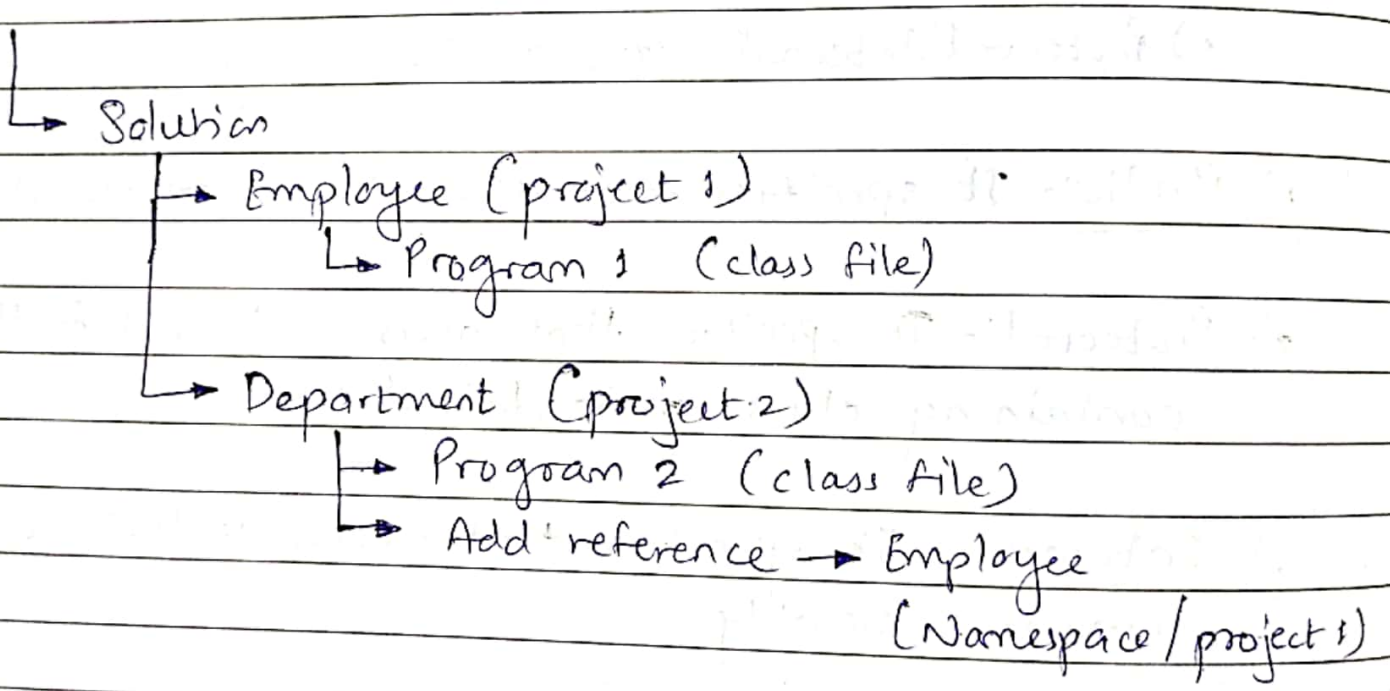
Project Name → Employee Dept

Location → D:\Access Modifier\

Solution Name → PublicAccess Modifier.

class Name → Employee Details.

(right click → solution → properties)



#

Protected :-

→ Ex // Applicable to the derived class.

Project 1

```
class empContact
{
    protected string contactNo = "123456789";
}
class Program1 : empContact
{
    protected int id = 1;

    protected void fun1()
    {
      Console.WriteLine ("Function 1");
    }

    static void Main (string[] args)
    {
      Program1 objP1 = new Program1();
      Console.WriteLine ("same class" + objP1.id);
      objP1 . fun1();
      Console.WriteLine ("Rule no 3" + objP1. ContactNo);
      Console.Readkey();
    }
}
```

DATE / /

Project 1

Project 2

```
public class empContact
{
    --
}
class Program2 : empContact
{
    Static void Main (string[] args)
    {
        Program2 objP2 = new Program2 ();
        Console.WriteLine ("Rule no 5" + objP2.contactNo);
        Console.Readkey ();
    }
}
```

Internal :-

- Ex

```
class Program1
{
    internal int id = 1;

    internal void fun1 ();
    {
        Console.WriteLine (" Function 1 ");
    }
```

```
static void Main (string[] args)
{
    Program1 objP1 = new Program P1 ();
} Console.WriteLine ("same class" + objP1.id);
    objP1.fun1();
    Console.ReadKey();
}
```

