C#

1) Features of C#
    a. Object oriented.
    b. Platform Independent – Windows, IOS, Android.
    c. Language Independent (Cross Language Reusability) – VC++.NET, VC#.NET & VB.NET – MSIL.
2) Constructors –
    a. It is a special method present under a class responsible for initializing the variable of that class.
    b. Constructors are non-value returning method.
    c. Each and every class requires constructor if you want to create the instance of the class.

```
class Test
{
        int I; string b; bool c;
}
```

Test a = new Test(); //valid

    d. It's responsibility of a programmer to define a constructor under his class and if he fails to do so, on behalf of the programmer an **implicit constructor** gets defined in the class by complier.

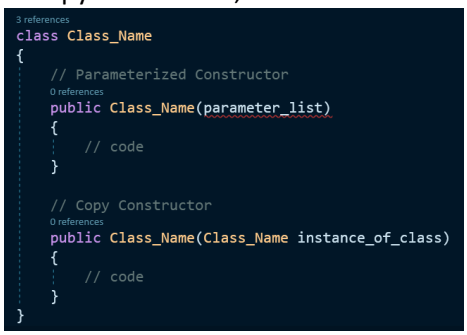
```
class Test
{
        int I; string b; bool c;
        Public Test() //Complier defined the constructor -
        {
                int I = 0; string b = "45"; bool c = false; //Initializing the variable
        }
}
Test a = new Test(); //valid
```

    e. Implicitly defined constructors are parameter less and these constructors are also known as default constructors.
    f. Implicitly defined constructors are public.
    g. Explicit constructors are defined by programmers. Explicit constructor can be parameter less or parameterized.

```
[<modifiers>] <Name>( [<Parameter List>] )
{
        //-statements.
}
```

Defining the constructor: Implicit or Explicit
Calling the constructor: Explicit

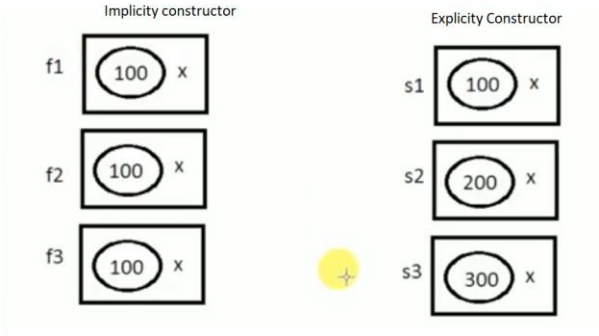
3) Types of Constructor
    a. Default or Parameter less constructor
        i. If a constructor method doesn't take any parameters then we call that as default or parameter less constructor.
        ii. These constructors are defined by programmers explicitly or else will be defined by complier if there is no explicit constructor under the class.
    b. Parameterized Constructor
        i. If a constructor method is defined with parameters we call that as parameterized constructor.
        ii. These constructors can only be defined by programmer explicitly.
    c. Copy Constructor
        i. If you want to create multiple instances with the same value then we use copy constructor.
        ii. In copy constructor, constructor takes the same class as parameter to it.



```
3 references
class Class_Name
{
    // Parameterized Constructor
    0 references
    public Class_Name(parameter_list)
    {
        // code
    }

    // Copy Constructor
    0 references
    public Class_Name(Class_Name instance_of_class)
    {
        // code
    }
}
```

    d. Static Constructor:
        i. If a constructor is explicitly declared by using static modifier we call that as static constructor.
        ii. All the constructor we have defined till now are non-static or instance constructor.
        [<Static Modifier>] [<Constructor Name>](){}
        iii. If a class contains any static variables then only implicit static constructor will be present or else we need to define explicitly.
        iv. Whereas non-static constructors will be implicitly defined in every class (except static class) provided we did not define them explicitly.
        v. Static constructor are responsible in initializing static variables.
        vi. Static constructors are never called explicitly they are implicitly called.
        vii. Static constructors are first to execute under any class.
        viii. Static constructors can't be parameterized so overloading static constructors is not possible.
        ix. **Defining the static constructor:** Implicit or Explicit
        x. **Calling the static constructor:** Implicit
4) Why constructors are needed?
    a. Every class requires a constructor to be present in it, if we want to create the instances of that class.
    b. Every class contains an implicit constructor if not defined explicitly and with the help of that implicit constructor the instance of the class can be created.

c. What is the need of defining a constructor explicitly again?
    i. Implicit constructor of a class will initialize variables of a class with same value even if we create multiple instances of that class.
    ii. If we define a constructor explicitly with parameters then we get a changes of initializing the fields or variables of the class with a new value every time we are creating instance of that class.



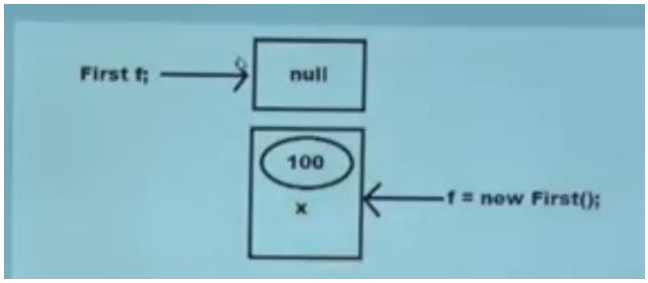
Note: Generally every class requires some value for execution and that values can be pass via constructors.

5) Static Constructor Vs Non-Static Constructors
    a. If a constructors is explicitly declared by using static modifier we call that constructors as static constructors whereas rest of others are non-static constructors only.
    b. Static fields/variables are initialized by static constructors.
    c. Non-static fields/variables are initialized by non-static constructors.
    d. Calling
        i. Static Constructors -> implicitly called.
        ii. Non-Static Constructors -> explicitly called.
    e. Execution
        i. Static Constructors ->
            1. Executes immediately once the execution of the class starts
            2. It is the first block of code to run under the class.
        ii. Non-Static Constructors ->
            1. It executes only after creating the instances of the class is created.
    f. Life Cycles
        i. Static Constructors
            1. Executes one and only one time
        ii. Non-Static Constructors
            1. Executes zero time if no instances is created.
            2. “N” times if “N” instances are created.
    g. Non-Static Constructors can parameterized but static constructors can’t have any parameters because static constructors are implicitly called.
    h. Non-Static constructors can be overloaded whereas static constructors can’t be overloaded.

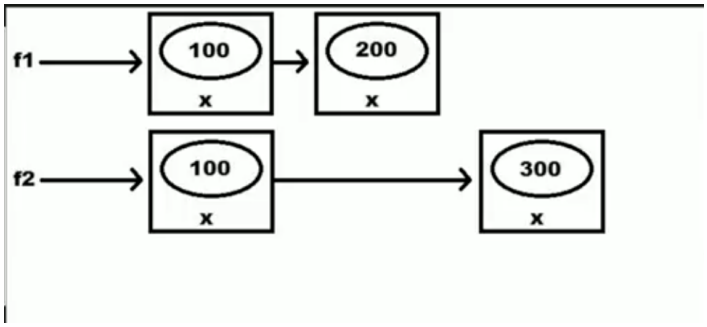
i. Every class contains an implicit constructor if not defined explicitly and those implicit constructors are defined based on the following criteria
    i. Every class expect static class contains an implicit non-static constructors if not defined with an explicit constructor.
    ii. Static constructors are implicitly defined only if that class contains any static fields/variables or else that constructors will not be present at all.

Class

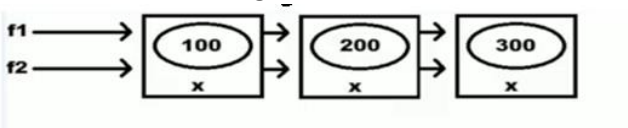
1) Class – It's a user-defined data type. It is blueprint or a plan. Class are collection of members.
2) Object – It is execution of the blueprint i.e. class.
3) Variable of class
    a. Copy of the class that is not initialized.



4) Instance of a class
    a. A copy of the class that is initialized by using new keyword which has its own memory and never shared with another instances.



5) Reference of a class
    a. A copy of the class that is initialized by using an existing instances and references of class will not have any memory allocation.
    b. They will be sharing the same memory of the instance that was assigned for initializing the variable.
    c. Reference of a class can be called as pointer to the instance.
    d. Every modification we perform on the members using instance reflects when we access those members through references and vise-verse.



Access Specifier

1) It's a special kind of modifiers using which we can define the scope of a type and its members.
2) Private: within the class.
    a. The private method is accessible only within the class in which it was defined.
    b. We cannot define Types as private i.e. user define types or types.
        i. Private class T {} – Not allowed.
    c. Default scope of all members under the class is private.
3) Internal: within the project, both child and non-child class.
4) Protected: within the same class and within the child class.
5) Protected Internal: Either within the project or within the child class of other/different project.
6) Public: Global Access.
7) The below table of understanding the scope of access specifier

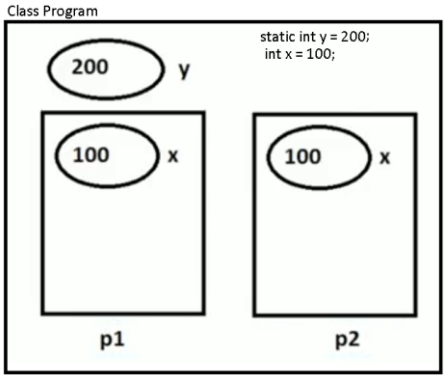
Cases	Private	Internal	Protected	Protected Internal	Public
Consuming Members of a class from **same** class	Yes	Yes	Yes	Yes	Yes
Consuming Members of a class from **Child** class of **same** project(**Inheritance**)	No	Yes	Yes	Yes	Yes
Consuming Members of a class from **non-Child** class of **same** project(Instance)	No	Yes	No	Yes	Yes
Consuming Members of a class from **Child** class of **another** project(**Inheritance**)	No	No	Yes	Yes	Yes
Consuming Members of a class from **non-Child** class of **another** project(Instance)	No	No	No	No	Yes

Note:

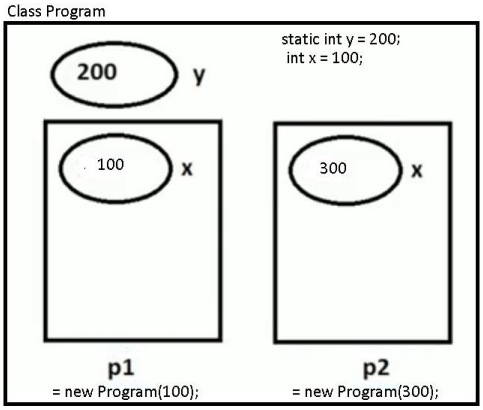
- Private, Protected, Protected internal cannot be used to define/declare a class.
- Default access specifier of class is internal, if not mentioned/specified.
- Default scope of all members under the class is private, if not mentioned/specified.
- **On Member** of the **class** we can apply **5 Scopes** but for **Types** we can apply only **2 Scopes (public & internal)**.

Variables

1) **Non-Static/Instance** Variable and **Static** Variables
    a. If a variables is explicit declared by using the static modifier or else if a variable is declared inside a static method then those variables are static and rest of the other are non-static.
    b. Static members of a class doesn't require the instance of class for initialization or execution also, where as non-static member of a class requires the instance of class both for initialization and execution.
    c. Static variables of a class are initialized immediately once the execution of a class starts whereas instance variables are initialized only after creating the class instance.
    d. In the life cycle of a class a static variable is initialized one and only once whereas instance variables are initialized for "0" times if no instance is created and "n" times if "n" instances are created.

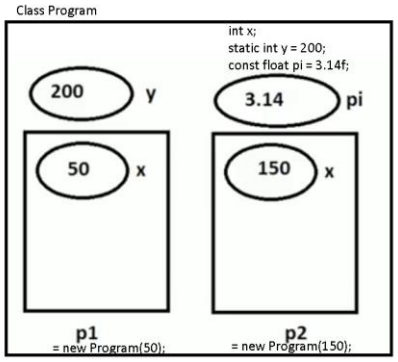


    e. Initialization of instance variable is associated with instance creation and constructor calling, so instance variables can be initialized through the constructor also.

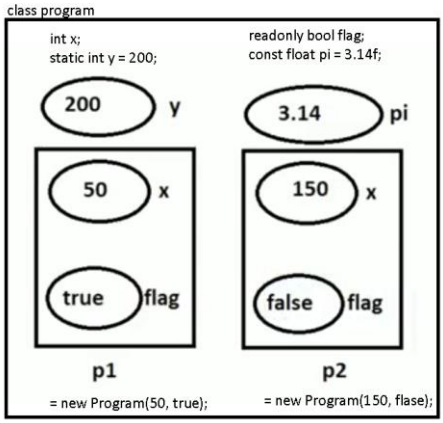


2) Constant Variable –
    a. If a variable is declared using **const** keyword we call it as constant variables.
    b. Constant variables can't be modified once after there are declared.
    c. Constant variables must be declared and initialized at the same time.
    d. The behavior of constant variable is similar as of static variable with only one difference
        i. Static variables can be modified but constant variables cannot be modified.

e. Constant variables are also initialized one and only one time in the life cycle of a class.
f. Constant variables doesn't require instance of class for accessing or initializing.



3) Read-only
   a. If a variable is declared using **readonly** keyword we call it as read-only variables.
   b. Read-only variables can't be modified like constant variables but after initialization.
   c. It is compulsory to initialize the read-only variables at the time of declaration, they can also be initialized under the constructor.
   d. The behavior of read-only variables are similar to instance variables
      i. Initialized only after creating the instance of a class.
      ii. Each instance of a class has a separate copy of read-only variable.
   e. The only difference between read-only variables and instance variable is
      i. Instance variable can be modified after instance of a class is created but read-only variables cannot be modified after instance of a class is created.



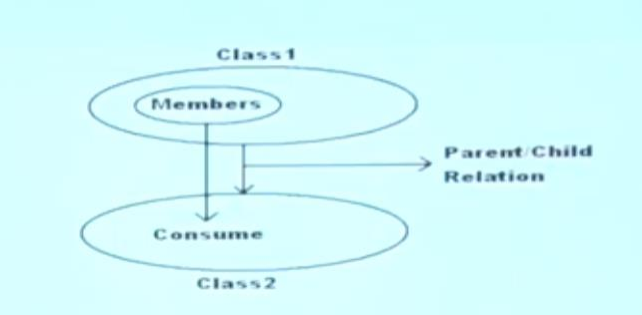
4) Constant Vs Readonly
   a. Constant variable is fixed value for the whole class.
   b. Readonly variables are fixed value for the specific instance of a class.

Note:

1) Single copy and modifiable - Static variables.
2) Multiple copy and modifiable – Non-Static/Instance variables.
3) Single copy and non-modifiable – Constant variables.
4) Multiple copy and non-modifiable – Read-only variables.

Inheritance – Re-usability

1) It's a mechanism of consuming the members of one class in another class by establishing **parent/child** relationship between the classes which provides re-usability.



2) How to inherit a class:

```
[<modifiers>] class <child class> : <parent class>

class A
{
 -Members
}
class B : A
{
 -Consuming the members of A from here
}
```

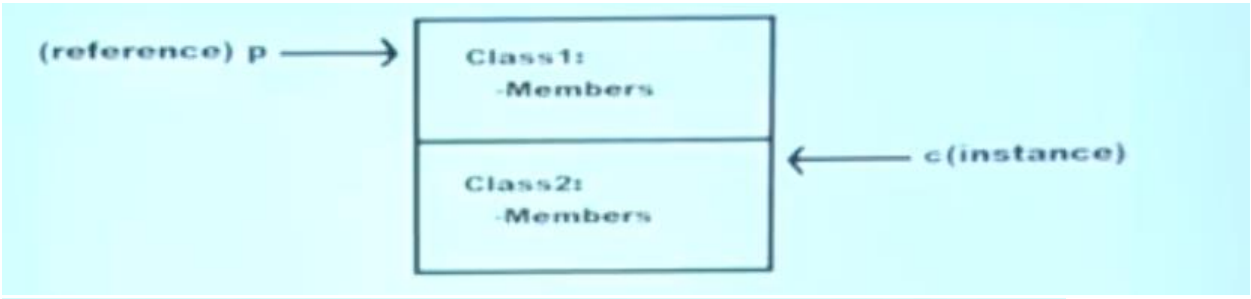
3) Terminology
    a. Class A – Base or Parent or Super class
    b. Class B – Derived or Child or Sub class
4) In inheritance, child class can consume member of the parent class as if it is the owner of those members expect **private members of parent class**.

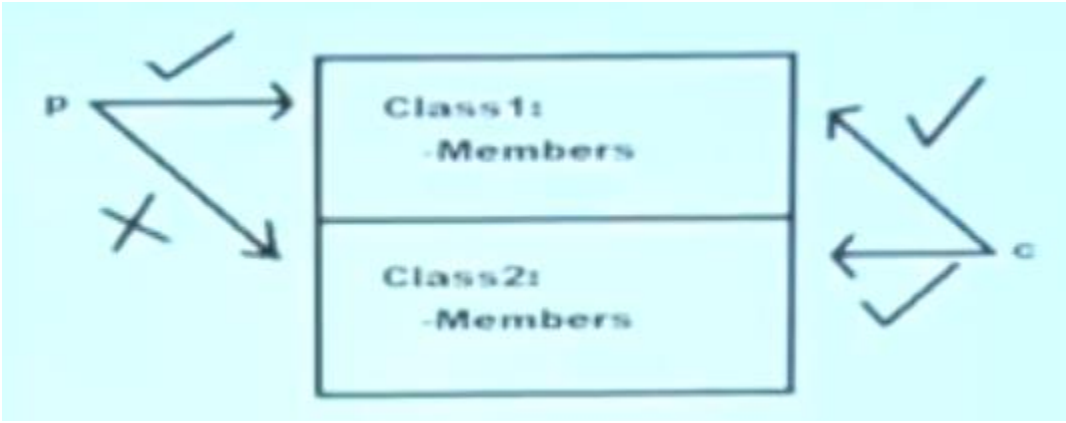
Important rules to remember while working with inheritance

1) Parent class constructor should be accessible to child class, otherwise inheritance will not be possible.
    a. If you call the parent class constructor than only the parent class variables/members will be initialized and they can be consumed under the child class.
    b. Child class constructor implicitly calls the Parent class constructor.
2) In inheritance, child class can access parent class's members but parent classes can never access any member that is purely defined under the child class.

3) We can initialize a parent class's variable by using the child class instance to make it as reference, So that the reference will be consuming the memory of child class instance, but in this case also we can't call any pure child class members by using the reference

```
static void main()
{
        Class1 p; //p is variable of Class1
        Class2 c = new Class2(); //c is instance of Class2
        p = c; //p is a refernce of parent class created by using child class instance
        p.Test1();
        p.Test2();
        p.Test3(); //Not Allowed, Test3 is method of Class3
        Console.ReadLine();
}
```



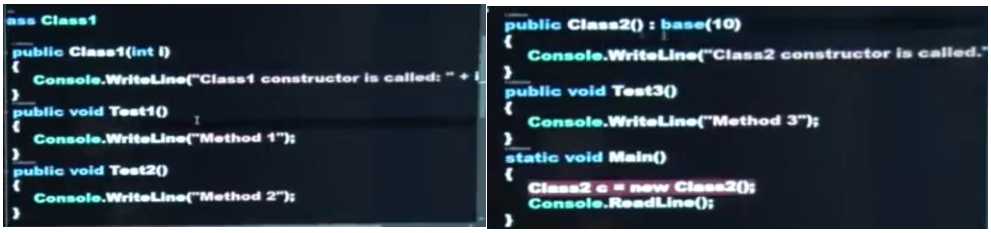


4) Every class that is defined by us or pre-defined in the libraries of the language has a **default parent class** i.e. **Object** class of **system** namespace.
    a. Members of the object class are accessible from anywhere.
    b. Members of the object class are **Equals(), GetHashCode(), GetType() and ToString().**
    c. **Object** class is the ultimate base class.

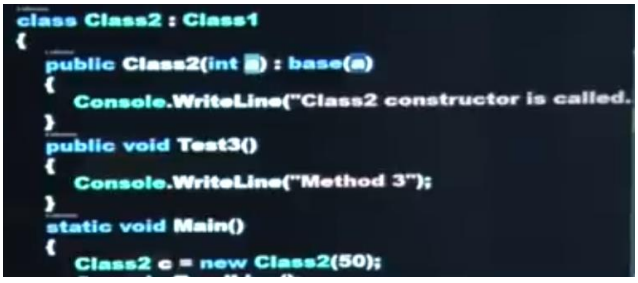
```
//Object is the ultimate root class
//Implicity inherits the object class
//since it was not inherit by any class
class Class1 : Object
{
}

class Class2 : Class1
{
}

class Class3 : Class2
{
}

void Main()
{
        Class3 c = new Class3();
        //We are able to access the object class members
        //Since Class3 has inherited by Class2
        //Class2 has inherited by Class1
        //Class1 has inherited by Object Class
        //Object is Grand Parent of Class3
        c.Equals(); c.ToString();
}
```

5) In CSharp, we don't have support for multiple inheritance through classes, what we are provided is only single inheritance through classes.
6) If parent class constructor is parameterized and default constructor is not defined than child class cannot call the parent class constructor implicitly. So to overcome the problem, programmer should explicitly call parent class constructor from child class and pass values to those parameters.
    a. To call parent class constructor from child class, **base** keyword is used.