→ Ex

```
namespace Employee

internal class EmployeeContact
{
    internal string contactNo = "123456789";
}
class Project1
{
    ==
    static void Main (string[] args);
    {
        Employee Contact objCon = new Employee Contact ();
        Console.WriteLine ("Different class" + objCon.contactNo);
    }
}
```

Protected Internal:-

```
namespace Employee
{
  class EmployeeContact
  {
    protected internal string contactNo = "123456789";
  }
  class Program1 : EmployeeContact
  {
    protected internal int id = 1;

    protected internal void fun1()
    {
      Console.WriteLine ("Function 1");
    }
    static void Main (string[] args)
    {
      Program1 objP1 = new Program1 ();
      Console.WriteLine ("Same class" + objP1.id);
      objP1.fun1();
      Console.WriteLine ("Inherited class" + objP1.contactNo);

      EmployeeContact objCon = new EmployeeContact ();
      Console.WriteLine ("Different class" + objCon.contactNo);
      Console.Readkey();
    }
  }
}
```

Static :-

- It is applicable to the class and members (variables, functions).
- Static class :- In this, there is no need to create an instance of a class (object)

- Important points about static :-

① Static class contains only static members, which means we can't create non-static variable, non-static functions.
② We can't inherit static class.
③ Inside static function, each member should be static.
④ Inside static class, each member should be static.

→ Ex

```
namespace staticExample

class Program
{
    static int static_var = 10;
    int nonstatic_var = 20;

    void normalFunction ()
    {
        Console.WriteLine (static_var);
        Console.WriteLine (Nonstatic_var);
    }
```

```
static void staticFunchian()
{
    Console.WriteLine (static_var);
    Console.WriteLine (nonstatic_var);
}
static void Main (string[] args)
{
    Console.WriteLine (static_var);
    Program objP = new Program();
    Console.WriteLine (objP. nonstatic_var);
    Console.WriteLine (" ------ funchian ------ ");
    objP.normal funchian ();
      static funchian ();

    Console.ReadKey();
}
}
```

Constructor:-

- Constructor is a special type of method and constructor is method which gets called when object of a class gets created.
- It is used to initialize class members or to initialize resources.

Rules of constructor:-

1) Constructor name is a same as class name.
2) It should not return value.
3) We can't call constructor from program directly.
4) It is not necessary / compulsion to declare a constructor.
5) If we will not provide any constructor then default constructor called automatically.
6) If we provide any constructor then compiler will not provide you default constructor or vice versa.

→ Ex

```
namespace ConstructorExample.

class Demo
{
    private int i, j;

    public Demo()          // Demo(this)
    {
      Console.WriteLine("Inside Default constructor");
      i = 10;
      j = 20;
    }
    public Demo(int x, int y) // Demo(this, x, y)
    {
      Console.WriteLine("Inside Parameterized constructor");
      i = x;
      j = y;
    }
```

```
public Demo (Demo Copy)      // Demo (this, x.y)
{
    Console.Writeline ("Inside Copy constructor");
    i = copy.i;
    j = copy.j;
}
~Demo ()
{
    Console.Writeline ("Inside Destructor");
    Console.ReadLine ();
}
}
class Program
{
    public static void Main ()
    {
        Demo obj1 = new Demo ();        // Demo (obj1)
        Demo obj2 = new Demo (30,30); // Demo (obj2,30,30)
        Demo obj3 = new Demo (obj2);
        Console.ReadKey ();
    }
}
```

Destructor:-

- It is used to deinitialize the allocated resources
- Destructor name is also same as class name.
- It don't have return value.
- There is no need to call destructor from a program.
- '~' operator is used to ~~call destructor~~ write destructor
- After calling destructor, CLR invoke (call) one feature called as garbage collection.

Inheritance :-

- It is main principle of OOP's, that allows you to inherit the feature of some another class. which ~~allows~~ means reusuability.
- In this, to inherit the class ':' operator is used.
- Original class called as Base class. and new class is called as child class or derived class or sub class.

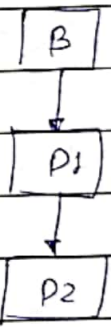
Types of Inheritance :-

1) Single level
2) Multilevel
3) Hierarchical

1) Single level :-


```
class B
{ --
}
class D : B
{ --
}
```

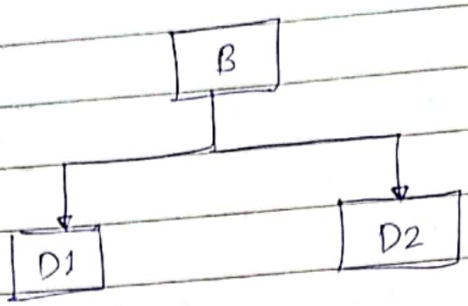
2) Multilevel =




```
class B
{ --
}
class D1 : B
{ --
}
class D2 : D1
{ --
}
```

DATE / /

3) Hierarchical :-




```
class B
{ --
}
class D1 : B
{ --
}
class D2 : B
{ --
}
```

ABSTRACT CLASS :-

Abstraction means → Hiding something from the outside the world.