Eg. 1)

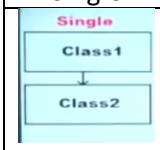
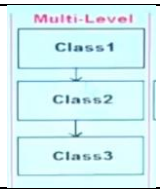
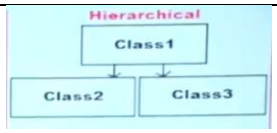
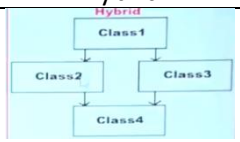
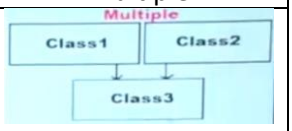


E.g. 2)

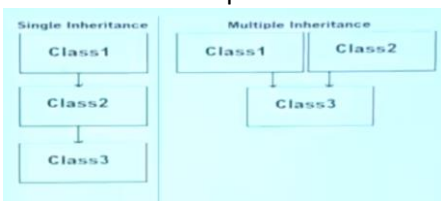


Types of Inheritance

1) No. of parent classes a child have or the no. of child classes a parent class have

Single	Multi-Level	Hierarchical	Hybrid	Multiple
				

2) If at all a class has 1 immediate parent class to it we call it as **Single** inheritance and if it has more than one immediate parent class to it we call it **multiple** inheritance.



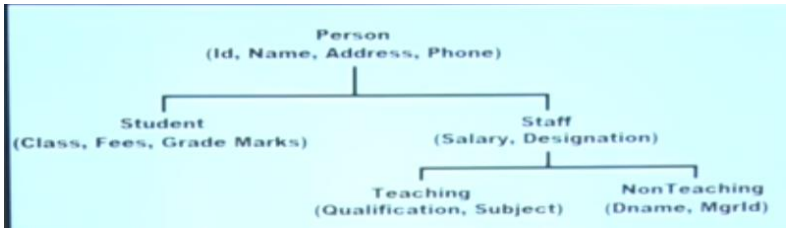
How to use inheritance

Entity – it is living or non-living object associated with set of attributes.

1) Identify the entities that are associated with the application we are developing.
    a. E.g. School Application: Student, Teaching Staff & Non- Teaching Staff.
2) Identify the attributes of each and every entities

Student	Teaching Staff	Non-Teaching Staff
Id	Id	Id
Name	Name	Name
Address	Address	Address
Phone	Phone	Phone
Class	Designation	Designation
Marks	Salary	Salary
Grade	Qualification	Dname
Fees	Subject	MgrId

3) Identify the common attributes of each entity and put them in a hierarchical order



4) Define the class representing the entities that are put in hierarchical order

```
public class Person
{
        public int Id;  public string Name,Address;  public long Phone;
}

public class Student : Person
{
        int Class;      float Marks;    float fees;
}

public class Staff
{
        public double salary; public string Designation;
}

public class Teaching : Staff
{
        string qualifaction, Subject;
}

public class NonTeaching : Staff
{
        string Dname;
        int MgrId;
}
```

Polymorphism - Method Overloading

1) Method overloading is approach of defining method with multiple behaviors where behavior of the method will be changing based on the parameters of that method.
   i.e. input changes -output changes

```
1 reference
public void Test() { Console.WriteLine("Test"); }
1 reference
public void Test(int i) { Console.WriteLine("Test with int"); }
1 reference
public void Test(string s) { Console.WriteLine("Test with string"); }
1 reference
public void Test(string s, int i) { Console.WriteLine("Test with string, int"); }
1 reference
public void Test(int i, string s) { Console.WriteLine("Test with int, string"); }
//Below is not allowed - return type
//public string Test() { Console.WriteLine("This function is invalid. Ambiguity issue."); }
0 references
static void Main(string[] args)
{
    Console.WriteLine("Hello World!");
    Program p = new Program();
    p.Test();
    p.Test(10); p.Test("String");
    p.Test("string", 10); p.Test(10, "String");
    Console.ReadLine();
}
```

2) Why **return** types are not considered? – Compile time error – Ambiguity issue

```
1 reference
public void Test() { Console.WriteLine("Test"); }
1 reference
public string Test() { Console.WriteLine("Test with diff return type"); }
```

   a. It causes ambiguity between the methods.
   b. Complier won't be able to figure out which method to call, return type comes into picture at the end of execution but Calling of the method comes into picture at the start and if complier doesn't knw where to start it will give you error "Ambiguity method"

3) Rules of Method overloading
   a. Number of parameters(inputs)
   b. Type of parameters(inputs)
   c. Order of parameters(inputs)
   d. Return Types are not taken into consideration.

Polymorphism - Method Overriding

1) Method overriding is an approach of re-implementing a parent class method under the child class with same signature.

```
1 reference
class ParentLoad
{
    1 reference
    public void Show() { Console.WriteLine("Parent Show Method"); }
    //Overridable
    2 references
    public virtual void Test() { Console.WriteLine("Parent Test Method"); }
}

2 references
class ChildLoad : ParentLoad
{
    //OverLoading
    1 reference
    public void Show(int i) { Console.WriteLine("Child Show Method"); }
    //Overriding
    2 references
    public override void Test() { Console.WriteLine("Child Test Method"); }
    0 references
    static void Main(string[] args)
    {
        Console.WriteLine("Hello World!");
        ChildLoad childLoad = new ChildLoad();
        childLoad.Show();
        childLoad.Show(10);

        childLoad.Test(); //Child method will be called since it is overridden in child class

        Console.ReadLine();
    }
}
```

2) Difference

Method Overloading	Method Overriding
1) In this case, we define multiple methods with same name by changing their parameters. 2) This can be performed either within a class as well as between parent child classes also. 3) While overloading a parent class method under the child class, child class doesn't require to take any permission from the parent class. 4) Overloading is all about **defining multiple behaviors** to a method.	1) In this case, we define multiple methods with same name and same parameters. 2) This can be performed only between parent child classes it can never be performed within the same class. 3) While overriding a parent class method under child class required a permission from the parent class. i.e. **virtual** keyword is required in the parent class method public virtual void Test() {} 4) Overriding is all about **changing the behavior** of **parent class** under **child class.**

3) If we want to override a parent method under child class first that method should be declared by using "**virtual**" modifier in parent class.
4) Any **virtual** method of the parent class can be overridden by the child class if required by using "**override**" modifier in child class.

```
Class1
{
        public virtual void Test() { }          //Overridable
}

Class2 : Class1
{
        public override void Test() { }         //Overridden
}
```

Polymorphism - Method Hiding/Shadowing

1) Differ & Commons

Method overriding	Method hiding\shadowing
1) Approach of re-implementing a parent classes method under the child class exactly with the same name and signature. 2) Child class **re-implements** its parent class methods which are **declared** as **virtual** 3) With permission – virtual –override keywords 4) In overriding, a parent class reference can call child classes overridden members.	1) Approach of re-implementing a parent classes method under the child class exactly with the same name and signature. 2) Child class **re-implements** any parent class method even if it is **not declared** as **virtual**. 3) Without permission – new keywords. 4) In hiding, a parent class reference cannot call child class's members which are re-implemented by hiding approach i.e. **new** keyword.

2) We re-implement a parent classes method under the child class using 2 approaches
    a. Method overriding - **With permission – virtual – override keywords**
    b. Method hiding/shadowing – **Without permission – new keyword**
3) After re-implementing parent classes methods under child classes
    a. The child class instance will start calling the local methods (re-implemented methods) only.
    b. If required in any case, we can also call parent class method from child classes by 2 approaches
        i. By **Creating instance** of the parent class under the child class
        ii. By using "**base**" keyword, we can calls parent class method.
            1. "**base**" & "**this**" keyword cannot be called inside **static** block.
4) A parent class reference even if created by using child class instance can't access any members that are **purely defined** under the child class but can call **overridden** members of the child class.
    a. **Overridden** members are **not** considered as **pure child class** members.
5) Members which are **re-implemented** by using **method hiding** are considered as **pure child class** members hence are **not accessible** to **parent's references**.

```
class ParentClass
{
    5 references
    public virtual void Test1() { Console.WriteLine("Parent Test1"); }
    3 references
    public void Test2() { Console.WriteLine("Parent Test2"); }
    0 references
    public void Test3() { Console.WriteLine("Parent Test3"); }
}

4 references
class ChildClass : ParentClass
{
    5 references
    public override void Test1() { Console.WriteLine("Child Test1"); } //Overridden
    1 reference
    public new void Test2() { Console.WriteLine("Child Test2"); } // Method hidding
    //it is execute with warning - use new keyword if hiding is intended.
    0 references
    public void Test3() { Console.WriteLine("Parent Test3"); }
    1 reference
    public void ParentTest1() { base.Test1(); } // Calling Base/Parent Class method
    1 reference
    public void ParentTest2() { base.Test2(); } // Calling Base/Parent Class method
    0 references
    static void Main(string[] args)
    {
        ParentClass parentClass = new ParentClass();
        parentClass.Test1(); parentClass.Test2(); // Calling Parent Class method using instance of Parent Class
        ChildClass childClass = new ChildClass();
        //Calling Parent class method using Child class instance but by base keyword in the method.
        childClass.ParentTest1(); childClass.ParentTest2();
        childClass.Test1(); childClass.Test2(); // Calling child class methods which are overridden & shadowed.
        Console.WriteLine("--------------------------------------------------------------------------------");
        ChildClass childClass1 = new ChildClass(); //childClass1 is the instance ChildClass
        //parentClass1 is reference of parent class by using instance of child class
        ParentClass parentClass1 = childClass1;
        parentClass1.Test1(); // Overridden method of child class is called i.e. "Child Test1"
        parentClass1.Test2(); // Parent class method is called.

        Console.ReadLine();
    }
```

```
D:\DefaultPathForProjects\VS2019\CSharp\W
Parent Test1
Parent Test2
Parent Test1
Parent Test2
Child Test1
Child Test2
--------------------------------------------------------------------------------
Child Test1
Parent Test2
```

Polymorphism – Operator Overloading

1) **Method Overloading** is an approach of **defining multiple behaviors** to a **method** and those **behaviors** will **vary** based on the **input parameters** to that **method**.

```
String str = "Hello how are you";
str.Substring(14);      you
str.Substring(10);      are you
str.Substring(10, 3);   are
```

2) **Operator Overloading** is an approach of **defining multiple behaviors** to an **operator** and those **behaviors** will **vary** based on the **operand types between** which the **operator** is **used**. **E.g.**
   - \+ is an **addition** operator when used **between** 2 **numeric operands**
   - \+ is an **concatenation** operator when used **between** 2 **string operands**

   a. Number + Number => Addition
   b. String + String => concatenation
3) In the libraries of the language, operator +, - , * , / functionality is pre-defined as operator method.
   a. public static int operator +(int a, int b) → int z = 10 + 20;
   b. public static int operator –(int a, int b) → int z = 10 -20;
   c. public static string operator +(string a, string b)

      string s1 = “Hello”; string s2 = “world”;
      string s3 = s1 + s2;

d. public static bool operator >(int a, int b)
    int a = 10; int b = 20;
    bool b = a > b;
e. public static bool operator ==(string a, string b)
    string s1 = “Hello”; string s2 = “world”;
    bool b = s1 == s2;
f. public static bool operator !=(string a, string b)
        string s1 = “Hello”; string s2 = “world”;
        bool b = s1 != s2;
g. public static bool operator -(string a, string b) → **No Predefined implementation is libraries.**

    string s1 = “Hello”; string s2 = “world”;
    string s3 = s1 - s2; //**Error** – Operator ‘-’ cannot be applied to operand type ‘string’ & ‘string’.

4) How to define a operator

```
[<modifiers>] static <return type> operator <operatorsign>(<operand types list>)
{
    --logic
}
public static Matrix operator +( Matrix obj1, Matrix obj2)
{
    return new Matrix();
}
```

5) E.g. Code Example for Operator overloading & Method Overriding

```
15 references
public class Matrix
{
    int a, b, c, d;
    4 references
    public Matrix(int a, int b, int c, int d)
    {
        this.a = a; this.b = b; this.c = c; this.d = d;
    }
    //Operator Overloading for + for Matrix objects
    1 reference
    public static Matrix operator +(Matrix obj1, Matrix obj2)
    {
        return new Matrix(obj1.a + obj2.a, obj1.b + obj2.b, obj1.c + obj2.c, obj1.d + obj2.d);
    }
    //Operator Overloading for - for Matrix objects
    1 reference
    public static Matrix operator -(Matrix obj1, Matrix obj2)
    {
        return new Matrix(obj1.a - obj2.a, obj1.b - obj2.b, obj1.c - obj2.c, obj1.d - obj2.d);
    }
    //Method overriding of ToString method
    0 references
    public override string ToString()
    {
        return $"{a}\t{b}\n{c}\t{d}\n";
    }
}
0 references
public class TextMatrix
{
    0 references
    static void Main()
    {
        Matrix m1 = new Matrix(20,10,5,0);
        Matrix m2 = new Matrix(10,20,30,40);
        Matrix m3 = m1 + m2;
        Matrix m4 = m1 - m2;

        Console.WriteLine(m1); Console.WriteLine(m2); Console.WriteLine(m3); Console.WriteLine(m4);
        Console.ReadLine();
    }
```

```
D:\DefaultPat
20      10
5       0

10      20
30      40

30      30
35      40

10      -10
-25     -40
```

Abstract Classes & Methods

1) A method without any method body is known as an **Abstract method**, what the method contains is only declaration of the method.

```
public abstract void Add(int a, int b);
```

2) A class which **contains** any **abstract members (methods)** in it is known as Abstract class.

```
abstract class Math
{
        public abstract void Add(int x, int y);
}
```

3) To define a method or a class as abstract we require to use the "**abstract**" keyword on them.
4) If a method is declared as abstract under any class then child class of that class is responsible for **implementing the method without fail.**
5) The concept of abstract method will be nearly similar to the concept of Method Overriding.
6) Difference between method overriding and abstract methods

Method overriding	Abstract Method

public class Parent
{
public abstract void show()
{
//Implementation
}
}
public class ChildClass : Parent
{
//**Optional**
public override void show()
{
//Re-Implemenentation
}
}
	public abstract class AbstractParent
	{
	public abstract void show();
	}
	public class ChildClass : AbstractParent
	{
	//**Mandatory**
	public override void show()
	{
	//Implemenentation
	}
	}

7) Abstract Class can contain both abstract & non-abstract methods.
    a. Abstract Methods - **Liability**
    b. Non- abstract methods - **Assets**
8) Child class of Abstract Class should
    a. Implement each and every abstract method of parent class by using "**override**" keyword. - **Liability**
    b. Then only we can consume non-abstract method of parent class. – **Assets**
9) We cannot create instance of Abstract class but we can create reference of abstract class using the child class instance.
10) Parent class reference can call child classes overridden methods but can never call purely defined methods defined under child class.

```
3 references
abstract class AbstractParent
{
    2 references
    public void add(int x, int y) { Console.WriteLine(x + y); }
    2 references
    public void subtract(int x, int y) { Console.WriteLine(x - y); }
    3 references
    public abstract void multiple(int x, int y);
    3 references
    public abstract void divide(int x, int y);
    1 reference
    public static void show(string msg) { Console.WriteLine(msg); }
}
2 references
class Child : AbstractParent
{
    3 references
    public override void multiple(int x, int y) { Console.WriteLine(x * y); }
    3 references
    public override void divide(int x, int y) { Console.WriteLine(x / y); }
    1 reference
    public void ParentClassWontBeAbleToCallMe() { Console.WriteLine("Pure Child Method"); }
    0 references
    static void Main(string[] args)
    {
        AbstractParent.show("Test"); //Static Members can be called but instance of abstract class can't be created.
        Child child = new Child(); //Instance of Child class
        child.add(1, 2); child.subtract(1, 2); child.multiple(1, 2); child.divide(2, 2); child.ParentClassWontBeAbleToCallMe();
        Console.WriteLine("------------------------------------------------------------");
        //abstractParent is reference of Abstract class using the child class instance
        AbstractParent abstractParent = child;
        //Parent class reference can call child classes overridden methods but not purely defined methods
        abstractParent.add(1, 2); abstractParent.subtract(1, 2); abstractParent.multiple(1, 2); abstractParent.divide(2, 2);
        //abstractParent.ParentClassWontBeAbleToCallMe(); //Error
        Console.ReadLine();
    }
}
```

```
D:\DefaultPathForProj
Test
3
-1
2
1
Pure Child Method
------------------
3
-1
2
1
```

How to implement Abstract Class & Method

1) Advantage of Abstract method
    a. When logic varies from class to class then it is a good idea to implement an abstract method in parent class and child class will be forced to implement the abstract method
    b. We will make the child class to implement the method with particular signature & name but implementation logic can be different.
2) Identify the entities.

```
Entities: Rectangle, Circle, Triangle, Cone
```

```
Rectangle: Width, Height
Circle: Radius, Pi
Triangle: Width, Height
Cone: Radius, Height, Pi
```

3) Identify the common attributes of these entities.

```
Width,Height,Radius,PI
```

4) Identify the common functionality of these entities and declare it as abstract

```
public abstract double GetArea();
```

5) Now Define the common attribute and functionality in abstract class

```
public abstract class Figure
{
        Width,Height,Radius,PI;
        public abstract double GetArea();
}
```

6) Actual Implementation

a.

```
3 references
public abstract class Figure
{
    public double Width, Height, Radius;
    public const float PI = 3.14f;
    6 references
    public abstract double GetArea();
}
3 references
public class Rectangle : Figure
{
    1 reference
    public Rectangle(double width, double height){ this.Width = width; this.Height = height;  }
    2 references
    public override double GetArea() { return Width * Height; }
}
3 references
public class Circle : Figure
{
    1 reference
    public Circle(double radius) { this.Radius = radius; }
    2 references
    public override double GetArea() { return PI * Radius * Radius; }
}
3 references
public class Cone : Figure
{
    1 reference
    public Cone(double radius, double height) { this.Radius = radius; this.Height = height; }
    2 references
    public override double GetArea()
    {
        return PI * Radius * (Radius + Math.Sqrt(Height * Height + Radius * Radius));
    }
}
```

b.

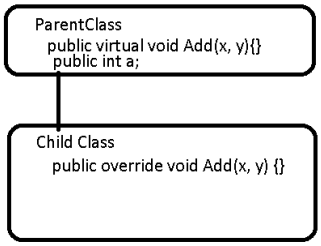
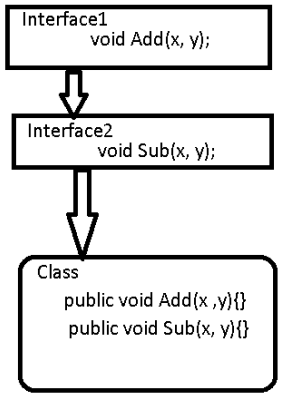
```
0 references
public class TestFigure
{
    0 references
    static void Main(string[] args)
    {
        Rectangle rectangle = new Rectangle(10.0,20.0);
        Circle circle = new Circle(15.0);
        Cone cone = new Cone(15.0, 5.0);

        Console.WriteLine($"Area of Rectangle :> {rectangle.GetArea()}");
        Console.WriteLine($"Area of Circle :> {circle.GetArea()}");
        Console.WriteLine($"Area of Cone :> {cone.GetArea()}");

        Console.ReadLine();
    }
}
```

```
D:\DefaultPathForProjects\VS2019\CSharp\AbstractCla
Area of Rectangle :> 200
Area of Circle :> 706.5000236034393
Area of Cone :> 1451.2164374533024
```

Interface

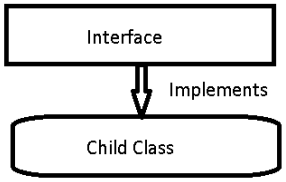
1) Difference & Common

Class	Interface

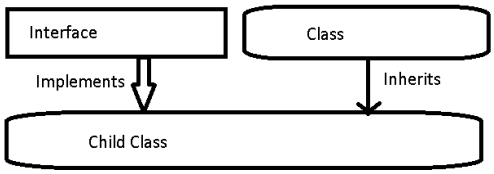
1) Class is a user-defined data type.
`[<modifier>] class[<ClassName>]`
`{`
2) `}`
3) Class : Non-Abstract methods (Methods with method body)
a. Abstract Class :
• Non-Abstract methods (Methods with method body)
• Abstract Methods (Methods without method body)
4) Generally a class inherits from another class to **consume or re-implement** the members of its parent class.
5) Default scope of the **class** is **private.**
6) We can declare any fields/variables under a class.
7) **Re-Implementation – Optional**
8) 
	1) Interface is a user-defined data type.
	`[<modifier>] interface [<IInterfaceName>]`
	`{`
	`-Abstract member declarations`
	2) `}`
	3) Interface : contains only **abstract methods (**Methods without method**)**
	4) If a class is inherited from an interface then it has to **implement** the members of its parent class.
	5) Default scope of the **interface** is **public.**
	6) We can't declare any fields/variables under an interface.
	7) **Implementation – Mandatory**
	8) 

2) Every abstract method of an interface should be implemented by the child class of the interface without fail (Mandatory).

3) A class can implement/inherit from an interface.



4) A class can inherit from a class and interface at a time.



5) By Default every member of interface is abstract so we don't require to use **abstract** modifier on it again just like we do in case of abstract class.

```
public abstract class AbsParent
{
        public abstract void Add(int z, int a);
}

public interface IInterface1
{
        void Add(int z, int a);
}
```

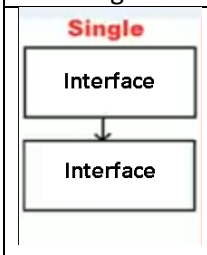
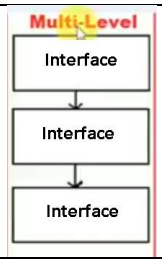
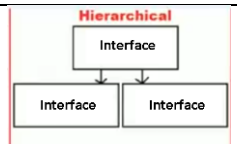
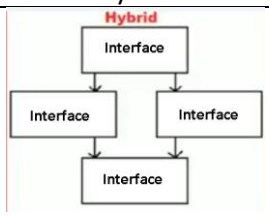
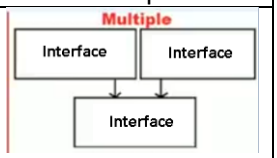
6) If required an interface can inherit from another interface.
7) Every member of an interface **should** be **implemented** under the child class of the interface **without fail**, but while implementing we **don't require** to use "**override**" modifier just like we have done in case of abstract class.
8) We **cannot create** an **object** of **interface** but we can **create reference** of **interface** using the **child class instance**.
9) E.g.

```
1 reference
public abstract class AbstractParent
{
    2 references
    public abstract void showMsg(string msg);
    2 references
    public virtual void TestMethod2() { Console.WriteLine("Test  Method2"); }
}
1 reference
public interface ICalculator1{  void Add(int x, int y); }

2 references
public interface ICalculator2 : ICalculator1  { void Sub(int x, int y);  }

2 references
class ChildClass : AbstractParent, ICalculator2
{
    3 references
    public void Add(int x, int y) { Console.WriteLine( x + y ); } //Interface - Abstract Method implementation Mandatory
    3 references
    public void Sub(int x, int y) { Console.WriteLine(x - y); }   //Interface - Abstract Method implementation Mandatory
    2 references
    public override void showMsg(string msg) { Console.WriteLine(msg); } //Abstract class- Abstract Method implementation Mandatory
    2 references
    public override void TestMethod2() { Console.WriteLine("Child TestMethod2"); } //Abstract class - Method re-implementation optional

    0 references
    static void Main(string[] args)
    {
        ChildClass childClass = new ChildClass(); //Object/instance of Child class
        childClass.Add(100, 50); childClass.Sub(100, 75);
        childClass.TestMethod2(); childClass.showMsg("Abstract method implementation");
        Console.WriteLine("-------------------------------------");
        ICalculator2 calculator2 = childClass; //Reference of interface created using child class instance
        calculator2.Add(200, 50); childClass.Sub(200, 75);
        //Error - because the below methods are not the members of interface. access denied.
        //calculator2.TestMethod2(); calculator2.showMsg("Abstract method implementation"); //Error

        Console.ReadLine();
    }
}
```

```
D:\DefaultPathForProjects\VS2019\CSharp
150
25
Child TestMethod2
Abstract method implementation
-------------------------------------
250
125
```

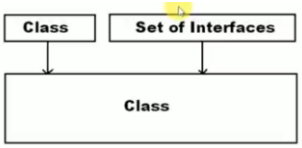
Multiple Inheritance using interfaces

1) Multiple inheritance using interface

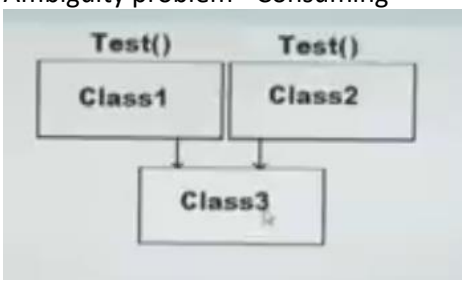
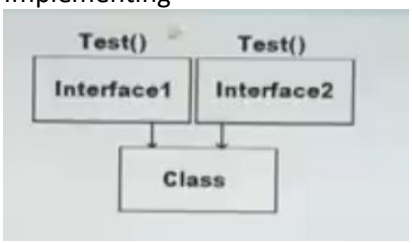
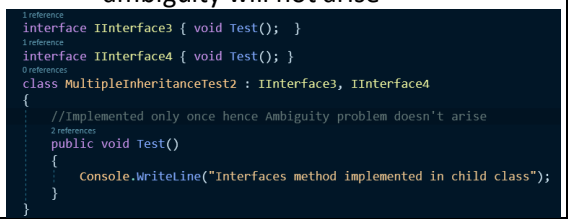
Single	Multi-Level	Hierarchical	Hybrid	Multiple
				

2) Even if multiple inheritance is not supported through classes in CSharp, it is still supported thru Interfaces

3) A class can have one and only one immediate parent class, whereas the same class can have any number of interfaces as its parent i.e. **multiple inheritance is supported in CSharp thru interface.**



4) Why multiple inheritance is not supported thru class and how is it supported thru interface?

Class	Interface
1) Ambiguity problem - Consuming  2) Since child class needs to consume or re-implement the parent class methods, ambiguity issue arises.	1) No Ambiguity problem - Implementing  2) Since child class need to implement the interface abstract method, ambiguity will not arise 

```
1 reference
interface IInterface3 { void Test();  }
1 reference
interface IInterface4 { void Test(); }
0 references
class MultipleInheritanceTest2 : IInterface3, IInterface4
{
    //Implemented only once hence Ambiguity problem doesn't arise
    2 references
    public void Test()
    {
        Console.WriteLine("Interfaces method implemented in child class");
    }
}
```

5) We never come across ambiguity problem when we work with multiple inheritance with interface,
6) Two way to implement interfaces
   a. Implicitly implementation
      We can call the methods using child class instances.
   b. Explicitly implementation
      We can call the methods using reference of interface
7) E.g.

```
3 references
public interface IInterface1
{
    3 references
    void Test(); void Show();
}
3 references
public interface IInterface2
{
    3 references
    void Test();  void Show();
}
2 references
class MultipleInheritanceTest : IInterface1, IInterface2
{
    //Explicity Implementation of abstract method of interface
    2 references
    void IInterface1.Show() { Console.WriteLine("Declared in interface1 and implemented in class"); }
    2 references
    void IInterface2.Show() { Console.WriteLine("Declared in interface2 and implemented in class"); }
    //Implicity Implementation of abstract method of interface
    //Both IInterface1, IInterface2 will assume the Test method belongs to them.
    5 references
    public void Test() { Console.WriteLine("Interfaces method implemented in child class"); }
    0 references
    public static void Main(string[] args)
    {
        MultipleInheritanceTest multipleInheritanceTest = new MultipleInheritanceTest();
        multipleInheritanceTest.Test();
        //Error - Since show method is implemented using explicity implementation
        // it can only be called via reference of IInterface1 or IInterface2
        //multipleInheritanceTest.Show();
        Console.WriteLine("-----------------------------------");
        IInterface1 interface1 = multipleInheritanceTest; //interface1 is reference of IInterface1
        interface1.Test();       interface1.Show();
        Console.WriteLine("-----------------------------------");
        IInterface2 interface2 = multipleInheritanceTest; //interface2 is reference of IInterface2
        interface2.Test();       interface2.Show();
        Console.ReadLine();
    }
}
```

```
D:\DefaultPathForProjects\VS2019\CSharp\InterfaceExamples\In
Interfaces method implemented in child class
-----------------------------------
Interfaces method implemented in child class
Declared in interface1 and implemented in class
-----------------------------------
Interfaces method implemented in child class
Declared in interface2 and implemented in class
```

Struct

1) Class is a user-defined type.
2) Structure is also a user-defined type.
3) Difference between Class & Struct

<table>
<tr><th>Class</th><th>Struct</th></tr>
<tr><td>
1) Class is reference type.
2) Memory allocation for instance of class is performed on managed heap memory.
3) Classes are used when we need to represent entities with large volume of data.
4) In-case of class, “new” keyword is mandatory for creating the instance of the class.
5) Fields of a class can be initialized at the time of declaration.
<pre>
class Test
{
        int i = 10; //Allowed
}
</pre>
6) We can define any constructor under the class that is either parameter less or parameterized and if no constructor is defined then there will be an implicit constructor which is default constructor.
<pre>
class Test1
{
        int i;
        public Test1(){} public Test1(int i){ this.i = i; }
        public void Display(){Console.WriteLine("Test1:>Display :> "+i);}
        static void Main()
        {
                Test1 test1 = new Test1();
                test1.Display(); //Output : Test1:>Display :> 0

                Test1 test11 = new Test1(10);
                test11.Display();//Output : Test1:>Display :> 10
        }
}
</pre>
7) If “Zero” constructors are defined in a class after compilation there will be 1 constructor (Implicit) if we define “n” constructors in a class after compilation there will be “n” constructors only.
8) Class can be inherited by other classes i.e. class supports inheritance.
9) Class can implement interfaces
</td><td>
1) Struct is value type.
2) Memory allocation for instance of Struct is performed on stack memory.
3) Structs are used when we need to represent entities with smaller volume of data.
4) In-case of class, “new” keyword is optional for creating the instance of the structure.
5) Fields of a Structs cannot be initialized at the time of declaration.
<pre>
struct Test1
{
        int i; //Allowed
        int j = 10; //Not allowed.
}
</pre>
6) Parameter less constructor i.e. default constructor is always implicit and can’t be defined explicitly again, what we can define is only parameterized constructor.
<pre>
struct Test1
{
        int i;
        public Test1(int i){ this.i = i; }
        public void Display(){Console.WriteLine("Test1:>Display :> "+i);}
        static void Main()
        {
                Test1 test1 = new Test1();
                test1.Display(); //Output : Test1:>Display :> 0

                Test1 test11 = new Test1(10);
                test11.Display();//Output : Test1:>Display :> 10
        }
}
</pre>
7) If “Zero” constructors are defined in a Struct after compilation there will be 1 constructor (Implicit) if we define “n” constructors in a class after compilation there will be “n + 1” constructors only.
8) Structure can’t be inherited by other structures i.e. structure doesn’t support inheritance.
9) Structures can implement interfaces
</td></tr>
</table>

4) Generally Class & Structure are used to represent entities.
5) **Structure** in **C Language** can contain only **Fields** in it whereas **Structure** in **CSharp** can contain most of the members what a class can contain like **Fields, Methods, Parameterized Constructors, Properties, Indexers and Operator Methods etc**.

```
3 references
public struct MyStruct
{
    public int iFields;
    1 reference
    public int testProperty { get; set; }
    0 references
    public enum testEnum { Red = 1, Yellow = 2 }
    0 references
    public void TestMethods() { }
    private string[] testIndexer;

    0 references
    public string this[int i]
    {
        get { return testIndexer[i]; }
        set { testIndexer[i] = value;  }
    }
    0 references
    public MyStruct(int iFields, int testProperty, string[] testIndexer)
    {
        this.iFields = iFields;
        this.testProperty = testProperty;
        this.testIndexer = testIndexer;
    }
}
```

6) All **pre-defined data types** under the libraries of our language which comes under **reference** type category i.e. **String & Object** are pre-define **class** whereas all **pre-defined data types** under the libraries of our language which comes under **value type** category i.e. **Int (Int32), float (Single), bool (Boolean), GUID** are **structure**.
7) If structure contains any fields then we need to initialize those fields either by
    a. Explicitly calling default constructor with the help of “new” keyword.

```
struct Test1
{
        int i;
        public void Display(){Console.WriteLine("Test1:>Display");}
        static void Main()
        {
                Test1 test1 = new Test1(); test1.Display();
        }
}
```

    b. Else if we are not using “new” keyword for creating the instance we need to explicitly assign value to fields referring it thru the instance and assign value.

```
struct Test1
{
        int i;
        public void Display(){Console.WriteLine("Test1:>Display");}
        static void Main()
        {
                Test1 test1;
                test1.i = 10;
                test1.Display();
        }
}
```

8) E.g. Code.

```
6 references
struct MyStruct2
{
    public int i;
    1 reference
    public MyStruct2(int i) { this.i = i; }
    3 references
    public void Display() { Console.WriteLine("Method in Structure MyStruct2 : " + i); }
}
3 references
struct MyStruct3
{
    2 references
    public void Display() { Console.WriteLine("Method in Structure MyStruct3"); }
    0 references
    static void Main(string[] args)
    {
        //If struct has fields declared in it then "new" keyword is mandatory or
        //else we need to intialize the each and every fields explicitly
        //Type 1
        MyStruct2 myStruct2;
        //myStruct2.Display();  // Error if you try to access display method directly
                                // without initializing the fields
                                // Since while using field in display method it will be unassigned & un-initialized.
        myStruct2.i = 10;       // Fields should be initialized explicity before being used.
        myStruct2.Display();

        //Type 2
        MyStruct2 myStruct21 = new MyStruct2(); // Fields will be initialized by default constructor
        myStruct21.Display();

        //Type 3
        MyStruct2 myStruct22 = new MyStruct2(30); // Fields will be initialized by explicity constructor
        myStruct22.Display();
        Console.WriteLine("-----------------------------------------------------");
        //If struct has not fields declared in it then "new" keyword is optional
        MyStruct3 myStruct3; myStruct3.Display(); //Type 1
        MyStruct3 myStruct31 = new MyStruct3(); myStruct31.Display(); //Type 2
        Console.ReadLine();
    }
}
```

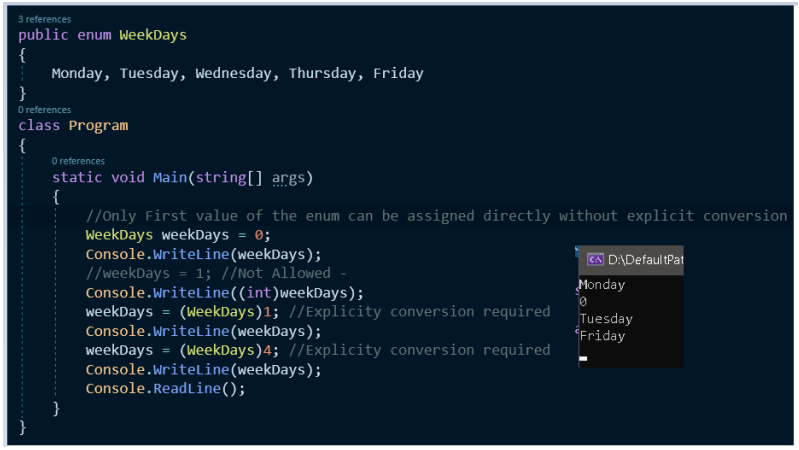
```
D:\DefaultPathForProjects\VS2019\CSharp\S
Method in Structure MyStruct2 : 10
Method in Structure MyStruct2 : 0
Method in Structure MyStruct2 : 30
-----------------------------------
Method in Structure MyStruct3
Method in Structure MyStruct3
```

Enumeration

1) Enumeration are user-defined type. Enumeration are set of named constant values.
2) How to define Enum Types

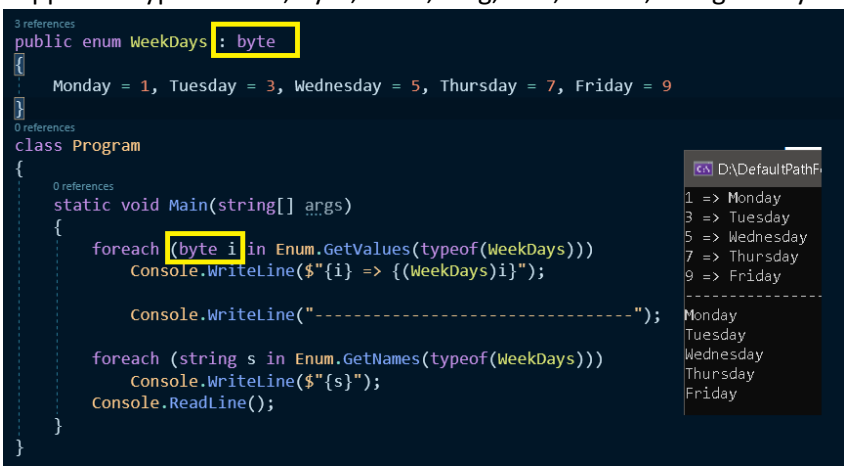
```
[<modifiers>] enum <EnumName> : [<Type>]
{
        -List of named constant values
}

public enum Colors {Red, Blue, Pink}
```

3) Use of Enums
    a. When you have list of supported values you can list those values under the Enum.
    b. It reduces human errors
    c. When we need to define a property or fields with only a set of values to choose from we use Enums.
4) As Enums are user defined type, so it always better to define an Enum directly under the namespace but it is also possible to define Enum under a class or structure.
5) Enum are value type category.
6) Only first value can be taken over by assigning zero not for remaining values, for remaining values we need to explicitly convert the Enum as shown below



7) Accessing all the Enum values using foreach
    a. GetNames
    b. GetValues

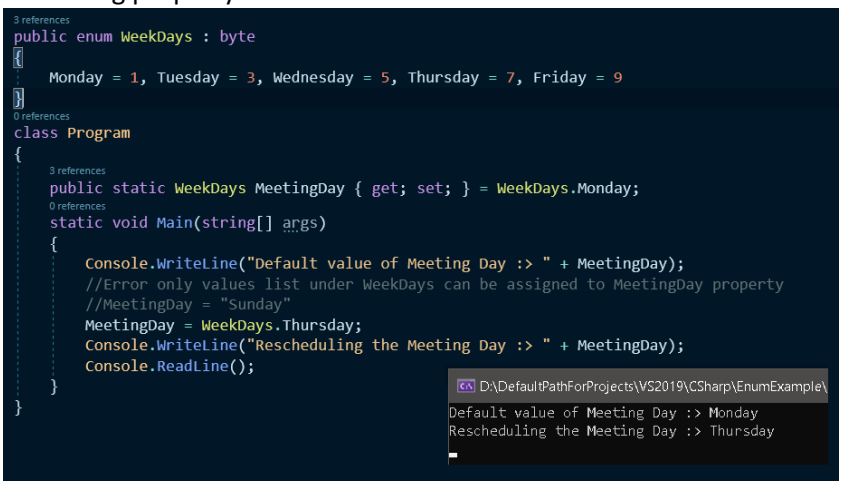
8) Int is default type for enum
   a. Supported types are int, byte, short, long, uint, ushort, ulong and sybte.



```
3 references
public enum WeekDays : byte
{
    Monday = 1, Tuesday = 3, Wednesday = 5, Thursday = 7, Friday = 9
}
0 references
class Program
{
    0 references
    static void Main(string[] args)
    {
        foreach (byte i in Enum.GetValues(typeof(WeekDays)))
            Console.WriteLine($"{i} => {(WeekDays)i}");

        Console.WriteLine("--------------------------------");

        foreach (string s in Enum.GetNames(typeof(WeekDays)))
            Console.WriteLine($"{s}");
        Console.ReadLine();
    }
}
```

```
D:\DefaultPathF
1 => Monday
3 => Tuesday
5 => Wednesday
7 => Thursday
9 => Friday
--------------------------------
Monday
Tuesday
Wednesday
Thursday
Friday
```

9) E.g. of Enum using property



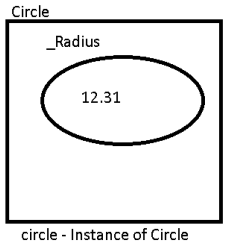
```
3 references
public enum WeekDays : byte
{
    Monday = 1, Tuesday = 3, Wednesday = 5, Thursday = 7, Friday = 9
}
0 references
class Program
{
    3 references
    public static WeekDays MeetingDay { get; set; } = WeekDays.Monday;
    0 references
    static void Main(string[] args)
    {
        Console.WriteLine("Default value of Meeting Day :> " + MeetingDay);
        //Error only values list under WeekDays can be assigned to MeetingDay property
        //MeetingDay = "Sunday"
        MeetingDay = WeekDays.Thursday;
        Console.WriteLine("Rescheduling the Meeting Day :> " + MeetingDay);
        Console.ReadLine();
    }
}
```

```
D:\DefaultPathForProjects\VS2019\CSharp\EnumExample\
Default value of Meeting Day :> Monday
Rescheduling the Meeting Day :> Thursday
```

Properties

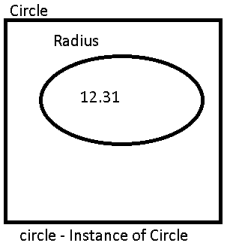
1) Property is a member of class using which we can expose values associated with a class to the outside environment.
2) Why we need Property
    a. Private variables are not accessible via instance of the class though memory allocation is done.

```
public class Circle { double _Radius = 12.31;}

public class TestCircle
{
        public static void Main()
        {
                Circle c = new Circle();
                //Error Since _Radius is declared as private.
                double radius = c._Radius;
        }
}
```



    b. Public variables are accessible via instance of the class and now any one can access the variable and get and set the value of the variable. Now we cannot restrict the use of the variable.

```
public class Circle { public double Radius = 12.31;}

public class TestCircle
{
        public static void Main()
        {
                Circle c = new Circle();
                //Public variable accessible via instance of the class
                double radius = c.Radius; //Getting old value
                c.Radius = 56.71;         //Setting new value
        }
}
```



    c. To restrict the use of the variable never declare them as public.
    d. We can define a Getter Methods to get the value of the private variables now user cannot set the private variable

```
public class Circle
{
        double _Radius = 12.31;
        public double GetRadius(){ return _Radius;} //Provides only Get Access
}

public class TestCircle
{
        public static void Main()
        {
                Circle c = new Circle();
                //Getting private variable value via method
                double radius = c.GetRadius(); //Getting old value
                // Error Setting a new value not possible
                // Since _Radius is private variable
                //c._Radius = 56.71;
        }
```

    e. We can define a Setter method to set the value of the private variables now user cannot get the private variable

```
public class Circle
{
        double _Radius = 12.31;
        public void SetRadius(double radius) //Provides only Set Access
        { _Radius = radius ;}
}

public class TestCircle
{
        public static void Main()
        {
                Circle c = new Circle();
                //Setting private variable value via method
                c.SetRadius(56.78); //Setting a new value
        }
}
```

f. We can define both Getter & Setter Method for the private variable now user can set and get the variable of the private variable via those methods.

```
public class Circle
{
        double _Radius = 12.31;
        public double GetRadius(){ return _Radius;} //Provides only Get Access
        public void SetRadius(double radius) //Provides only Set Access
        { _Radius = radius ;}
}

public class TestCircle
{
        public static void Main()
        {
                Circle c = new Circle();
                //Getting private variable value via method
                double radius = c.GetRadius(); //Getting old value
                //Setting private variable value via method
                c.SetRadius(56.78); //Setting a new value
        }
}
```

g. Property is combination of Getter & Setter method combine in one block of code.

```
public class Circle
{
        double _Radius = 12.31;
        public double RadiusProperty
        {
                get { return _Radius;}
                set { _Radius = value; }
        }
}

public class TestCircle
{
        public static void Main()
        {
                Circle c = new Circle();
                //Getting private variable value via Property
                double radius = c.RadiusProperty; //Getting old value
                //Setting private variable value via Property
                c.RadiusProperty = 56.71;         //Setting new value
        }
}
```