→ Rules :-

- can't create an object of abstract class.
- abstract class = combo of complete and incomplete method.
- abstract class have zero or more methods.
- used this two key word before function declaration "public abstract"

- Program:-

```
namespace AbstractClassExample
{
    abstract class Demo1
    {
        public void fun()
        {
            Console.WriteLine("Inside Funchian");
        }
        public abstract void gun();
    }
    class hello : Demo1
    {
        public override void gun()
        {
            Console.WriteLine("Inside gun Funchian");
        }
    }
    Class Program
    {
        Static void Main (string [] args)
        {
            // Demo1 obj = new ~~dem~~ Demo1 ();
            hello obj2 = new hello ();
            obj2 . gun();
            obj2 . fun();
            Console. Readkey();
        }
    }
}
```

Interface :-

- Rules :-
  - Interface contains function declaration only
  - By default all function are public and abstract
  - can't create an object
  - class → can inherit multiple interface
  - interface → can't inherit multiple class
  - interface → can't inherit multiple interface
  - interface → can't inherit class.
  - interface declared by 'interface' keyword.

- Program :-

```
namespace InterfaceExample
{
  interface Interface1
  {
    void fun1Inter1();
    void fun2Inter1();
  }
  interface Interface2();
  {
    void function1Inte2();
    void function2Inte2();
  }
  class clsDerived : Interface1, Interface2
  {
    public void fun1Inter1();
    {
      Console.WriteLine("Function 1 in Interface 1");
    }
```

```
    public void fun2Inter1()
    {
      Console.WriteLine(" Function 2 in Interface 1");
    }
    public void function1Inte2()
    {
      Console.Writeline (" Function 1 in Interface 2);
    }
    public void function2Inter2()
    {
      Console.Writeline (" Function 2 in Interface 2);
    }
}
Class Program
{
   static void Main (string [] args)
   {
      clsDerived D1 : new clsDerived ();
      D1. Fun1Inter1 ();
      D1. Fun2Inter1 ();
      D1. function1Inte2 ();
      D1.. function2Inte2 ();
      Console. Write
      Console. Readkey ();
   }
}
}
```

DATE / /

Property:-

- Inside class, if variable or field is declared as public then anyone can accept and anyone can modify. So it is not secure. If we create private ~~var~~ variable or field no one can access or modify. If you want to access private member, we can use two functions 'get' and 'set'.

- Program:-

```
namespace PropertyExample.
{
  class Employee
  {
    private string empName;
    private int empID;

    public string propertyEname      //Property
    {
      set
      {
        empName = Value;
      }
      get
      {
        return empName;
      }
    }
    public int propertyID
    {
```

Sum of {1} & {2} is {0}", (num1+num2),
num1, num2);

⇒ sum of 10 & 20 is 30

```
        set
        {
            empID = value;
        }
        get
        {
            return empID;
        }
    }
}
class Program
{
    static void Main (string [] args)
    {
        Employee Emp = new Employee ();
        Emp.propertyEname = "ABC";
        Emp.propertyID = 101;
        Console.WriteLine ("Employee name is : {0}",
                                Emp.propertyEname);
        Console.WriteLine ("ID is : ", Emp.propertyID);
        Console.Readkey();
    }
}
}
```

DATE / /

Polymorphism:-

-ex

```
class cls_fun_overloading
{
① void fun()
  { ..
  }
② void fun( int parameter 1)
  { ..
  }
③ void fun (int parameter 1, float parameter 2)
  { ..
  }
④ void fun (float parameter 1)
  { ..
  }
⑤ void fun( float parameter 1, int parameter 2)
  { ..
  }
}
```

Function Overloading:-

Defining multiple behaviours inside same class with the same name is called as function overloading

Function overloading = Compile time polymorphism

- Rules :- (function overloading)

1) We can overload the function by changing its datatype of input argument (② & ④)
2) We can overload the function by changing no.of argument. (② & ③)
3) We can overload the function by changing sequence of argument
4) We can't overload function by changing it's return value

ex

```
int fun (int Parameter1)
{
--
}
```

5) We can't overload function by changing ~~its~~ its access modifier.

ex

```
public void fun (int parameter 1)
{
--
}
```

- Method Overriding :-

- Rules :-

1) In this type of polymorphism, function binding is perform during runtime.
2) Run time polymorphism is also called as late binding.
3) To achieve the concept of runtime polymorphism we have to ~~overriding~~ override it.