3) Declare a Property

```
[<modifiers>] <type> <Name>
{
 [ get { <stmt's> } ]  //Get Accessor
 [ set { <stmt's } ]   //Set Accessor
}
```

```
public double RadiusProperty
{
 get { return Radius; }
 set { Radius = value; }
}
```

```
public double RadiusProperty
{
 get { return Radius; }     //Represents a value returning method without parameter
 set { Radius = value; }    //Represents a non-value returing method with parameter
}
```

4) Three type of Properties

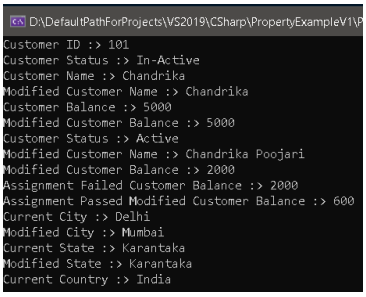
Read-Only Property	Write-Only Property	Read-Write Property

class Circle
{
double _Radius;
//Readonly Property
public double RadiusProperty
{
get {return _Radius;}
}
}
	class Circle
	{
	double _Radius;
	//WriteOnly Property
	public double RadiusProperty
	{
	set{_Radius = value;}
	}
	}
		class Circle
		{
		double _Radius;
		//Read-Write Property
		public double RadiusProperty
		{
		get {return _Radius;}
		set{_Radius = value;}
		}
		}

5) Advantages of using property
    a. Only get access, only set access or both get and set access can be applied on the property.
    b. We can impose the restriction on the data
    c. Conditional access and Conditional assignment through property.
6) Property Example 1-

```
5 references
public enum Cities { Delhi, Mumbai, Chenni, Kolkata, Bengaluru, Hyderbad }
3 references
public class Customer
{
    1 reference
    public Customer(int CustId, string CustName, bool Status, double Balance, Cities City, string State)
    {
        this._custId = CustId; this._custName = CustName; this.City = City;
        this._Status = Status; this._Balance = Balance; this._State = State;
    }
    int _custId; string _custName; bool _Status;
    double _Balance; Cities _City; string _State;
    1 reference
    public int CustId { get { return _custId; } set { _custId = value; } }
    5 references
    public string CustName { get { return _custName; } set { if(Status == true) _custName = value; } }
    7 references
    public bool Status { get { return _Status; } set { _Status = value; } }
    9 references
    public double Balance { get { return _Balance; }
        set
        {
            //Status should be active and value should be greater than or equal to 500
            // Then only assignment will be done.
            if (Status == true && value >= 500) _Balance = value;
        }
    }
    4 references
    public Cities City { get { return _City; } set { if (Status == true) _City = value; } }
    2 references
    public string State {  get { return _State; } protected set { if (Status == true) _State = value; } }
    //Auto-Implemented Property & Auto-Property Initializer
    1 reference
    public string Country { get; } = "India";
}
```

```
0 references
static void Main(string[] args)
{
    Customer customer = new Customer(101,"Chandrika", false, 5000.00, Cities.Bengaluru, "Karantaka");
    Console.WriteLine($"Customer ID :> {customer.CustId}");
    Console.WriteLine($"Customer Status :> {(customer.Status == true ? "Active" : "In-Active")}");

    Console.WriteLine($"Customer Name :> {customer.CustName}");
    customer.CustName = "Chandrika Poojari"; //  Assignment fails since Status is In-Active
    Console.WriteLine($"Modified Customer Name :> {customer.CustName}");

    Console.WriteLine($"Customer Balance :> {customer.Balance}");
    customer.Balance -=3000; // Assignment fails since Status is In-Active
    Console.WriteLine($"Modified Customer Balance :> {customer.Balance}");

    customer.Status = true;
    Console.WriteLine($"Customer Status :> {(customer.Status == true ? "Active" : "In-Active")}");

    customer.CustName = "Chandrika Poojari"; // Assignment succeeded since Status is Active
    Console.WriteLine($"Modified Customer Name :> {customer.CustName}");

    customer.Balance -= 3000; // Assignment succeeded since Status is Active
    Console.WriteLine($"Modified Customer Balance :> {customer.Balance}");

    customer.Balance -= 1600; // Assignment failed since Balance will be less than 500
    Console.WriteLine($"Assignment Failed Customer Balance :> {customer.Balance}");

    customer.Balance -= 1400; // Assignment succeeded since Balance will be greater than 500
    Console.WriteLine($"Assignment Passed Modified Customer Balance :> {customer.Balance}");

    Console.WriteLine($"Current City :> {customer.City}");
    customer.City = Cities.Mumbai;
    Console.WriteLine($"Modified City :> {customer.City}");

    Console.WriteLine($"Current State :> {customer.State}");
    //customer.State = "Maharastra"; B'Coz current class is not a child class of Customer
    Console.WriteLine($"Modified State :> {customer.State}");

    Console.WriteLine($"Current Country :> {customer.Country}");

    Console.ReadLine();
}
```



7) Introduction of Property in C# -
    a. Auto-Implemented Property was introduced in C# 3.0 but there was a restriction user need to create a read-write property.

        public string Country { get; set; }
    b. In C# 6.0, the above restriction was removed and now we can create only read-only or only write-only or read-write auto-implemented property.

        public string Test1 { get; } //Read-only Property
        public string Test2 { set; } //Write-only Property
        public string Test3 { get; set; } //Read-Write Property
    c. In C# 6.0, they introduced Auto-Property Initializer

        Public string Name { get; set; } = “Chandrahas”;
    d. The access specifier i.e. scope was only applicable to the property earlier but in C# 2.0 we can define scopes(access specifier) for the accessor also

        Earlier

            string _state;
            Public string State { get { return _state;} set {_state = value;}}

        C# 2.0 and later

            string _state;
            Public string State
            {
                get { return _state;}
                protected set {_state = value;}

            }
    e. Both the accessor cannot have access specifier at the same time.
    f. For more…
        https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/access-modifiers

Indexers

1) Indexers – Indexers allow instances of a class or struct to be indexed just like arrays.
2) If we define a indexer in a class it starts behaving like a virtual array

```
[<modifiers>] <type> this[<type> <name>]
{
        [get{<stmts>}]  //Get Accessor
        [set{<stmts>}]  //Set Accessor
}
```

Accessing via indexer via integer	Accessing indexer via string

`public object this[int index]`
`{`
`  get`
`  {`
`  if(index == 0){ return employeeId; }`
`  return null`
`  };`
`  set`
`  {`
`  if(index == 0){ employeeId = (int)value; }`
`  };`
`}`
	`public object this[string name]`
	`{`
	`  get`
	`  {`
	`  if(name.toUpper() == "EMPLOYEEID"){ return employeeId; }`
	`  return null`
	`  };`
	`  set`
	`  {`
	`  if(name.toUpper() == "EMPLOYEEID"){ employeeId = (int)value; }`
	`  };`
	`}`

3) Example

```
5 references
public enum Department {  Software, Accounts, HR, Admin , IT}
6 references
public enum Designation { Jr_Enigneer,Sr_Enigneer,Manager,Head_HR,Admin}
7 references
public enum EmployeeIndexerName { EmployeeId= 1, EmployeeName, Salary , Department, Designation }
3 references
public class Employee
{
    int employeeId; string employeeName; double salary; Department department; Designation designation;
    1 reference
    public Employee(int employeeId, string employeeName, double salary, Department department, Designation designation)
    {
        this.employeeId = employeeId;  this.employeeName = employeeName;   this.salary = salary;
        this.department = department;  this.designation = designation;
    }
    7 references
    public object this[int index]
    {
        get
        {
            if (index == 1) return employeeId;
            else if (index == 2) return employeeName;
            else if (index == 3) return salary;
            else if (index == 4) return department;
            else if (index == 5) return designation;
            return null;
        }
        set
        {
            if (index == 1) employeeId = (int)value;
            else if (index == 2)   employeeName = (string)value;
            else if (index == 3)   salary = (double)value;
            else if (index == 4)   department = (Department)value;
            else if (index == 5)   designation =(Designation)value;
        }
    }

    6 references
    public object this[string name]
    {
        get
        {
            if (name.ToUpper() == nameof(employeeId).ToUpper()) return employeeId;
            else if (name.ToUpper() == nameof(employeeName).ToUpper()) return employeeName;
            else if (name.ToUpper() == nameof(salary).ToUpper()) return salary;
            else if (name.ToUpper() == nameof(department).ToUpper()) return department;
            else if (name.ToUpper() == nameof(designation).ToUpper()) return designation;
            return null;
        }
        set
        {
            if (name.ToUpper() == nameof(employeeId).ToUpper()) employeeId = (int)value;
            else if (name.ToUpper() == nameof(employeeName).ToUpper()) employeeName = (string)value;
            else if (name.ToUpper() == nameof(salary).ToUpper()) salary = (double)value;
            else if (name.ToUpper() == nameof(department).ToUpper()) department = (Department)value;
            else if (name.ToUpper() == nameof(designation).ToUpper()) designation = (Designation)value;
        }
    }
}
```

```
0 references
static void Main(string[] args)
{
    Employee employee = new Employee(1001,"Chandrahas",2500000, Department.Software,Designation.Sr_Enigneer);
    Console.WriteLine($"Employee Id :> {employee[1]}");
    Console.WriteLine($"Employee Name :> {employee[2]}");
    Console.WriteLine($"Employee Salary :> {employee[3]}");
    Console.WriteLine($"Employee Department :> {employee[4]}");
    Console.WriteLine($"Employee Designation :> {employee[5]}");

    employee[2] = "Chandrahas Poojari";
    employee[EmployeeIndexerName.Salary.ToString()] = 30000000.00;
    employee[(int)EmployeeIndexerName.Designation] = Designation.Manager;

    Console.WriteLine("---------------After changes --- ------------------");

    Console.WriteLine($"Employee Id :> {employee[EmployeeIndexerName.EmployeeId.ToString()]}");
    Console.WriteLine($"Employee Name :> {employee[EmployeeIndexerName.EmployeeName.ToString()]}");
    Console.WriteLine($"Employee Salary :> {employee[EmployeeIndexerName.Salary.ToString()]}");
    Console.WriteLine($"Employee Department :> {employee[EmployeeIndexerName.Department.ToString()]}");
    Console.WriteLine($"Employee Designation :> {employee[EmployeeIndexerName.Designation.ToString()]}");

    Console.ReadLine();
}
```

```
D:\DefaultPathForProjects\VS2019\CSharp\IndexerExam
Employee Id :> 1001
Employee Name :> Chandrahas
Employee Salary :> 2500000
Employee Department :> Software
Employee Designation :> Sr_Enigneer
---------------After changes --- ------------
Employee Id :> 1001
Employee Name :> Chandrahas Poojari
Employee Salary :> 30000000
Employee Department :> Software
Employee Designation :> Manager
```

4) Example – 2

```
public class TempRecord
{
    // Array of temperature values
    float[] temps = new float[10]
    {
        56.2F, 56.7F, 56.5F, 56.9F, 58.8F,
        61.3F, 65.9F, 62.1F, 59.2F, 57.5F
    };

    // To enable client code to validate input
    // when accessing your indexer.
    public int Length => temps.Length;

    // Indexer declaration.
    // If index is out of range, the temps array will throw the exception.
    public float this[int index]
    {
        get => temps[index];
        set => temps[index] = value;
    }
}
```

```
static void Main()
{
    var tempRecord = new TempRecord();

    // Use the indexer's set accessor
    tempRecord[3] = 58.3F;
    tempRecord[5] = 60.1F;

    // Use the indexer's get accessor
    for (int i = 0; i < 10; i++)
    {
        Console.WriteLine($"Element #{i} = {tempRecord[i]}");
    }

    // Keep the console window open in debug mode.
    Console.WriteLine("Press any key to exit.");
    Console.ReadKey();
}
/* Output:
    Element #0 = 56.2
    Element #1 = 56.7
    Element #2 = 56.5
    Element #3 = 58.3
    Element #4 = 58.8
    Element #5 = 60.1
    Element #6 = 65.9
    Element #7 = 62.1
    Element #8 = 59.2
    Element #9 = 57.5
*/
```

5) Example 3

```
2 references
class SampleCollection<T>
{
    // Declare an array to store the data elements.
    private T[] arr = new T[100];

    // Define the indexer to allow client code to use [] notation.
    6 references
    public T this[int i]
    {
        get => arr[i];
        set => arr[i] = value;
    }
}
```

```
0 references
static void Main()
{
    var stringCollection = new SampleCollection<string>();
    stringCollection[0] = "Hello, World";
    Console.WriteLine(stringCollection[0]);

    var employeeCollection = new SampleCollection<Employee>();
    employeeCollection[0] = new Employee(1001,"Chandrahas Poojari",3000000.00,
                                         Department.Software,Designation.Sr_Enigneer);
    employeeCollection[1] = new Employee(1002, "Sunny", 3500000.00,
                                          Department.Software, Designation.Sr_Enigneer);
    Console.WriteLine($"Employee Data :> {employeeCollection[0]}");
    Console.WriteLine($"Employee Data :> {employeeCollection[1]}");
    Console.ReadLine();
}
```

In Employee class toString method was overridden to get this result

```
D:\DefaultPathForProjects\VS2019\CSharp\IndexerExample\IndexerExample\bin\Debug\netcoreapp3.1\IndexerExample.exe
Hello, World
Employee Data :> Employee Id :> 1001,    Employee Name :> Chandrahas Poojari,    Employee Salary :> 3000000
Employee Department :> Software,         Employee Designation :> Sr_Enigneer

Employee Data :> Employee Id :> 1002,    Employee Name :> Sunny,         Employee Salary :> 3500000
Employee Department :> Software,         Employee Designation :> Sr_Enigneer
```

6) Indexer Overview

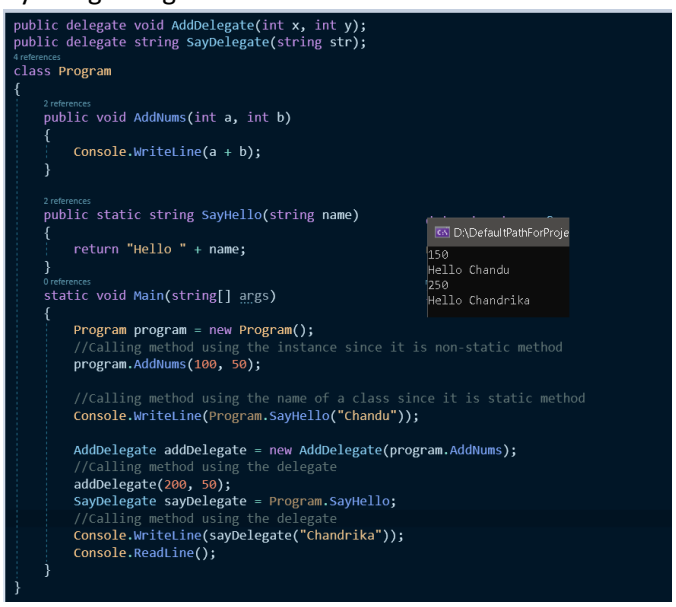
- Indexers enable objects to be indexed in a similar manner to arrays.
- A `get` accessor returns a value. A `set` accessor assigns a value.
- The this keyword is used to define the indexer.
- The value keyword is used to define the value being assigned by the `set` accessor.
- Indexers do not have to be indexed by an integer value; it is up to you how to define the specific look-up mechanism.
- Indexers can be overloaded.
- Indexers can have more than one formal parameter, for example, when accessing a two-dimensional array.

7) For More

https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/indexers/#:~:text=Indexers%20allow%20instances%20of%20a,that%20their%20accessors%20take%20parameters.

Delegates

1) Delegate is a type safe function pointer
2) A Delegate holds the reference of a method and then calls the method for execution.
3) In C#, we can call a method by three ways
    a. By using the instance of a class, if it is a non-static method.
    b. By using the name of a class, if it is a static method.
    c. By using Delegate.



4) To call a method by using a delegate we have 3 steps
    a. Define a delegate

```
[<modifiers>] delegate void|type <Name>([<parameter list>]);

//Delegate for method having two int parameter and void return type
public delegate void AddDelegate(int a,int b);
public void AddNumber(int x,int y) {}

//Delegate for method having one string parameter and string return type
public delegate string SayDelegate(string str);
public static string SayHello(string name){ return "Hello" +name}
```

    b. Instantiating the delegate-Creating instance of delegate since it is a reference type.

```
[<DelegateName>] [<ObjectName>] = new [<DelegateName>]([<MethodNameWithParenthesis>]);
AddDelegate addDelegate = new AddDelegate(p.AddNums);
SayDelegate sayDelegate = new SayDelegate(Program.SayHello);
                          or
[<DelegateName>] [<ObjectNameofDelegate>] = [<MethodNameWithParenthesis>];
AddDelegate addDelegate = p.AddNums;
SayDelegate sayDelegate = Program.SayHello;
```

    c. Now calling the delegate by passing the required parameter values, so that internally the method which is bounded with the delegate gets executed.

```
[<ObjectNameofDelegate>]([<RequiredParameter>]);
addDelegate(200,50);
sayDelegate("Chandrika");
                          Or
[<ObjectNameofDelegate>].Invoke([<RequiredParameter>]);
addDelegate.Invoke(200,50);
sayDelegate.Invoke("Chandrika");
```

5) While defining a delegate, we should remember two things
    a. The return type of delegate should be exactly be same as return type of the method.
    b. The parameters of delegate should be same as parameters of the method.

```
//Delegate for method having two int parameter and void return type
public delegate void AddDelegate(int a,int b);
public void AddNumber(int x,int y) {}

//Delegate for method having one string parameter and string return type
public delegate string SayDelegate(string str);
public static string SayHello(string name){ return "Hello" +name}
```

6) Delegate is reference type and it should be define under namescape (namespace is a logical container of types).
7) Example of Delegate – Define, instantiate & calling/Invoking the delegate

```
namespace DelegateExample
{
    //Step-1) Defining the a Delegate
    public delegate void AddDelegate(int x, int y);
    public delegate string SayDelegate(string str);
    3 references
    class Program
    {
        1 reference
        public void AddNums(int a, int b)
        {
            Console.WriteLine(a + b);
        }

        1 reference
        public static string SayHello(string name)
        {
            return "Hello " + name;
        }
        0 references
        static void Main(string[] args)
        {
            Program program = new Program();

            //Step-2) Instantiating the delegate
            AddDelegate addDelegate = new AddDelegate(program.AddNums);
            SayDelegate sayDelegate = Program.SayHello;

            //Step-3) Calling the method via delegate
            addDelegate(200, 50);  addDelegate.Invoke(100, 50);
            Console.WriteLine(sayDelegate("Chandrika"));
            Console.WriteLine(sayDelegate.Invoke("Suraj"));
            Console.ReadLine();
        }
    }
}
```

```
D:\DefaultPathFo
250
150
Hello Chandrika
Hello Suraj
```

Multicast Delegate

1) Multicast delegate holds the reference of more than one method and calls the all those methods for execution at once when the delegate is invoked.
2) We use (+=) for multicast delegate e.g.

```
//Instaniting the delegate
[DelegateName] [DelegateObject] = [MethodName1WithoutParenthesis];
//Performing Mulitcast Delegate
[DelegateObject] += [MethodName2WithoutParenthesis];
//Delegate will call both the methods i.e. MethodName1 & MethodName2
[DelegateObject]();
```

3) Advantage of using a multicast delegate
    a. When we need to call multiple methods with same signature with one invocation, we can use multicast delegate.
4) Rough Example of multicast delegate.

```
public delegate void CallDelegate();

class TestMulticastDelegate
{
        public void MethodName1() { Console.WriteLine("MethodName1")}
        public void MethodName2() { Console.WriteLine("MethodName1")}

        public void Initalize()
        {
                //Instaniting the delegate
                CallDelegate callDelegate = MethodName1;
                //Performing Mulitcast Delegate
                callDelegate += MethodName2;
                //Delegate will call both the methods i.e. MethodName1 & MethodName2
                callDelegate();
        }
}

//Output
MethodName1
MethodName2
```

5) Don't while using multicast delegates
    a. If method has return type or out parameter then **don't use multicast delegate** because the return type or out parameter value will be overridden by the last executed methods value.
6) E.g. for multicast delegate

```
namespace DelegateExample
{
    public delegate void RectangleDelegate(double Width, double Height);
    2 references
    public class Rectangle
    {
        2 references
        public void GetArea(double Width, double Height)
        { Console.WriteLine($"Area of Rect :> {Width * Height}"); }
        2 references
        public void GetPerimeter(double Width, double Height)
        { Console.WriteLine($"Area of Rect :> {2 * (Width + Height)}"); }
        0 references
        public static void Main()
        {
            Rectangle rectangle = new Rectangle();
            rectangle.GetArea(100,50); rectangle.GetPerimeter(100, 50); //Normal
            Console.WriteLine("====================");
            //instaniting the delegate
            RectangleDelegate rectangleDelegate = rectangle.GetArea;
            // using += make the delegate a multicast delegate
            rectangleDelegate += rectangle.GetPerimeter;
            //Invocation Type 1
            rectangleDelegate(200, 50);
            Console.WriteLine("====================");
            //Invocation Type 2
            rectangleDelegate(200, 50);
            Console.ReadLine();
        }
    }
}
```

```
D:\DefaultPathForProjects\V
Area of Rect :> 5000
Area of Rect :> 300
====================
Area of Rect :> 10000
Area of Rect :> 500
====================
Area of Rect :> 10000
Area of Rect :> 500
```

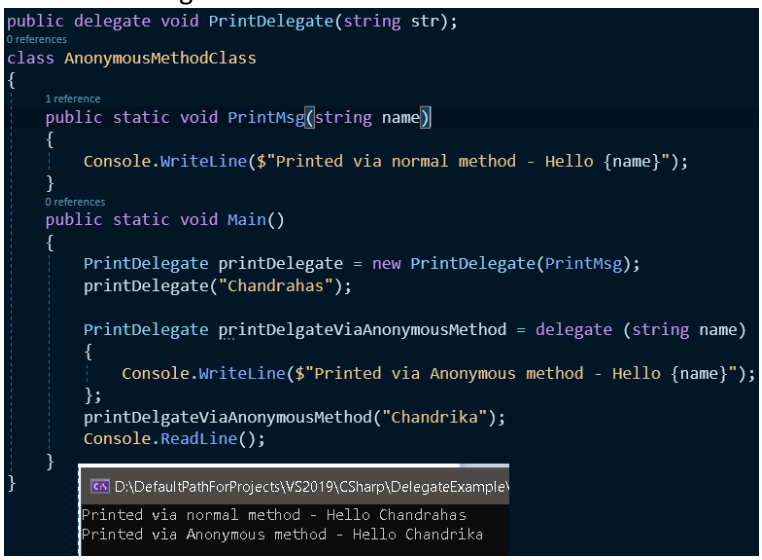
Anonymous Methods – Delegates

1) A method without name is called as Anonymous methods. It is also known as inline delegate.
2) Anonymous methods provides us a way to create a delegate object without writing separate method.
3) Anonymous method is declared when delegate is initialized.
4) Anonymous methods are used by when we have lesser line of code to execute.
5) How to define anonymous method

```
[<DelegateName>] [<DelegateObjectName>] = delegate([<Parameter List>])
{
        --Statments
};
[<DelegateObjectName>]([<Parameter Inputs List>]);
                    or
[<DelegateName>] [<DelegateObjectName>] = delegate
{
        --Statments
};
[<DelegateObjectName>]();
```

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

class AnonymousMethods
{
        public static void Main()
        {
                //Anonymous method is created when Delegate is initialized
                PrintDelegate printDelegate= delegate(string str)
                {
                        Console.WriteLine($"Hello {str}");
                };

                printDelegate("Chandrahas");
        }
}
```

6) Anonymous method E.g. 1



Reference –

https://www.tutorialsteacher.com/csharp/csharp-anonymous-method
https://www.youtube.com/watch?v=OJktQUsGiBY&t=1s

7) Anonymous method E.g. 2

```
0 references
class AnonymousMethodClass2
{
    private delegate int CalculateDelegate(int a,int b);
    0 references
    public static void Main()
    {
        CalculateDelegate addDelegate = delegate (int a, int b)
        {
            return a + b;
        };
        CalculateDelegate mulDelegate = delegate (int a, int b)
        {
            return a * b;
        };
        Console.WriteLine($"Add Delegate :> {addDelegate(10,20)}");
        Console.WriteLine($"Multiple Delegate :> {mulDelegate(10, 20)}");
        Console.ReadLine();
    }
}
```

```
D:\DefaultPathForProjects\VS20
Add Delegate :> 30
Multiple Delegate :> 200
```

8) Anonymous method E.g. 3 – Passing Anonymous method as method parameter

```
2 references
public class AnonymousMethodClass3
{
    public delegate void PrintNumberDelegate(int number);
    1 reference
    public void IncrementAndPrintNumber(PrintNumberDelegate printNumberDelegate, int incrementalNumber)
    {
        incrementalNumber += 200;
        printNumberDelegate(incrementalNumber);

    }
}

0 references
class Program2
{
    0 references
    public static void Main()
    {
        AnonymousMethodClass3 anonymousMethodClass3 = new AnonymousMethodClass3();
        //Passing Anonymous method as method parameter
        anonymousMethodClass3.IncrementAndPrintNumber(
                delegate (int number)
                {
                    Console.WriteLine($"Incremented number :> {number}");
                }, 100
            );
        Console.ReadLine();
    }
}
```

```
D:\DefaultPathForProjects\VS2019\CSharp
Incremented number :> 300
```

9) Anonymous method E.g. 4 – Anonymous method can be used as event handlers

```
2 references
public class AnonymousMethodClass4
{
    public event EventHandler MySaveSuccessMessage;
    1 reference
    public void Save()
    {
        // Saving Activity
        MySaveSuccessMessage(this, EventArgs.Empty);
        //MySaveSuccessMessage.Invoke(this, EventArgs.Empty);
    }
}

0 references
class Program3
{
    0 references
    public static void Main()
    {
        AnonymousMethodClass4 anonymousMethodClass4 = new AnonymousMethodClass4();
        //Anonymous method used by Event Handler
        anonymousMethodClass4.MySaveSuccessMessage += delegate(object sender, EventArgs eventArgs)
        {
            Console.WriteLine("Saved Successful");
        };
        anonymousMethodClass4.Save();

        Console.ReadLine();
    }
}
```

```
D:\DefaultPathForProj
Saved Successful
```

10) Some points to remember while using Anonymous methods

- It cannot contain jump statement like goto, break or continue.
- It cannot access ref or out parameter of an outer method.
- It cannot have or access unsafe code.
- It cannot be used on the left side of the is operator.

Lambda Expression-Delegate

1) Lambda Expression is a shorthand for Anonymous methods
2) Operator "=>" is used for Lambda expression.
3) Parameter type can be ignored while declaring the Lambda expression because delegate is aware of the return type and parameter type.
4) How to declare Lambda expression

```
() => {};
```

```
[<DelegateName>] [<DelegateObjectName>] = (<ParameterName>) =>
{
        //Statement
}
[<DelegateObjectName>]([<ParameterInput>]);
```

5) E.g. 1

```
0 references
public class LambdaExpression
{
    private delegate int CalculateDelegate(int a, int b);
    0 references
    public static void Main()
    {
        //Type 1 - you can specify parameter type while declaring Lambda
        CalculateDelegate addDelegate = (int a, int b) => { return a + b; };
        //Type 2 - you can ignore parameter type while declaring Lambda
        CalculateDelegate mulDelegate = (a, b) => { return a * b; };

        Console.WriteLine($"Add Delegate :> {addDelegate(10, 20)}");
        Console.WriteLine($"Multiple Delegate :> {mulDelegate(10, 20)}");
        Console.ReadLine();
    }
}
```

```
D:\DefaultPathForProjects\VS2
Add Delegate :> 30
Multiple Delegate :> 200
```

6) E.g. 2

```
2 references
public class LambdaExpression2
{
    public delegate void PrintNumberDelegate(int number);
    public event EventHandler MySaveSuccessMessage;
    1 reference
    public void IncrementAndPrintNumber(PrintNumberDelegate printNumberDelegate, int incrementalNumber)
    {
        incrementalNumber += 200;
        printNumberDelegate(incrementalNumber);
    }
    1 reference
    public void Save()
    {
        // Saving Activity
        MySaveSuccessMessage(this, EventArgs.Empty);
    }
}
0 references
class LEProgram
{
    0 references
    public static void Main()
    {
        LambdaExpression2 lambdaExpression2 = new LambdaExpression2();
        //Anonymous method used by Event Handler
        lambdaExpression2.MySaveSuccessMessage += (sender, eventArgs) =>
        {
            Console.WriteLine("Saved Successful");
        };
        //Passing Anonymous method as method parameter
        lambdaExpression2.IncrementAndPrintNumber(
                (int number) => {
                    Console.WriteLine($"Incremented number :> {number}");
                    lambdaExpression2.Save();
                }, 100
            );
        Console.ReadLine();
    }
}
```

```
D:\DefaultPathForProjects\VS2019\
Incremented number :> 300
Saved Successful
```

Func, Action and Predicate Delegates – Generic Delegates

Func	Action	Predicate
1) It is a delegate for a function that may or may not take parameter and return a value. 2) Func<> takes maximum of 16 input parameter and 1 output parameter. 3) Returns a value.	1) It is a delegate for a function that may or may not take parameter and doesn't return a value. 2) Action<> takes minimum of Zero input parameter and maximum of **16 input parameter** and return type would be void. 3) Doesn't return any value.	1) It is a specialized version of Func<> that takes only one argument and always returns boolean value as result. 2) Predicate takes only one input parameter and **return type would be Boolean**. 4) Returns only Boolean.

1) How to Func, Action & Predicate Delegate look like

```
public delegate TResult Func<in T,out TResult>(T arg);
```

```
public delegate void Action<in T>(T obj);
```

```
public delegate bool Predicate<in T>(T obj);
```

2) E.g. Action

```
0 references
public class GenericDelegate
{
    private delegate void PrintDelegate(string msg);
    private delegate void MeaningLessDelegate();
    0 references
    public static void Main()
    {
        PrintDelegate printDelegate = (msg) => Console.WriteLine($"Via PrintDelegate => {msg}");
        printDelegate?.Invoke("Hi All");
        //Action takes maximum of 16 input parameter & return type is always void
        Action<string> printAction = (msg) => Console.WriteLine($"Via Action<string> => {msg}");
        printAction("My Name is Chandrahas");
        printAction?.Invoke("I am learning C#"); //Null Check

        Console.WriteLine("-------------");
        MeaningLessDelegate meaningLessDelegate = () => Console.WriteLine($"Via meaningLessDelegate");
        meaningLessDelegate();

        //Action can also take zero parameters
        Action meaningLessAction = () => Console.WriteLine($"Via meaningLessAction");
        meaningLessAction();
        meaningLessAction?.Invoke(); //Null Check

        Console.ReadLine();
    }
}
```

```
D:\DefaultPathForProjects\VS2019\CSharp\DelegateExa
Via PrintDelegate => Hi All
Via Action<string> => My Name is Chandrahas
Via Action<string> => I am learning C#
-------------
Via meaningLessDelegate
Via meaningLessAction
Via meaningLessAction
```

3) E.g. Func

```
0 references
public class GenericDelegate2
{
    private delegate double CalculateDelegate(int a, float b, double c);
    0 references
    public static void Main()
    {
        CalculateDelegate addDelegate = (x , y, z) =>  (double)(x + y + z);
        Console.WriteLine($"Via addDelegate :> {addDelegate?.Invoke(10,20.0f,30.0)}");

        //Func can take maximum of 16 input and 1 out parameter
        Func<int, float, double> multiplyFunc = (x, y) => (double)(x * y);
        Func<int, float, double, double> addFunc = (x, y, z) => (double)(x + y+ z);
        //Func can be treated as Predicate<int>
        Func<int, bool> isGreaterThan5Func = (value) => (value > 5 ? true : false);

        Console.WriteLine($"Via addFunc :> {addFunc?.Invoke(10, 20.0f, 30.0)}");
        Console.WriteLine($"Via multiplyFunc :> {multiplyFunc?.Invoke(10, 20.0f)}");
        Console.WriteLine($"Via isGreaterThan5Func :> {isGreaterThan5Func?.Invoke(10)}");
        Console.WriteLine($"Via isGreaterThan5Func :> {isGreaterThan5Func?.Invoke(4)}");

        Console.ReadLine();
    }
}
```

```
D:\DefaultPathForProjects\VS2019\CSha
Via addDelegate :> 60
Via addFunc :> 60
Via multiplyFunc :> 200
Via isGreaterThan5Func :> True
Via isGreaterThan5Func :> False
```

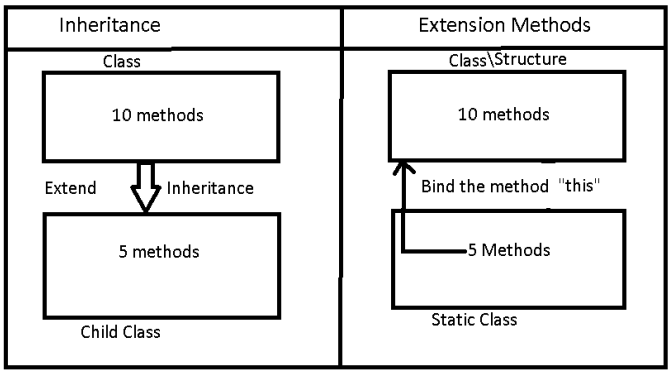
4) E.g. Predicate

```
0 references
public class GenericDelegate3
{
    private delegate bool IsGreateThan10Delegate(int number);
    0 references
    public static void Main()
    {
        IsGreateThan10Delegate isGreateThan10Delegate = (number) => (number > 10 ? true : false);
        Console.WriteLine($"Via addDelegate :> {isGreateThan10Delegate?.Invoke(10)}");

        //Predicate only take 1 input parameter and always returns boolean.
        Predicate<int> isGreaterThan10Predicate = (number) => (number > 10 ? true : false);
        Predicate<string> isStringGreaterThan5Predicate = (stringvalue) => (stringvalue.Length > 10 ? true : false);

        Console.WriteLine($"Via isGreateThan10Predicate :> {isGreaterThan10Predicate?.Invoke(10)}");
        Console.WriteLine($"Via isStringGreaterThan5Predicate :> {isStringGreaterThan5Predicate?.Invoke("Hi Predicate")}");

        //Func can be treated as Predicate<int>
        Func<int, bool> isGreaterThan5Func = (value) => (value > 5 ? true : false);
        Console.WriteLine($"Via isGreaterThan5Func :> {isGreaterThan5Func?.Invoke(10)}");

        Console.ReadLine();
    }
}
```

```
D:\DefaultPathForProjects\VS2019\CSharp\DelegateExar
Via addDelegate :> False
Via isGreateThan10Predicate :> False
Via isStringGreaterThan5Predicate :> True
Via isGreaterThan5Func :> True
```

Extension Methods

1) This new feature has been introduced in C# 3.0.
2) Extension method is a mechanism of adding new methods into an existing class or structure without modifying the source code of the original type.
    a. The original type (Class or Structure) doesn't require re-compilation.
    b. We don't require permission from the original type (Class or Structure).
3) Inheritance is a mechanism using which we can extend the functionality of the class.
Limitation using inheritance for extending
    a. We cannot apply inheritance on Sealed Class
    b. We cannot apply inheritance on Structure. So if the original type is not a class and it's a structure.
4) Diagram



5) Defining extension methods

```
static class [<classname>]
{
        public static [<ReturnType>] [<ExtensionMethodName()>](this [<NameOfBindingClass>])
        {
                //Statement
        }

        public static [<ReturnType>] [<ExtensionMethodName()>](this [<NameOfBindingClass>],[<Formal ParameterList>])
        {
                //Statement
        }
}
```

6) For binding the extension method to the class we need to use "**this**" keyword.
7) Extension methods are define as static but once they are bounded with any class or structure they turn into non-static method and hence we are able to call it using an instance variable of the binding class.

```
class Program
{
        public void Test1(){ Console.WriteLine("Test Method");}
        public static void Main()
        {
                Program p = new Program();
                p.Test1();
                //Extenstion method called via instance variable
                p.ExtendedTest2();
        }
}

static class ExtendedProgram
{
        //Defined the method as static
        public static void ExtendedTest2(this Program p)
        { Console.WriteLine("Extended Test Method"); }
}
```

8) If an Extension method is defined with the same name and signature of an existing method of binding class, then extension method will not be called and the first preference always goes to the original method(Binding class method) only

```
3 references
class Program //Assume Source code is not avaliable
{
    // Preferance wil be given to OG method
    1 reference
    public void Test1() { Console.WriteLine("Test1 Method"); }
}
0 references
static class ExtendProgram  //Extension Class
{
    //Same name and Signature of Extension method
    0 references
    public static void Test1(this Program p) { Console.WriteLine("Extended Test1 Method"); }
}
0 references
public class TestProgram
{
    0 references
    static void Main(string[] args)
    {
        Program p = new Program();
        p.Test1();
        Console.ReadLine();
    }
}
```

```
D:\DefaultPathForProje
Test1 Method
```

9) The first parameter of an extension method should be the name of the type (class or strut name) to which that method has to be bounded with and this parameter is not taken into consideration while calling the extension method via instance variable of binding class.
    a. If an extension method is defined with “n” parameters then while calling it we only need to pass “n-1” parameters only because the binding parameter is excluded.
10) An extension method should have one and only one binding parameter and it should be in the first place of the parameter list.

```
3 references
class Program2
{
    1 reference
    public void Test1() { Console.WriteLine("Test Method"); }
}

0 references
static class ExtendedProgram2
{
    //1 Binding Parameter & 1 Formal Parameter = 2 Parameters defined
    //Defined with "n" parameters
    1 reference
    public static void ExtendedTest2(this Program2 p, int i)
    { Console.WriteLine($"Extended Test Method :> {i}"); }
}
0 references
public class TestProgram2
{
    0 references
    public static void Main()
    {
        Program2 p = new Program2();
        p.Test1();
        //Extenstion method called via instance variable
        //Calling Extension method with only formal parameter
        //Binding Parameter will be excluded while calling the Ext. Method
        //Called with only "n-1" parameter
        p.ExtendedTest2(10);
    }
}
```

```
Microsoft Visual Studio Debug Console
Test Method
Extended Test Method :> 10
```

11) Extension method can be extend functionality to Structure

```
0 references
static class ExtendedProgram3
{
    //Extend Factorial to Integer Struct
    3 references
    public static long Factorial(this Int32 value)
    {
        return (value == 0 || value == 1 ? 1 : (value == 2 ? 2 : value * Factorial(value - 1)));
    }
}
0 references
public class TestProgram3
{
    0 references
    public static void Main()
    {
        //Integer is a struct
        int number = 5;
        Console.WriteLine($"Factorial of {number} is {number.Factorial()}");

        number = 10;
        Console.WriteLine($"Factorial of {number} is {number.Factorial()}");

        Console.ReadLine();
    }
}
```

```
Select D:\DefaultPathForProjects\VS
Factorial of 5 is 120
Factorial of 10 is 3628800
```

12) Extension method can extend functionality to sealed classes

```
0 references
static class ExtendedProgram4
{
    //Extending functionality to a sealed class
    2 references
    public static string ToProperCase(this string oldString)
    {
        if (oldString.Trim().Length == 0) return oldString;
        string newString = null;
        oldString = oldString.ToLower();
        foreach (string str in oldString.Split(' '))
        {
            if (str.Trim() == "") continue;
            char[] cArray = str.ToCharArray();
            cArray[0] = Char.ToUpper(cArray[0]);
            if (newString == null)
                newString = new String(cArray);
            else
                newString += $" {new String(cArray)}";
        }
        return newString;
    }
}

0 references
public class TestProgram4
{
    0 references
    public static void Main()
    {
        //string is a sealed class
        string value = "hi, i am chandrahas pOojari.";
        Console.WriteLine(value.ToProperCase());
        value = "hi,  i am chandrahas pOojari. ";
        Console.WriteLine(value.ToProperCase());
        Console.ReadLine();
    }
}
```

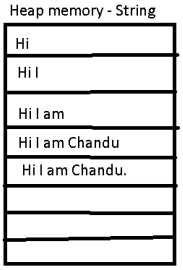
```
D:\DefaultPathForProjects\VS2019\C
Hi, I Am Chandrahas Poojari.
Hi, I Am Chandrahas Poojari.
```

Differences between String and StringBuilder

1) Strings are immutable (unable to be changed) it creates new copies if any manipulation is conducted.
2) StringBuilder is mutable – It doesn't creates new copies if any manipulation is conducted.
3) Heap memory representation of String & StringBuilder

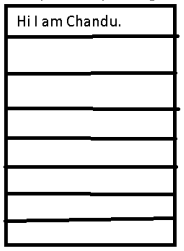
```
public void TestString()
{
        string str = "Hi";
        str += " I";
        str += " am";
        str += " Chandu";
        str += ".";
}
```

5 copys are created.
Heap memory - String



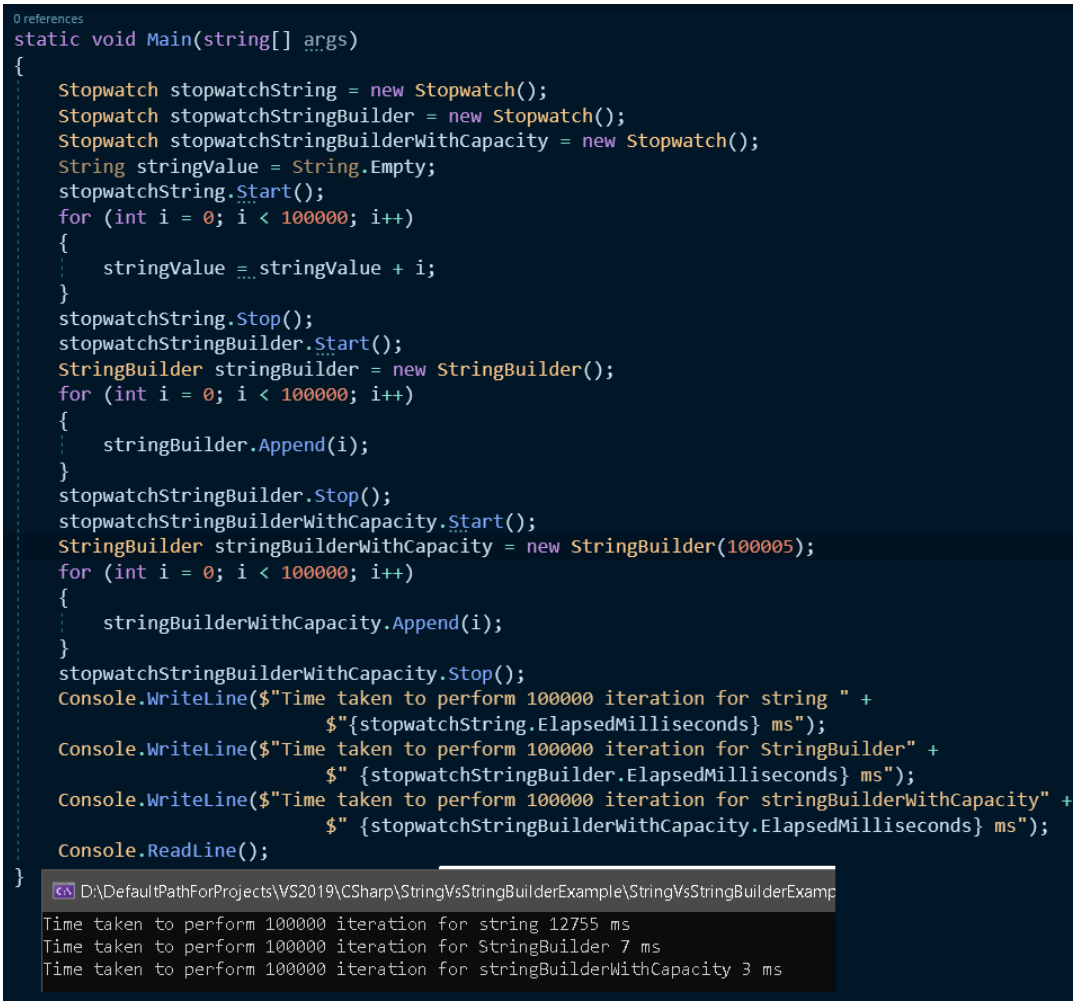