4) If same name and protype function is defined across the classes with 'virtual' keyword in base class and 'override' keyword in derived class. Then it's called as 'overriding'.

→ Program:-

```
class clsBase
{
    public virtual void fun()
    {
        Console.WriteLine ("Base class ");
    }
}
class clsDerived : clsBase
{
    public override void fun()
    {
        Console.WriteLine ("Derived Class");
    }
}
class Program
{
    Void Main()
    {
        clsDerived obj = new clsDerived();
        obj.fun();
        Console.ReadKey();
    }
}
```

Method Shadowing:-

We can reimplement a parent / base class method in child / derived class method, by using two ways.-

1. method overriding (by using override keyword)
2. method shadowing (by using new keyword)

ex (method shadowing)

```
class clsBase
{
  public void fun()
  {
    Console.WriteLine("Base class");
  }
}
class clsDerived : clsBase
{
  public new void fun()
  {
    Console.WriteLine("Derived class");
  }
}
class Program
{
  Void Main()
  {
    clsDerived objDerived = new clsDerived();
    objDerived.fun();
    Console.Readkey();
  }
}
```

Base keyword:-

It is used to access the member of base class within derived class.

ex

Program:-

```
class clsBase
{
  int a = 10;
  public void fun()
  {
    Console.WriteLine("Base Class");
  }
}
class clsDerived : clsBase
{
  int a = 25;
  public void derived_function()
  {
    Console.Writeline("a");
    Console.Writeline(base.a);
    fun();
    base.fun();
    Console.Readkey();
  }
  public void fun()
  {
    Console.writeline("Derived class");
  }
}
```

object "can" created	object "can't" created
sealed, normal class, derived class	Abstract, static, interface.

```
class Program
{
    Void Main()
    {
        cls Derived objDerived = new cls Derived();
        objDerived.derived_function();
        Console.Readkey();
    }
}
```

Sealed class:-

Sealed class can't be inherit
object can be created of sealed class to access the member and member function.

- It is a class that can't be inherit. Then you can declare a class as a sealed class. If you want to prevent other users from creating derived class from that class
- The "sealed" keyword is used to indicate that class is sealed and cannot be inherit.
- We can create object of sealed class.

ex

Method Reimplement

'can' reimplement	'can't' reimplement
virtual class, shadowing, method overriding	sealed

PAGE No.
DATE / /

Collection in classes:-

There are two types of collection in class

① Generic collection.
② Non-Generic collection.

Generic collection = stack, queue, linked list, array are example of generic class.

ex

→ Program:-

```
class Program
{
  static void Main (string[] args)
  {
    int[] arr = {10,20,30,60,50};
    Array.Reverse (arr);
    for (int i=0 ; i<arr.length ;;++)
    {
      Console.Writeline (arr[i]);
    }

    stack <int> s = new stack <int> ();
    s.Push (10);
    s.Push (20);
    s.Push (30);
    s.Push (40);
    s.Push (50);
```

```
foreach (var item in s)
    Console.WriteLine (item + ' , ');
Console.WriteLine ("\n");

List <int> d = new List <int> ();
d.Add (20);
d.Add (30);
d.Add (40);
for (int p=0; p<d.count ; p++)
{
  Console.WriteLine (d [p]);
}
Console.Readkey ();
  }
}
```

Generic Function:-

Generic Concept:-

- Generic allows you to define generic class as well as generic methods. In generic class you have to write a place holder, for it's type of it's variable or methods.
- Generic class can be defined using angular brackets '<...>'
- It increases reusuability of code
- Generic are type safe.