```
public void TestStringBuilder()
{
        StringBuilder sb = new StringBuilder("Hi");
        sb.Append(" I");
        sb.Append(" am");
        sb.Append(" Chandu");
        sb.Append(".");
}
```

Only 1 Copy is created.
Heap memory- StringBuilder



4) When to use
    a. String
        i. When there is no frequent changes in the initial string value assigned to it.
    b. StringBuilder
        i. When we are sure there will be frequent changes in the initial string value assigned to it.
5) String is slower than string builder & StringBuilder is slower than StringBuilder were initial capacity is provided.
6) String is under "System" namespace
7) StringBuilder is under "System.Text" namespace.

8) Performance wise it is better to use StringBuilder & if you provide StringBuilder with initial capacity then it enhances the performance more.
9) E.g. of Performance between string, string builder & string builder with initial capacity

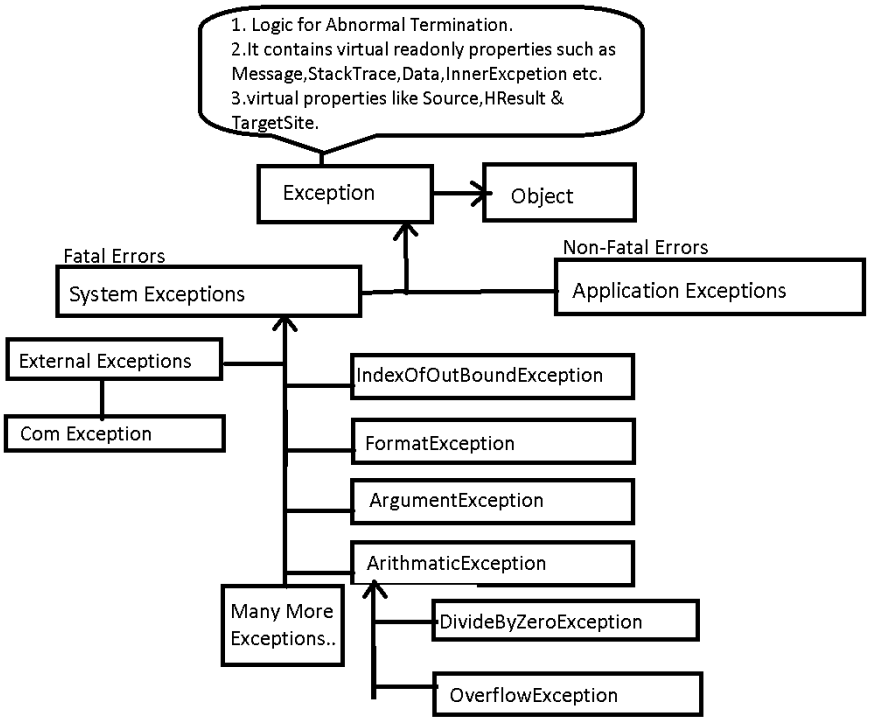


```
0 references
static void Main(string[] args)
{
    Stopwatch stopwatchString = new Stopwatch();
    Stopwatch stopwatchStringBuilder = new Stopwatch();
    Stopwatch stopwatchStringBuilderWithCapacity = new Stopwatch();
    String stringValue = String.Empty;
    stopwatchString.Start();
    for (int i = 0; i < 100000; i++)
    {
        stringValue = stringValue + i;
    }
    stopwatchString.Stop();
    stopwatchStringBuilder.Start();
    StringBuilder stringBuilder = new StringBuilder();
    for (int i = 0; i < 100000; i++)
    {
        stringBuilder.Append(i);
    }
    stopwatchStringBuilder.Stop();
    stopwatchStringBuilderWithCapacity.Start();
    StringBuilder stringBuilderWithCapacity = new StringBuilder(100005);
    for (int i = 0; i < 100000; i++)
    {
        stringBuilderWithCapacity.Append(i);
    }
    stopwatchStringBuilderWithCapacity.Stop();
    Console.WriteLine($"Time taken to perform 100000 iteration for string " +
                      $"{stopwatchString.ElapsedMilliseconds} ms");
    Console.WriteLine($"Time taken to perform 100000 iteration for StringBuilder" +
                      $" {stopwatchStringBuilder.ElapsedMilliseconds} ms");
    Console.WriteLine($"Time taken to perform 100000 iteration for stringBuilderWithCapacity" +
                      $" {stopwatchStringBuilderWithCapacity.ElapsedMilliseconds} ms");
    Console.ReadLine();
}
```

```
D:\DefaultPathForProjects\VS2019\CSharp\StringVsStringBuilderExample\StringVsStringBuilderExamp
Time taken to perform 100000 iteration for string 12755 ms
Time taken to perform 100000 iteration for StringBuilder 7 ms
Time taken to perform 100000 iteration for stringBuilderWithCapacity 3 ms
```

Exceptions and Exception Handling

1) Two types of Errors
    a. Compile Time Errors
    b. Runtime Errors- Cause abnormal termination of the program.
2) **Exception is a class responsible for abnormal termination of a program whenever runtime error occurs in a program.**
3) Compile Time Errors occurs due to syntax missing and other various reasons
4) Runtime Errors occurs
    a. Wrong implementation of logic –
    b. Wrong value to supplied to the code
    c. Missing Resources
    d. Many more ----
5) Few examples of Exception classes are listed below
    a. IndexOutOfBoundException
    b. DivideByZeroException
    c. OverflowException
    d. FormatException
    e. And many more..
6) Exception



Exception Handling

1) Exception Handling is process of stopping abnormal termination of the program whenever a runtime error is occurring.
2) Advantages of Exception Handling
    a. Abnormal termination stops so that statement that are not related with errors can be executed.
    b. We can display user friendly error messages to end users so that we can describe about the error.
    c. We can perform corrective action to resolve the problems that may come into picture due to the error.
3) How to handle exception

```
try
{
        -Statments which will cause runtime errors
        -Statments which doesn't require execution when
                the runtime error occured.
}
catch(<Exception Class Name> <variable>)
{
        -Statments which should execute only when there is
                runtime error.
}
```

4) If any statement of try causes an error, from that line of code the control jumps to catch block searching for matching catch block
    a. If there is matching catch block that can handle the exception, abnormal termination stops it will execute the code in the catch block and then to first line after the catch block.
    b. If there is no matching catch block for the error then abnormal termination will occur.
5) Finally block –
    a. Mandatory execution for code which is inside the finally block.
    b. Finally block executes even if there is abnormal termination of program.

Try Catch Finally	Try Finally	Try Catch

try
{
-Open a file on HD
-Write into the file
}
catch(Exception ex)
{
-Error Messages
}
finally
{
-Close the file
}
	try
	{
	-Open a file on HD
	-Write into the file
	}
	finally
	{
	-Close the file
	}
		try
		{
		-Open a file on HD
		-Write into the file
		-Close the fileS
		}
		catch(Exception ex)
		{
		-Error Messages
		-Close the file
		}

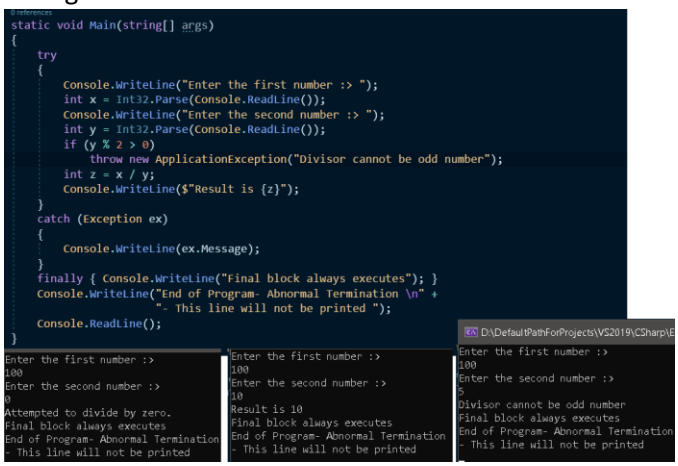
6) Try Catch Finally –Exception are handled and mandatory execution of code inside finally block.
7) Try Catch – Exceptions are handled.
8) Try Finally – Exception are not handled, abnormal termination is guaranteed but mandatory execution of code inside finally block.

Exceptions – Applications Exceptions

1) System exceptions are thrown by CLR on bases of predefined conditions i.e. overflow, format and divide by zero etc.
2) Application exceptions are thrown & created by programmer on the bases of business logic.
3) How to create & throw Application Exception

```
ApplicationException ex = new ApplicationException("<ErrorMessage>");
throw ex;
                                    OR
throw new ApplicationException("<Error Message>");
```

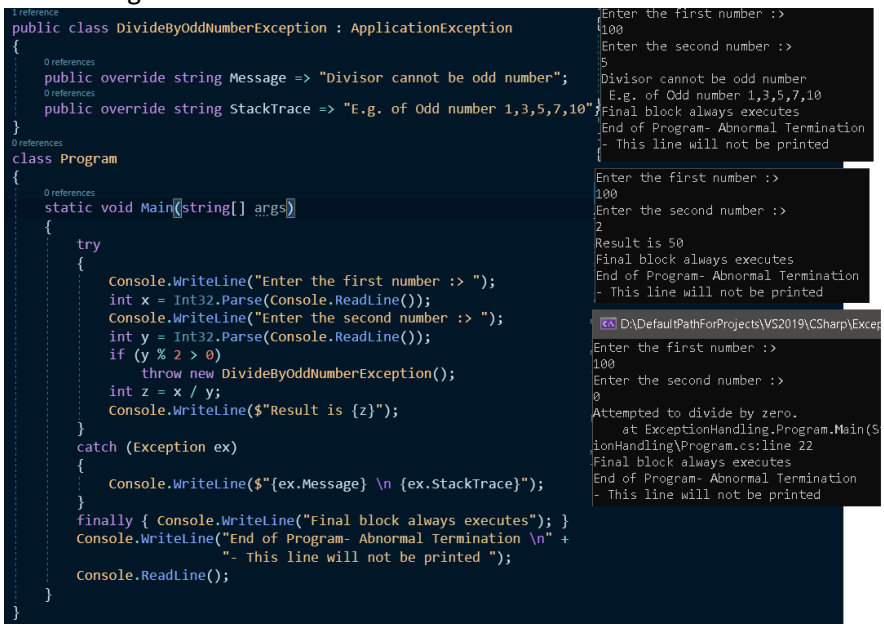
4) Application Exception E.g.



5) **Defining our own Application Exception Class :**
   a. Inherit the class from ApplicationException
   b. Override virtual read-only properties such as Message, Data, Source, StackTrace etc.

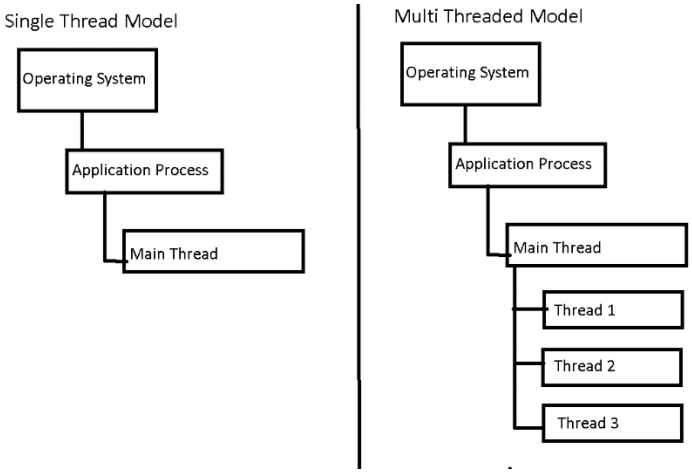
```
public class [<UserDefinedExceptionName>] : ApplicationException
{
        public override Message
        {
                get { return "<ErrorMessage>"; }
        }
}
```

6) Application Exception class E.g.



Multi-Threading

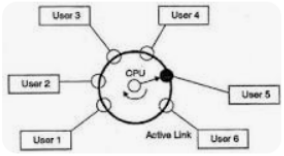
1) **Multi-Tasking** – Programs are capable for running more than one application (Task) at a time. Operating Systems are multi-tasking systems. To execute application, it uses processes internally to run this applications.
2) **Process** – is a part of the Operating System responsible for executing the program.
    a. Background Process – Services e.g. Database Engine, Required Program of OS.
3) Every Application by default contains one thread to execute the program and that is known as **Main Thread**. So every program is by default single threaded model.
4) **Single Threaded Model** – All the logic is run by one thread and it executes the logic one after the other. Drawback of Single Threaded Model
    a. Actions are completed one after the other. If one action is delayed than below/next action also needs to wait till the action above/before it is completed.
5) **Multi-Threading -** Process having multiple threads and each thread is executing simultaneously.



6) Multi-Threading works based on **Time-Sharing concept**.
    a. All the threads that created under the process will be given equal important to execute.

What is Time Sharing System explain with example?

A time sharing system **allows many users to share the computer resources simultaneously**. In other words, time sharing refers to the allocation of computer resources in time slots to several programs simultaneously. For example a mainframe computer that has many users logged on to it.



7) Advantages of Multi-Threading
    a. Simultaneous execution of logic hence time will be saved.
    b. Maximum utilization of CPU Resources.
8) How to implement Multi-Threading

```
Thread [<ThreadObjectName>] = new Thread([<MethodNameWithoutParenthesis>]);
[<ThreadObjectName>].start();
```

9) Multi-Threading E.g.

```
class Program
{
    1 reference
    public static void ThreadTest1()
    {
        for (int i = 0; i < 50; i++)
             Console.Write("  TT1 :>"+i+"\t");
    }
    1 reference
    public static void ThreadTest2()
    {
        for (int i = 0; i < 50; i++) {
            Console.Write("  TT2 :>" + i + "\t");
            if (i == 25)
            {
                Console.Write("Sleep Start \t");
                Thread.Sleep(5);
                Console.Write("Sleep End \t");
            }
        }
    }

    1 reference
    public static void ThreadTest3()
    {
        for (int i = 0; i < 50; i++)
            Console.Write("  TT3 :>" + i + "\t");
    }
    0 references
    static void Main(string[] args)
    {
        Thread thread1 = new Thread(ThreadTest1);
        Thread thread2 = new Thread(ThreadTest2);
        Thread thread3 = new Thread(ThreadTest3);
        thread1.Start(); thread2.Start(); thread3.Start();
        Console.ReadLine();
    }
}
```

TT2 Thread was Sleeping

```
TT1 => 2 2 2 2 2 2 2 2 => ThreadTest1
TT2 => 2 2         2 2 2 => ThreadTest2
TT3 => 2 2 2 2 2 2 2 2 => ThreadTest3
```

```
D:\DefaultPathForProjects\VS2019\CSharp\MultiThreadingExamples\MultiThreadingExamples\bin\Debug\netcoreapp3.1\MultiThreadingExamples.exe
 TT3 :>0         TT1 :>0         TT1 :>1         TT1 :>2         TT1 :>3         TT1 :>4         TT1 :>5         TT1 :>6
TT1 :>7         TT1 :>8         TT1 :>9         TT1 :>10        TT1 :>11        TT1 :>12        TT1 :>13        TT1 :>14
TT1 :>15        TT1 :>16        TT1 :>17        TT1 :>18        TT1 :>19        TT1 :>20        TT1 :>21        TT1 :>22
TT1 :>23        TT1 :>24        TT1 :>25        TT1 :>26        TT1 :>27        TT1 :>28        TT1 :>29        TT1 :>30
TT2 :>0         TT2 :>1         TT2 :>2         TT2 :>3         TT2 :>4         TT2 :>5         TT2 :>6         TT2 :>7
TT2 :>8         TT2 :>9         TT2 :>10        TT2 :>11        TT2 :>12        TT2 :>13        TT2 :>14        TT2 :>15
TT2 :>16        TT2 :>17        TT2 :>18        TT2 :>19        TT2 :>20        TT2 :>21        TT2 :>22        TT2 :>23
TT2 :>24        TT2 :>25      Sleep Start       TT3 :>1         TT3 :>2         TT3 :>3         TT3 :>4         TT3 :>5
TT3 :>6         TT3 :>7         TT3 :>8         TT3 :>9         TT3 :>10        TT3 :>11        TT3 :>12        TT3 :>13
TT3 :>14        TT3 :>15        TT3 :>16        TT3 :>17        TT3 :>18        TT3 :>19        TT3 :>20        TT1 :>31
TT1 :>32        TT1 :>33        TT1 :>34        TT3 :>21        TT3 :>22        TT3 :>23        TT3 :>24        TT3 :>25
TT3 :>26        TT3 :>27        TT3 :>28        TT3 :>29        TT3 :>30        TT3 :>31        TT3 :>32        TT3 :>33
TT3 :>34        TT3 :>35        TT3 :>36        TT3 :>37        TT3 :>38        TT3 :>39        TT3 :>40        TT3 :>41
TT3 :>42        TT3 :>43        TT3 :>44        TT3 :>45        TT3 :>46        TT3 :>47        TT3 :>48        TT3 :>49
TT1 :>35        TT1 :>36        TT1 :>37        TT1 :>38        TT1 :>39        TT1 :>40        TT1 :>41        TT1 :>42
TT1 :>43        TT1 :>44        TT1 :>45        TT1 :>46        TT1 :>47        TT1 :>48        TT1 :>49      Sleep End
TT2 :>26        TT2 :>27        TT2 :>28        TT2 :>29        TT2 :>30        TT2 :>31        TT2 :>32        TT2 :>33
TT2 :>34        TT2 :>35        TT2 :>36        TT2 :>37        TT2 :>38        TT2 :>39        TT2 :>40        TT2 :>41
TT2 :>42        TT2 :>43        TT2 :>44        TT2 :>45        TT2 :>46        TT2 :>47        TT2 :>48        TT2 :>49
```

10) Thread Constructors take 4 overload but we will be discussing only below two
    a. ThreadStart -
        i. Takes no input parameter and return type is void

```
0 references
class MultiThreadingClass2
{
    1 reference
    public static void ThreadTest1()
    {
        for (int i = 0; i < 50; i++)
            Console.Write("  TT1 :>" + i + "\t");
    }
    0 references
    static void Main(string[] args)
    {
        //ThreadStart is a pre-defined delegate
        //with no input parameters and void as return type
        //ThreadStart threadStart = new ThreadStart(ThreadTest1);
        //ThreadStart threadStart = ThreadTest1;
        //ThreadStart threadStart = delegate () { ThreadTest1(); };
        ThreadStart threadStart =  () => { ThreadTest1(); };

        Thread thread1 = new Thread(threadStart);
        thread1.Start();
        Console.ReadLine();
    }
}
```

    b. ParameterizedThreadStart –
        i. Takes 1 input parameter as **object** and return type is void.
        ii. In ParameterizedThreadStart, we need to pass the parameter via **Start** method.
        iii. The method should declare the input parameter as **object.**
        iv. The ParameterizedThreadStart's are not type safe.

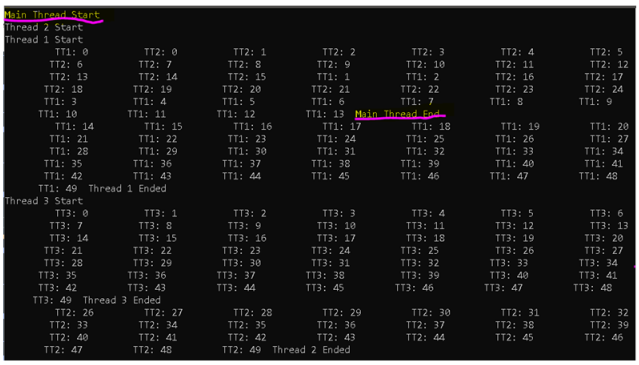
```
0 references
class MultiThreadingClass3
{
    1 reference
    public static void ThreadTest1(object maxLength)
    {
        int length = Convert.ToInt32(maxLength);
        for (int i = 0; i < length; i++)
            Console.Write("  TT3 :>" + i + "\t");
    }
    0 references
    static void Main(string[] args)
    {
        //ParameterizedThreadStart is a pre-defined delegate
        //with "1" input parameters and void as return type
        //ParameterizedThreadStart parameterizedThreadStart = new ParameterizedThreadStart(ThreadTest1);
        ParameterizedThreadStart parameterizedThreadStart = ThreadTest1;

        Thread thread1 = new Thread(parameterizedThreadStart);
        //We need to pass the parameter via Thread -> Start method
        thread1.Start(50);
        Console.ReadLine();
    }
}
```

Thread Join –

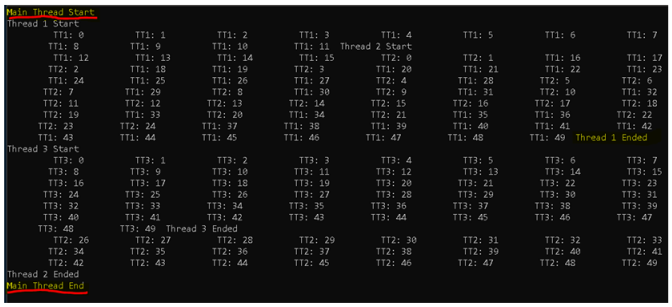
- Join – Main Thread waits until all the child thread finish's execution and then exists.
- Join([<TimePeriod>]) – Main Thread will wait until the specify Time period and then it will exit even if the child thread have not finished the execution.
- Thread Join Example

```
public class ThreadJoinExample
{
    1 reference
    public static void Thread1Test()
    {
        Console.WriteLine("Thread 1 Start");
        for (int i = 0; i < 50; i++)
            Console.Write($"\t TT1: {i}  ");
        Console.WriteLine("Thread 1 Ended");
    }
    1 reference
    public static void Thread2Test()
    {
        Console.WriteLine("Thread 2 Start");
        for (int i = 0; i < 50; i++)
            if (i == 25) { Thread.Sleep(5000);  }
            else { Console.Write($"\t TT2: {i}  "); }
        Console.WriteLine("Thread 2 Ended");
    }
    1 reference
    public static void Thread3Test()
    {
        Console.WriteLine("Thread 3 Start");
        for (int i = 0; i < 50; i++)
            Console.Write($"\t TT3: {i}  ");
        Console.WriteLine("Thread 3 Ended");
    }
    0 references
    public static void Main()
    {
        Console.WriteLine("Main Thread Start");
        Thread thread1 = new Thread(Thread1Test);
        Thread thread2 = new Thread(Thread2Test);
        Thread thread3 = new Thread(Thread3Test);
        thread1.Start(); thread2.Start(); thread3.Start();
        //Adding Join Makes sure that Main Thread will
        //Wait for child threads to complete the execution.
        thread1.Join(2000 ); thread2.Join(); thread3.Join();
        Console.WriteLine("Main Thread End");
    }
```

Without Thread.Join()-Main Thread doesn't wait for child thread 2 finish Execution.



With Thread.Join( ) - Main Thread waits for child thread to finish execution and then it ends



With Thread.Join([<Time Period>]) - Main Thread wait until the specified time period mentioned in the join parameter and it exits even if the child thread have not finished the execution.

Locking resources –

- To overcome the problem of multiple threads accessing the same resources & getting un-expected results we need to add **lock** on that resource.
- So that at a time only one thread is accessing the resource.
- How to lock the resources

```
lock(this)
{
        //Statements
}
        or
lock([<objectname>])
{
        //Statements
}
```

- Thread Locking Example

```
2 references
class ThreadLocking
{
    3 references
    public void Display()
    {
        lock (this)
        {
            Console.Write("[Hi all, I am");
            Thread.Sleep(5000);
            Console.WriteLine(" learning C Sharp.]");
        }
    }

    0 references
    public static void Main()
    {
        ThreadLocking threadLocking = new ThreadLocking();
        Thread thread1 = new Thread(threadLocking.Display);
        Thread thread2 = new Thread(threadLocking.Display);
        Thread thread3 = new Thread(threadLocking.Display);
        thread1.Start(); thread2.Start(); thread3.Start();
        Console.ReadLine();
    }
}
```

Unexcepted result occured since we had not used thread locking

```
[Hi all, I am[Hi all, I am[Hi all, I am learning C Sharp.]
learning C Sharp.]
learning C Sharp.]
```

Excepted Result occured, Since we had locking implemented. Locking ensures that at a time only one thread is accessing the resoucres and other threads will be waiting for the current executing thread to finish. so that the resouces is free to use for other threads.

```
[Hi all, I am learning C Sharp.]
[Hi all, I am learning C Sharp.]
[Hi all, I am learning C Sharp.]
```

Thread Priority

1) Thread Priority indicates how frequently the thread will gains CPU resources.
2) Thread Priority
    a. Highest – 4
    b. Above Normal – 3
    c. Normal – 2 – Default.
    d. Below Normal – 1
    e. Lowest – 0
3) By default all the thread will have Normal Priority
4) Programmer can change thread priority.
5) Operating system doesn't assign priority of threads.
6) Thread Priority Example.

```
0 references
class ThreadPriorityExample
{
    static int counter1, counter2;
    1 reference
    public static void Counter1Increment()
        { while (true) counter1++; }
    1 reference
    public static void Counter2Increment()
        { while (true) counter2++; }
    0 references
    public static void Main()
    {
        Thread counter1Thread = new Thread(Counter1Increment);
        Thread counter2Thread = new Thread(Counter2Increment);
        counter1Thread.Priority = ThreadPriority.Lowest;
        //Counter2Thread will get more CPU resources
        counter2Thread.Priority = ThreadPriority.Highest;
        counter1Thread.Start(); counter2Thread.Start();
        Console.WriteLine("Main Thread Sleep Start");
        Thread.Sleep(1000);
        Console.WriteLine("Main Thread Sleep End");
        try
        {
            //Abort has been deprected
            //Interrupt is use to terminate the thread.
            counter1Thread.Interrupt(); counter2Thread.Interrupt();
        }
        catch (ThreadInterruptedException)
        {
            counter1Thread.Join(); counter2Thread.Join();
        }
        finally{
            Console.WriteLine($"Counter1 :> {counter1}");
            Console.WriteLine($"Counter2 :> {counter2}");
            Console.WriteLine($"Differnce :> {counter2 - counter1}");
        }
        Console.ReadLine();
    }
}
```

Since the Thread Priority is set to normal by default. Both the threads will get equal access to CPU Resources depending on OS.

```
Main Thread Sleep Start
Main Thread Sleep End
Counter1 :> 172019489
Counter2 :> 193907848
Differnce :> 19905490
```

Since the Thread Priority for Counter2 was set to Highest More CPU resources were allocated to it. As you can observe that difference is huge.

```
Main Thread Sleep Start
Main Thread Sleep End
Counter1 :> 40619569
Counter2 :> 511323231
Differnce :> 471267459
```

7) Performance wise – Multi-Threaded Model is better than Single Threaded Model. In Multi-Threaded Model, Resources are shared between the threads created under main thread.

Collections

1) Collections are dynamic arrays.
2) Collection overcome the below issue Arrays have
    a. Arrays cannot be resized i.e. increasing the size after declaration.
    b. We cannot add items in the middle of the array.
    c. We cannot delete/remove items from the middle of the array.
3) Collection have feature known as Auto-Resizing, it has capability to add, delete items in the middle of the collections.
4) Collections are defined under System.Collection(Non-Generic)
    a. Stack, Heap, LinkedList, SortedList, HashTable, ArrayList…
5) Difference between Array & ArrayList

Array	ArrayList
Fixed Length	Variable Length
Not Possible to insert items in the middle	Possible to insert items in the middle
Not Possible to remove items in the middle	Possible to remove items in the middle

6) Auto-Resizing – collections increases in size whenever a new item is added but only when it doesn't have the capacity to hold it.
7) Example of Array List- Auto-Resizing, Add, Insert & Remove

```
0 references
class Program
{
    4 references
    public static void displayArrayList(ArrayList arrayList)
    {
        foreach (var item in arrayList)
            Console.Write($"{item} \t");
        Console.WriteLine();
    }
    0 references
    static void Main(string[] args)
    {
        //ArrayList Declared with with initial capacity as 2
        ArrayList arrayList = new ArrayList(2);
        Console.WriteLine("Inital Capacity "+arrayList.Capacity);
        arrayList.Add(100); arrayList.Add(200); arrayList.Add(300);
        Console.WriteLine("After adding 3 item. Capacity " + arrayList.Capacity);
        arrayList.Add(400); arrayList.Add(500); arrayList.Add(600);
        arrayList.Add(700);
        Console.WriteLine("After adding 7 item. Capacity " + arrayList.Capacity);
        arrayList.Add(800); arrayList.Add(900);
        Console.WriteLine("After adding 9 item. Capacity " + arrayList.Capacity);

        displayArrayList(arrayList);
        //Inserting items in the middle of ArrayList
        arrayList.Insert(5, 550);
        displayArrayList(arrayList);
        //Remove/Deleting items from the middle of the ArrayList
        arrayList.Remove(400);
        displayArrayList(arrayList);
        arrayList.RemoveAt(6);
        displayArrayList(arrayList);
        Console.ReadLine();
    }
```

```
Inital Capacity 2
After adding 3 item. Capacity 4
After adding 7 item. Capacity 8
After adding 9 item. Capacity 16
100     200     300     400     500     600     700     800     900
100     200     300     400     500     550     600     700     800     900
100     200     300     500     550     600     700     800     900
100     200     300     500     550     600     800     900
```

8) Difference between Hashtable & (Array & ArrayList)
    a. Array & Array List Key/Value combination where keys are pre-defined i.e. index
        i. It is hard to remember the index value for retrieving particular data. Array[0], Array[1] …. Array[n] – N is integer value.
    b. Hashtable stores the value and key is user-defined. Hashtable[“Chandrahas”], Hashtable[“PPP”] etc..
9) In Hashtable, keys can be double, integer or strings
10) How to define and add data in hashtable & also How to access value from hashtable

```
Hashtable [<hashtableObjectName>] = new Hashtable();
[<hashtableObjectName>].Add("KeyName1","Value1");
[<hashtableObjectName>].Add("KeyName2","Value2");
//Retrieving
Console.Write([<hashtableObjectName>][<KeyName2>]);
```

11) Hashtable’s internally generates hash code for key values. It fetches data based on the hashcode value of the key. Due to this reason Hashtable’s are more efficient as compare to ArrayList.
12) Example of Hashtable

```
0 references
public class HashTableExample
{
    3 references
    public static void Display(Hashtable hashtable)
    {
        //Keys are stored in hashcode for retrieving data faster
        foreach (var key in hashtable.Keys)
            Console.WriteLine($"{key} : {hashtable[key]} - " +
                $"Hashcode : {key.GetHashCode()}");
        Console.WriteLine("---------------------------------");
    }
    0 references
    public static void Main()
    {
        //Key can be numeric, alphanumeric in hashtable
        Hashtable hashtableNumericKey = new Hashtable();
        hashtableNumericKey.Add(0,"Test Data");
        hashtableNumericKey.Add(1, "Test Data 2");
        Display(hashtableNumericKey);

        Hashtable htEmployee = new Hashtable();
        htEmployee.Add("EmpId",1860);
        htEmployee.Add("EmpName", "Chandrahas Poojari");
        htEmployee.Add("Job", "Sr. Software Dev.");
        htEmployee.Add("Salary", 100000000000);
        htEmployee.Add("Phone", 9858528525);
        htEmployee.Add("MrgId", 1000);
        Display(htEmployee);

        //Removing the key from hashtable
        htEmployee.Remove("Phone");
        Display(htEmployee);
        //Accessing values via key
        Console.WriteLine($"Emp Name : {htEmployee["EmpName"]} \n" +
                          $"Emp Job  : {htEmployee["EmpName"]} \n" +
                          $"Salary   : {htEmployee["Salary"]}");

    }
}
```

```
1 : Test Data 2 - Hashcode : 1
0 : Test Data - Hashcode : 0
---------------------------------
MrgId : 1000 - Hashcode : -1190594264
EmpName : Chandrahas Poojari - Hashcode : 765479156
EmpId : 1860 - Hashcode : -408625228
Phone : 9858528525 - Hashcode : -1761698529
Salary : 100000000000 - Hashcode : -1712704768
Job : Sr. Software Dev. - Hashcode : 1453317455
---------------------------------
MrgId : 1000 - Hashcode : -1190594264
EmpName : Chandrahas Poojari - Hashcode : 765479156
EmpId : 1860 - Hashcode : -408625228
Salary : 100000000000 - Hashcode : -1712704768
Job : Sr. Software Dev. - Hashcode : 1453317455
---------------------------------
Emp Name : Chandrahas Poojari
Emp Job  : Chandrahas Poojari
Salary   : 100000000000
```

Collections - Generic

1) Non-Generic Collections are not type safe.
2) Generic collections are combination of
    a. Arrays – Type Safe but doesn't have Fixed Length.
    b. Collections – Auto Resizing but not type safe.
    c. In C# 2.0, Generic Collections were introduced under "System.Collection.Generic" to overcome the issues such as type safe collection

.Net Collection	Generic Collection
Array list	List (Generic)
Hash table	Dictionary
Stack Stack	Generics
Queue	Queues Generics

c-sharpcorner

3) Advantages of Generic Collection.
    a. Type Safe
    b. Auto-Resize
4) How to define List<T>

```
List<T> where T is the type
        i.e. Pre-Define Types or User Define Type

Pre-Define Type Examples
        List<int> lstOfIntergers = new List<int>();
        List<string> lstOfStrings = new List<string>();

User Define Type Examples
        public class Customer
        {
                int CustomerId {get;set;}
                string CustomerName {get; set;}
        }

        List<Customer> lstOfCustomers = new List<Customer();
```

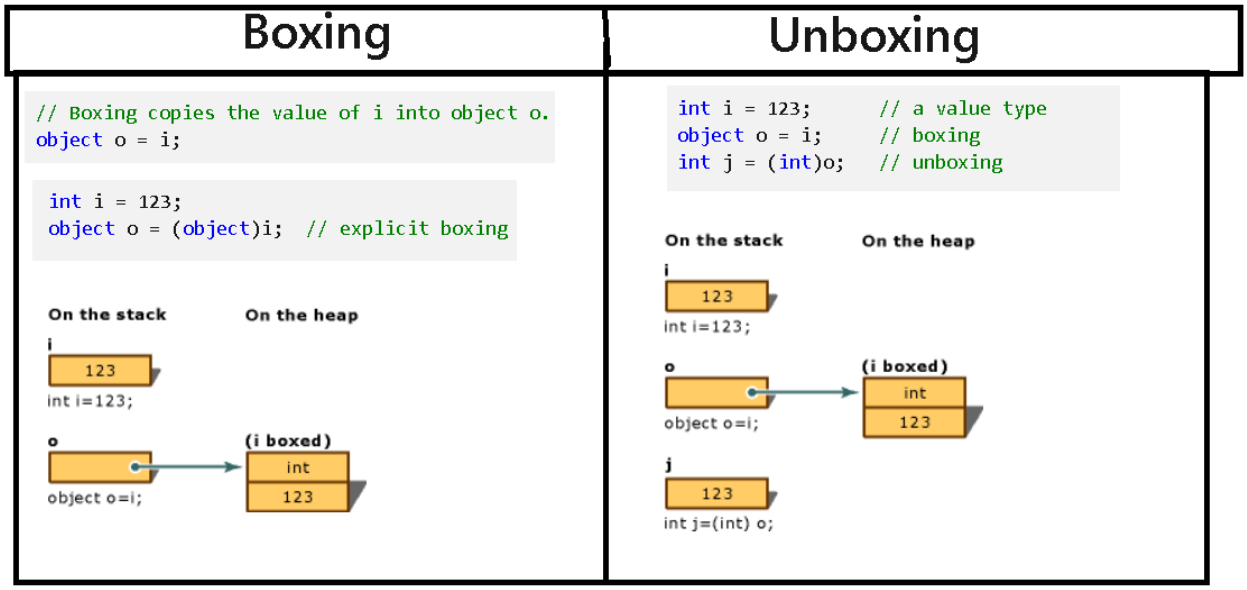
5) Example of List<T>

```
0 references
public class GenericListExample
{
    3 references
    public static void Display(List<int> lsts)
    {
        foreach (var item in lsts)
            Console.Write(item + " ");
        Console.WriteLine();
    }
    0 references
    public static void Main()
    {
        int[] int_arr = { 30, 40, 50, 60 };
        List<int> lstOfIntergers = new List<int>();
        lstOfIntergers.Add(10); lstOfIntergers.Add(20);
        lstOfIntergers.AddRange(int_arr);

        for (int i = 0; i < lstOfIntergers.Count; i++)
            Console.Write(lstOfIntergers[i] + " ");
        Console.WriteLine();

        lstOfIntergers.Insert(4, 55);
        Display(lstOfIntergers);

        lstOfIntergers.RemoveAt(1);
        Display(lstOfIntergers);

        lstOfIntergers.Remove(10);
        Display(lstOfIntergers);
        Console.ReadLine();
    }
}
```

```
10 20 30 40 50 60
10 20 30 40 55 50 60
10 30 40 55 50 60
30 40 55 50 60
```

Generics

1) Generics – C# 2.0
    a. We can create strongly Type entities.
    b. Improves performance, since we can avoid boxing and unboxing of the object.
    c. Less Coding
2) Boxing & Unboxing – performance issues are created if not used efficiently.
    a. Boxing – is an implicit conversion of value type to the object type. Boxing a value type allocates an object instance on the heap memory and copies the value into the new object.
    b. Unboxing - is an explicit conversion of object type to value type.



3) How to define Generic Method

```
public class [ClassName]
{
        //T - PlaceHolder
        //You specify int than T will be float
        public static void [MethodName]<T>(T a, T b)
        {

        }

        public static void Main()
        {
                [ClassName] [ClassNameObjectName]
                                = new [ClassName]();
                [ClassNameObjectName].[MethodName]<float>(10.25f,25.52f);
        }
}
```

4) Example of Non Generic & Generic Code

```
2 references
class NonGenericCompare
{
    3 references
    public bool Compare(object a, object b)
    {
        if (a.Equals(b))
            return true;
        return false;
    }

    0 references
    public static void Main()
    {
        NonGenericCompare nonGenericCompare = new NonGenericCompare();
        bool result = nonGenericCompare.Compare(10, 50);
        Console.WriteLine("Result :> "+result);

        result = nonGenericCompare.Compare(10f, 10f);
        Console.WriteLine("Result :> " + result);

        // I wanted to compare float & float
        // but it works since it is not type safe
        result = nonGenericCompare.Compare(10.25f, 25.50);
        Console.WriteLine("Result :> " + result);
        Console.ReadLine();
    }
}
```

```
2 references
class GenericCompare
{
    2 references
    public bool Compare<T>(T a, T b)
    {
        if (a.Equals(b))
            return true;
        return false;
    }

    0 references
    public static void Main()
    {
        GenericCompare genericCompare = new GenericCompare();
        bool result = genericCompare.Compare<int>(10, 50);
        Console.WriteLine("Result :> " + result);

        result = genericCompare.Compare<string>("a", "a");
        Console.WriteLine("Result :> " + result);

        // Below code gives me compile time error
        // Since we have specfied that we need to
        // compare float values it is not allowring
        // double value

        //result = genericCompare.Compare<float>(10.25f, 50.25);
        //Console.WriteLine("Result :> " + result);
        Console.ReadLine();
    }
}
```

5) Dynamic Vs var

Var	Dynamic
It was introduced in C# 3.0	It was introduced in C# 4.0
The variable are declared using “var” keyword are statically typed.	The variable are declared using “dynamic” Keyword are dynamically typed.
The type of the variable is decided at compile time.	The type of the variable is decided at run time.
If the variables does not initialized it throws error.	If the variables does not initialized it will not throw an error.
It cannot be used for Properties or Returning values from the method. It can be used only as local variable in the method.	It can be used for properties or returning values from the method.

6) Example of dynamic, Generic Class & Method

```
2 references
public class GenericCalculator
{
    2 references
    public T Add<T>(T a, T b)
    {
        dynamic a1 = a; dynamic b1 = b;
        return a1 + b1;
    }

    1 reference
    public T Sub<T>(T a, T b)
    {
        dynamic a1 = a; dynamic b1 = b;
        return a1 - b1;
    }
}

0 references
public class GenericCalculatorMethod
{
    0 references
    public static void Main()
    {
        GenericCalculator genericCalculator = new GenericCalculator();
        Console.WriteLine($"Generic Method: Add :> " +
            $"{genericCalculator.Add<float>(10f, 10f)}");
        Console.WriteLine($"Generic Method: Sub :>" +
            $" {genericCalculator.Sub<int>(20, 10)}");
        Console.WriteLine($"Generic Method: Add :> " +
            $"{genericCalculator.Add<string>("Hello ", "CJ")}");
    }
}
```

```
Generic Method: Add :> 20
Generic Method: Sub :> 10
Generic Method: Add :> Hello CJ
```

```
4 references
public class GenericCalculatorClass<T>
{
    2 references
    public T Add(T a, T b)
    {
        dynamic a1 = a; dynamic b1 = b;
        return a1 + b1;
    }

    2 references
    public T Sub(T a, T b)
    {
        dynamic a1 = a; dynamic b1 = b;
        return a1 - b1;
    }
}
0 references
public class TestGenericClass
{
    0 references
    public static void Main()
    {
        GenericCalculatorClass<float> genericCalculatorClassf
                                        = new GenericCalculatorClass<float>();
        Console.WriteLine($"Generic Class: Add float :> " +
                                        $"{genericCalculatorClassf.Add(10.25f, 6.5f)}");
        Console.WriteLine($"Generic Class: Sub float :>" +
                                        $" {genericCalculatorClassf.Sub(5.5f, 6.5f)}");

        GenericCalculatorClass<int> genericCalculatorClassi
                                        = new GenericCalculatorClass<int>();
        Console.WriteLine($"Generic Class: Add int :> " +
                                        $"{genericCalculatorClassi.Add(10, 10)}");
        Console.WriteLine($"Generic Class: Sub int :>" +
                                        $" {genericCalculatorClassi.Sub(20, 10)}");
    }
}
```

```
Generic Class: Add float :> 16.75
Generic Class: Sub float :> -1
Generic Class: Add int :> 20
Generic Class: Sub int :> 10
```

7) In case of generic collections the type of values, we want to store under the collections need not to be pre-defined typed only i.e. float, int, double, string etc. but it can be also some user defined type also.
8) Example of User-Defined Type for Generic collections

```
8 references
public class Character
{
    5 references
    public int CharacterId { get; set; }
    5 references
    public string CharacterName { get; set; }
    5 references
    public double PowerLevel { get; set; }
    5 references
    public string City { get; set; }
    1 reference
    public override string ToString()
    {
        return $"{nameof(CharacterId)} : {CharacterId} \n" +
               $"{nameof(CharacterName)} : {CharacterName} \n" +
               $"{nameof(City)} : {City} \n" +
               $"{nameof(PowerLevel)} : {PowerLevel} \n";
    }
}
0 references
public class GenericUserDefineTypeExample
{
    0 references
    public static void Main()
    {
        //User Defined Type - Character is used with Generic Collections
        List<Character> characters = new List<Character>();
        Character character1 = new Character()
        {
            CharacterId = 101, CharacterName = "Vegata",
            City = "Delhi", PowerLevel = 250000.0
        };
        Character character2 = new Character()
        {
            CharacterId = 102, CharacterName = "Goku",
            City = "Mumbai",PowerLevel = 500000.0
        };
        characters.Add(new Character()
        {
            CharacterId = 103,CharacterName = "Beerus",
            City = "Chennai",  PowerLevel = 5000000.0
        });
        characters.Add(character1);
        characters.Add(character2);
        foreach (Character character in characters)
            Console.WriteLine(character.ToString());