- Program:- (Generic function)

```
using system;
namespace GenericFuncEx
{
 class Program
 {
  public void EmployeeDetails<n> (n Para)
  {
    Console.WriteLine (para);
  }
  public void display <T> (T [] para)
  {
    int i;
    for (i=0; i<para.Length; i++)
    {
      Console.WriteLine (para [i]);
    }
  }

  Static void Main (string [] args)
  {
    Program P = new Program ();

    int [] arr = { 1, 2, 3, 4, 5 };
    P. display <int> (arr);

    Char [] brr = { 'a', 'b', 'c', 'd', 'e' };
    P. display <char> (brr);
```

```
string [] str = {"ABC", "PQR", "MNO"};
P.display<string> (str);

P. Employee Details ("LMN");
P. Employee Details (12);

Console.Readkey ();
    }
  }
}
```

→ Program:- (Generic Class)

```
using System;

namespace GenericClassEx
{
  class Demo <T1, T2>
  {
    T1 i;
    T2 j;

    public Demo (T1 no1, T2 no2)
    {
      i = no1;
      j = no2;
      Console.WriteLine (i);
      Console.WriteLine (j);
    }
  }
```

```
class Program
{
  Static void Main()
  {
    Demo<int, float> obj = new Demo<int, float>
                                (10, 11.30f);
    Demo<int, string> obj1 = new Demo<int, string>
                                (23, "ABC");
    Console.Readkey();
  }
}
}
```

Delegate :-

- If we want to pass value indirectly to the function then used delegate.

* • Indirectly means =>

(When method is static)

```
calculator1 C = new calculator1(Program.Addition);
Console.WriteLine("Addition of 5 and 10 is:{0}", C(5,10));
```

(When method is non-static)

```
Program P5 = new Program();
Calculator3 d6 = new calculator3(P5.division);
```

• Directly means =>

```
Program obj = new Program();
obj.fun();
```

- It is function pointer which holds reference of a method and then calls method for execution.

* - What if we want to pass a function as a parameter ?
  => The ans is 'delegate'.

- There are three steps involved while working with delegates :
  (1) Declare a delegate
  (2) Set a target method
  (3) Invoke a delegate
- A delegate can be declared using the 'delegate' keyword
- A delegate can be declared outside of the class or inside the class
- Return type of delegates and function should be same.
- Passing input parameter type of delegates and function should be same
- Syntax of delegate declaration :

[access modifier] delegate [returntype] [delegateName] ([Parameter])

- Delegates are similar to C++ function pointers, but delegate are fully object-oriented & ~~unlike~~ unlike C++ pointers to member functions, delegates encapsulates both an object instance and a method.
- Delegates can be chained together, for ex, multiple methods can be called on a single event.

~ Delegate Declaration :-

```
public delegate void delName (string str_name);
```

- Program:-

```
namespace DelegateExample
{
public delegate void delName (string str_name);

class Program
{
  delegate float delInside (float a, float b);
  public void fun (string para)
  {
    Console.WriteLine (para);
  }
  public void fun1 (string para)
  {
    Console.WriteLine (para);
  }
  static float multiplication (float paraA, float paraB)
  {
    return paraA * paraB;
  }
  static void Main ()
  {
    Program objClass = new Program ();
    // object of delegate
    delName objDel = new delName (objClass.fun);
    objDel ("ABC");
    objDel.Invoke ("PQR");

    delName objDel1 = new delName (objClass.fun1);
```

```
delInside objDelInside = new delInside (Program.multiplication
Console.WriteLine ("Inside class" + objDelInside (20.00F,
                                                  10.13f));
Console.ReadKey ();
}
}
}
```

Exception Handling:-

- Program:-

```
namespace ExceptionHandellingExample
{
  class Program
  {
    static void Main (string [] args)
    {
      try
      {
        Console.WriteLine ("Please enter employee id");
        int id = Convert.ToInt32 (Console.ReadLine ());
        Console.WriteLine (Id);
      }
      catch (Exception ex)
      {
        Console.WriteLine (ex.Message);
      }
    } Console.Readkey();
  }
}
```

→ Program:-

```
namespace Exception Handling Example
{
  class Program
  {
    static void Main (string [] args)
    {
      try
      {
        Console.WriteLine ("Please Enter Employee Id");
        int id = Convert.ToInt32 (Console.ReadLine());
        Console.WriteLine (id);
      }
      catch (Exception ex)
      {
        Console.Writeline (ex.Message);
      }
      finally
      {
        Console.Writeline ("Please Enter Employee Id");
        int Id = Conver..
        ..
      }
      Console.Readkey ();
    }
  }
}
```

* Format Exception:-

The exception that is thrown when the format of argument is invalid, or when a composite string is not well formed.

```
catch ( FormatException fx)
{
  Console.WriteLine (fx.Message);
}
```

* Divide By Zero Exception:-

The exception that is thrown when there is an attempt to divide an integral or decimal value by zero.

* Invalid Cast Exception:-

The exception that is thrown for invalid casting or explicit conversion.

Ex

```
try
{
  object obj = 10.10;
  int a = (int) obj;
}
catch (InvalidCastException ex)
{
  Console.WriteLine (ex.Message);
}
```