    }
}
```

```
CharacterId : 103
CharacterName : Beerus
City : Chennai
PowerLevel : 5000000

CharacterId : 101
CharacterName : Vegata
City : Delhi
PowerLevel : 250000

CharacterId : 102
CharacterName : Goku
City : Mumbai
PowerLevel : 500000
```

Reference

- https://www.geeksforgeeks.org/difference-between-var-and-dynamic-in-c-sharp/
- MS docs

Generics – Constraints

1) Constraints are used in Generic to restrict the types that can be substituted for type parameters
2) When constraints are applied on the generic method or class, the substituted type parameter needs to comply with the constraint defined. If it doesn't follow then compile time error occurs.
3) How to define Constraints

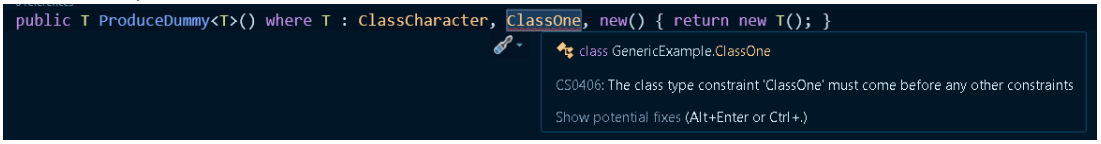
```
public class TestGenericConstraints
{
        public void Method1<SubstitutedP>(SubstitutedP a) where SubstitutedP : class
        {
                // This means Substituted Type Parameter should be
                // of class type i.e. reference type
        }
        public void Method2<SubstitutedP>(SubstitutedP a) where SubstitutedP : struct
        {
                // This means Substituted Type Parameter should be
                // of struct type i.e. value type
        }
        public void Method3<SubstitutedP>(SubstitutedP a) where SubstitutedP : new()
        {
                // This means Substituted Type Parameter should be
                // a reference type and also it should contain a
                // parameterless constructor
        }
        public void Method3<SubstitutedP>(SubstitutedP a)
                                        where SubstitutedP : IInterfaceName
        {
                // This means Substituted Type Parameter should be
                // a reference type and also it should implement the
                // Interface specified.
        }
        public void Method3<SubstitutedP>(SubstitutedP a)
                                        where SubstitutedP : ClassName, new()
        {
                // This means Substituted Type Parameter should be
                // a reference type and also it should inherit the class
                // specified & also it should contain a parameterless
                // constructor
        }
}
```

4) Example of Constraints…

```
2 references
public interface IDummyInterface { }
2 references
public class ClassCharacter { }
4 references
public class ClassOne { public ClassOne() { } }
3 references
public class ClassTwo : ClassCharacter { public ClassTwo() { } }
3 references
public class ClassThree : IDummyInterface { public ClassThree() { } }
2 references
class GenericConstraintExample
{
    1 reference
    public static T ProduceObjectForClass<T>()  where T : class, new() { return new T(); }
    1 reference
    public T ProduceOForInheritC<T>() where T : ClassCharacter, new() { return new T(); }
    1 reference
    public T ProduceObjectForInterFace<T>() where T : IDummyInterface, new() { return new T(); }
    1 reference
    public T Add<T>(T a, T b) where T:struct{ dynamic d1 = a; dynamic d2 = b;  return d1 + d2; }
    1 reference
    public static T ProducDefault<T>() where T : class { T returnVal = default(T); return returnVal; }
    0 references
    public static void Main()
    {
        GenericConstraintExample genericConstraintExample = new GenericConstraintExample();
        //Since Class Two is inherited with ClassCharacter & also has a parameterless constructor
        ClassTwo classTwo = genericConstraintExample.ProduceOForInheritC<ClassTwo>();
        Console.WriteLine("Object Created :> " +classTwo);
        //Since ClassThree has implemented IDummyInterface & contains parameterless contructor
        ClassThree classThree = genericConstraintExample.ProduceObjectForInterFace<ClassThree>();
        Console.WriteLine("Object Created :> " + classThree);
        //Since ClassOne is User Defined Type i.e. class & contains Parameterless constructor
        ClassOne classOne = ProduceObjectForClass<ClassOne>();
        Console.WriteLine("Object Created :> " + classOne);
        //Since we will be using value type float it will comply with the constraint.
        Console.WriteLine("Add :> float value :> "+genericConstraintExample.Add<float>(5.25f,1.25f));
        Console.WriteLine("Object will not be created."+ ProducDefault<ClassOne>());
    }
    //public T ProduceDummy<T>() where T : ClassCharacter, ClassOne, new() { return new T(); }
}
```

```
Object Created :> GenericExample.ClassTwo
Object Created :> GenericExample.ClassThree
Object Created :> GenericExample.ClassOne
Add :> float value :> 6.5
Object will not be created.
```

5) We can have only one primary constraint as a Base Class. i.e. to say the below code is illegal. Since multiple inheritance is not allowed in C#.



6) We can use default keyword for class & struct type
    a. Class type will provide null
    b. Struct type will provide 0

References

- https://www.youtube.com/watch?v=oitvN9YC_PU
- https://www.youtube.com/watch?v=PF_LPi_9lqo
- https://www.youtube.com/watch?v=aGvmqtfFjOw
- https://www.c-sharpcorner.com/UploadFile/ashish_2008/constraints-in-generics/

Generic Collections – IComparable<T> & IComparer<T>

1) For Sorting a primitive datatype such as int, float, double etc., we can directly use "**List.Sort()**".

```
0 references
static void Main(string[] args)
{
    List<int> lstData = new List<int>() { 50,10,5,45,8,100,2,1 };
    lstData.Sort();
    foreach (var item in lstData)
        Console.Write($"{item}\t");
}
```

```
1       2       5       8       10      45      50      100
```

2) For Sorting, user defined data types "**Sort()**" will give us a run time error if programmer have not implemented below interfaces
    a. IComparable<T> - When the class is mutable(can be changed)
        i. We use IComparable<T>, when we have the source code of the class to which it will be implemented.
        ii. It implements CompareTo(T name) method where T is class which needs to be compared with the current class "this".
    b. IComparer<T> - - When the class is immutable(cannot be changed)
        i. We use IComparer<T>, when we don't have the source code of the class to which it will be implemented.
        ii. It implements Compare(T a, T b) method where both the classes will be compared.
    c. For both return type is integer

```
Greater Than - > -  1
Less Than    - < - -1
Equal        - = -  0
```

3) Example for IComparable & IComparer.
    a. Part 1 – Implementation of IComparable & IComparer.

```
33 references
public class Character : IComparable<Character>
{
    16 references
    public int CharacterId { get; set; }
    12 references
    public string CharacterName { get; set; }
    16 references
    public long PowerLevel { get; set; }
    0 references
    public int CompareTo([AllowNull] Character other)
    {
        //Greater Than
        if (this.CharacterId > other.CharacterId)
            return 1;
        //Else Than
        else if (this.CharacterId < other.CharacterId)
            return -1;
        //Equal To
        else
            return 0;
    }
    0 references
    public override string ToString()
    {
        return $"{nameof(CharacterId)} - {CharacterId} \t" +
               $"{nameof(CharacterName)} - {CharacterName} \t" +
               $"{nameof(PowerLevel)} - {PowerLevel}\t \n";
    }
}
```

```
// Assuming Character class is immutable
// cannot be changed & Hence we used IComparer
// for comparing complex data
2 references
public class GenericComparerExample : IComparer<Character>
{
    0 references
    public int Compare([AllowNull] Character character1,
                       [AllowNull] Character character2)
    {
        //Greater Than
        if (character1.PowerLevel > character2.PowerLevel)
            return 1;
        //Else Than
        else if (character1.PowerLevel < character2.PowerLevel)
            return -1;
        //Equal To
        else
            return 0;
    }
}
```

b. Part 2 – Calling Sort on List Object.

```
0 references
public class GenericComparableExample
{
    2 references
    public static void DisplayData<T>(List<T> lst) where T :  class
    {
        foreach (var lstValue in lst)
            Console.Write(lstValue.ToString());
        Console.WriteLine("----------------");
    }
    0 references
    public static void Main()
    {
       Character character1 = new Character
            { CharacterId = 120, CharacterName = "SuperMan",
            PowerLevel = 200000 };
        Character character3 = new Character
            { CharacterId = 109, CharacterName = "Goku",
            PowerLevel = 10000000 };
        Character character2 = new Character
            { CharacterId = 150, CharacterName = "Vegata",
            PowerLevel = 200000 };
        Character character5 = new Character
            { CharacterId = 100, CharacterName = "One Punch Man",
            PowerLevel =20100};
        Character character10 = new Character
            { CharacterId = 1, CharacterName = "Batman",
            PowerLevel = 9999 };
        List<Character> characters = new List<Character>
            { character1, character5, character2, character10, character3 };
        DisplayData<Character>(characters);
        characters.Sort();
        DisplayData<Character>(characters);
    }
}
```

```
0 references
public class TestGenericComparerExample
{
    2 references
    public static void DisplayData<T>(List<T> lst) where T : class
    {
        foreach (var lstValue in lst)
            Console.Write(lstValue.ToString());
        Console.WriteLine("----------------");
    }
    0 references
    public static void Main()
    {
       Character character1 = new Character
            { CharacterId = 120, CharacterName = "SuperMan",
            PowerLevel = 200000 };
        Character character3 = new Character
            { CharacterId = 109, CharacterName = "Goku",
            PowerLevel = 10000000 };
        Character character2 = new Character
            { CharacterId = 150, CharacterName = "Vegata",
            PowerLevel = 200000 };
        Character character5 = new Character
            { CharacterId = 100, CharacterName = "One Punch Man",
            PowerLevel =20100};
        Character character10 = new Character
            { CharacterId = 1, CharacterName = "Batman",
            PowerLevel = 9999 };
        List<Character> characters = new List<Character>
            { character1, character5, character2, character10, character3 };
        DisplayData<Character>(characters);
        //Create Object of Class which has implemented the IComparer
        GenericComparerExample genericComparerExample = new GenericComparerExample();
        //Provide the object to Sort method.
        characters.Sort(genericComparerExample);
        DisplayData<Character>(characters);
    }
}
```

c. Part 3 Output

ICompareable- Sorted via "CharacterId"

```
CharacterId - 120       CharacterName - SuperMan        PowerLevel - 200000
CharacterId - 100       CharacterName - One Punch Man   PowerLevel - 20100
CharacterId - 150       CharacterName - Vegata  PowerLevel - 200000
CharacterId - 1         CharacterName - Batman  PowerLevel - 9999
CharacterId - 109       CharacterName - Goku    PowerLevel - 10000000
----------------
CharacterId - 1         CharacterName - Batman  PowerLevel - 9999
CharacterId - 100       CharacterName - One Punch Man   PowerLevel - 20100
CharacterId - 109       CharacterName - Goku    PowerLevel - 10000000
CharacterId - 120       CharacterName - SuperMan        PowerLevel - 200000
CharacterId - 150       CharacterName - Vegata  PowerLevel - 200000
----------------
```

IComparer - Sorted via "PowerLevel"

```
CharacterId - 120       CharacterName - SuperMan        PowerLevel - 200000
CharacterId - 100       CharacterName - One Punch Man   PowerLevel - 20100
CharacterId - 150       CharacterName - Vegata  PowerLevel - 200000
CharacterId - 1         CharacterName - Batman  PowerLevel - 9999
CharacterId - 109       CharacterName - Goku    PowerLevel - 10000000
----------------
CharacterId - 1         CharacterName - Batman  PowerLevel - 9999
CharacterId - 100       CharacterName - One Punch Man   PowerLevel - 20100
CharacterId - 120       CharacterName - SuperMan        PowerLevel - 200000
CharacterId - 150       CharacterName - Vegata  PowerLevel - 200000
CharacterId - 109       CharacterName - Goku    PowerLevel - 10000000
----------------
```

Collections – IEnumerable or IEnumerable<T>

1) IEnumerable is an interface that is implemented by all the collection classes and because they implement this IEnumerable interface every collection class contains a method called as GetEnumerator and because of that method we can use foreach loop on that collection.

```
IEnumerable
        -ICollection
                -IList - Index based.
                -IDictionary - Key, Value Combination
```

2) If we need to define our own class to behave like collection then we need to implement IEnumerable interface and write the implement GetEnumerator method.
3) How to make you class behave like collections

```
public class [<EntityClassName>] { public int Id { get; set; }  }
public class [<EnumeratorClassName>] : IEnumerator
{
        [<MyClassnameObject>] objectName; int currentIndex;
        public [<EnumeratorClassName>]([<MyClassnameObject>] objectName)
        {
                this.objectName = objectName;
                //Current index starts from -1
                currentIndex = -1;
        }
        public object Current { return  [<MyClassnameObject>][currentIndex]; }
        public bool MoveNext()
        {
                //Return true if data is present.
                //Return false if data is not present.
        }
         public void Reset() { }
}
public class [<MyClassname>] : IEnumerable
{
        List<[<EntityClassName>]> [<ListOfEntityName>]
                                = new List<[<EntityClassName>]>();
        public [<EntityClassName>] this[int index] //Indexer
        {
                get { return [<ListOfEntityName>][index]; }
        }
        public void Add([<EntityClassName>] EntityClassName)
        {
                [<EntityClassNameObject>].Add(EntityClassName);
        }
        public IEnumerator GetEnumerator()
        {
                return new [<EnumeratorClassName>](this);
        }
}
```

4) Example of IEnumerable

```
2 references
public class MetaVerseEnumerator : IEnumerator
{
    private MetaVerse metaVerse;
    private int currentIndex;
    private Character currentCharacter;
    1 reference
    public MetaVerseEnumerator(MetaVerse metaVerse)
    {
        this.metaVerse = metaVerse;
        currentIndex = -1;
    }

    0 references
    public object Current => currentCharacter;
    0 references
    public bool MoveNext()
    {
        if (++currentIndex >= metaVerse.Count) return false;
        else currentCharacter = metaVerse[currentIndex];
        return true;
    }
    0 references
    public void Reset() { }
}
```

```
3 references
public class MetaVerse : IEnumerable
{
    List<Character> characters = new List<Character>();
    1 reference
    public int Count { get { return characters.Count; } }
    1 reference
    public Character this[int index]
    {
        get { return characters[index]; }
    }
    0 references
    public void Add(Character character)
    {
        characters.Add(character);
    }
    0 references
    public void Remove(Character character)
    {
        characters.Remove(character);
    }
    0 references
    public IEnumerator GetEnumerator()
    {
        return new MetaVerseEnumerator(this);
    }
}
```

```
0 references
public class TestMetaVerse
{
    1 reference
    public static void DisplayData<T>(T lst) where T : class, IEnumerable
    {
        foreach (var lstValue in lst)
            Console.Write(lstValue.ToString());
        Console.WriteLine("----------------");
    }
    0 references
    public static void Main()
    {
        Character character1 = new Character
            { CharacterId = 120, CharacterName = "SuperMan",
            PowerLevel = 200000 };
        Character character3 = new Character
            { CharacterId = 109, CharacterName = "Goku",
            PowerLevel = 10000000 };
        Character character2 = new Character
            { CharacterId = 150, CharacterName = "Vegata",
            PowerLevel = 200000 };
        Character character5 = new Character
            { CharacterId = 100, CharacterName = "One Punch Man",
            PowerLevel =20100};
        Character character10 = new Character
            { CharacterId = 1, CharacterName = "Batman",
            PowerLevel = 9999 };
        MetaVerse metaVerse = new MetaVerse()
        { character1, character5, character2, character10, character3 };
        DisplayData<MetaVerse>(metaVerse);
    }
}
```

```
CharacterId - 120       CharacterName - SuperMan        PowerLevel - 200000
CharacterId - 100       CharacterName - One Punch Man   PowerLevel - 20100
CharacterId - 150       CharacterName - Vegata  PowerLevel - 200000
CharacterId - 1         CharacterName - Batman  PowerLevel - 9999
CharacterId - 109       CharacterName - Goku    PowerLevel - 10000000
----------------
```

LINQ – Language integrated Query

1) Using Linq, we can write queries on a wide variety of data sources
    a. Arrays
    b. Collections
    c. Database Tables
    d. Datasets
    e. Xml Data
2) Syntax SQL Vs Linq

```
Basic SQL Syntax
Select <ColumnList> from <TableName> [as <alias>] [<clauses>]

                              VS
Linq Syntax
from <alias> in <collection | arrays | datatables>
[<clauses>] select <alias>
```

3) Example of Linq

```
0 references
static void Main(string[] args)
{
    int[] arrayData = { 12, 10, 5, 48, 9, 80, 39, 85, 11, 16,
                        75, 88, 74, 55, 9, 1, -1, 5 };

    var getValueGreaterThan40 = from arrayValue in arrayData  //<alias> in <collection>
                                where arrayValue > 40         //<clauses>
                                orderby arrayValue descending //<clauses>
                                select arrayValue;            //select <alias>
    foreach (var item in getValueGreaterThan40)
        Console.Write(item +"\t");

}
```

4) Linq
    a. Linq to Objects – Arrays, Collections etc.
    b. Linq to Database– DataTables & Relational Database Tables
        i. Linq to ADO.NET
        ii. Linq to SQL – SQL Servers
        iii. Linq to Entities – SQL Servers, Oracles & SqlLite etc.
    c. Linq to Xml – Xml files
5) Linq to SQL - It is a query language that is introduced in .Net 3.5. Framework for working with relational database i.e. SQL Server.
6) Linq to SQL is not only about querying the data but also allows us to perform CRUD operations. CRUD stands for
    a. C – Create (insert)
    b. R - Read (select)
    c. U- Update
    d. D – Delete

    Note: we can also call store procedure by using Linq to SQL.

7) SQL => SQL Server Vs LINQ => SQL Server

SQL => SQL Server	LINQ => SQL Server
Runtime Syntax checking of SQL statements. i.e. extra load on server if syntax is not correct.	Compile-Time Syntax checking is performed by C# compiler itself using query engine under .NET framework.
Not Type Safe.	Type Safe.
No Intellisense Support.	Intellisense support is available.
Debugging of SQL statement is not possible	Debbuging of Linq SQL is possible.
Code is combination of Object Oriented and Relational.	Code is purely Object Oriented Code.

8) Remember

```
Table                   =>          Class
  - Columns             =>            - Property
  - Rows Or Records     =>            - Instance
Stored Procedures       =>          Methods
```

9) Sql Vs Linq Statements

```
select * | <collist> from <table> as <alias> [<clauses>]
                          VS
from <alias> in <table> [<clauses>] select <alias> | new { <collist>};
```

10) Clauses in SQL Vs Clauses in Linq.

Clauses in SQL	Clauses in Linq.
Where	Where
Group By	Group By
Order By	Order by
Having	Where clause if used after Group Clause then it acts like Having Clauses

11) Order By – Default ordering is ascending.

```
Default ordering is ascending

from employee in datacontext.Employees
orderby employee.Job
select employee

from employee in datacontext.Employees
orderby employee.Job descending
select employee
```

12) Where Clause

```
from employee in datacontext.Employees
where employee.Job == "Salesman"
select employee;
```

13) Group By Clause –
    a. Data is grouped into two part key part and result part
        i. Group Key – The key on which grouping was done
        ii. Group Result – where aggregate functions will be called i.e. Count, Avg, Max, Min, Sum

```
Group By in SQL
select DeptNo, Count(*) as 'Member Count' from Employee
Group By DeptNo

Group By in Linq
from employee in datacontext.Employees
group employee by employee.DeptNo into group
select new
        {
                GroupKey = group.Key
                GroupResult = group.Count()
        };
```

14) Group by Clause with Having clause.
    a. Linq does have "**Having**" clause, programmer need use "where" clause to achieve the result.
    b. **Where** clause used **before Group** clause will **behave** like normal "**where**" clause.
    c. **Where** clause used **after Group** clause will **behave** like "**having**" clause.

```
Group By -Having in Sql
select Job, Count(*) as 'Member Count' from Employee
group by Job Having Count(*) > 5

Group By - Having(Where) in Linq
from employee in datacontext.Employees
group employee by employee.Job into group
// Since we are using it after group clause
// It will behave like having clause.
where G.Count() > 5
select new
        {
                GroupKey = group.Key
                GroupResult = group.Count()
        };
```

15) Multiple Clauses

```
Multiple Clause in SQL
select Deptno, Count(*) as 'Clerk Count' from Employee
where Job = 'Clerk'
group by Deptno
Having Count(*) > 1
order by Deptno desc

Multiple Clause in Linq
from employee in datacontext.Employees
where employee.Job = 'Clerk'
group employee by employee.DeptNo into group
where group.Count() > 1
order by group.Key descending
select new
        {
                GroupKey = group.Key
                GroupResult = group.Count()
        };
```

16) In Linq – when multiple clauses are used. Clause needs to follow the Sequence as shown below
    i. Where clause – first where clause will execute sends the data to group by clause
    ii. Group by clause – will group the data and will send the data to “having” clause
    iii. Having (where) clause – will apply filters on the grouped data and remove unwanted data.
    iv. Order by clause – finally order by clause will arrange the data and display the data.

Repositories for Code :

- Github - ChandrahasJ

Reference & Thanks to:

- Naresh IT YouTube Channel – C# Playlist
- Microsoft Documentation.
- Jamie Kings YouTube Channel.
- Websites - C-Sharp Corner & Many more

Thank you