C# Tutorial for Beginners

Using System declaration

The namespace declaration, **using System**, indicates that the System namespace is being used.

A **namespace** is used to organize the code and is collection of classes. interfaces, structs, enums and delegates. A namespace can also contain another namespaces.

Main method is the entry point into the application.

Console.WriteLine examples

```
Console.WriteLine("Your Name is " +x);
```
Or
```
Console.WriteLine("Your Name is {0}",x);
```

Datatypes

https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/integral-numeric-types

string is a class, reference type

Escape Sequences

To print a symbol

e.g.:

```
string name = "\"Swapnadip\"";
string path= "D:\\!Desktop\\DATA FIX\\Appointment\\05042019";
Console.WriteLine(" {0}", name, path);
```

Here \” is the escape symbol for “ and \\ is for \

https://learn.microsoft.com/en-us/cpp/c-language/escape-sequences?view=msvc-170

Verbatim

Verbatim literal, is a string with an @ symbol prefix, as in @"Hello". Verbatim literals make escape sequences translate as normal printable characters to enhance readability.

Practical Example: Without Verbatim Literal: "C:\\Pragim\\DotNet\\Training\\Csharp" - Less Readable

With Verbatim Literal: @"C:\Pragim\DotNet\Training\Csharp" - Better Readable

```
string path1= @"D:\!Desktop\DATA FIX\Appointment\\05042019";
Console.WriteLine(" {0}", path1);
```
Output→ D:\!Desktop\DATA FIX\Appointment\05042019

Common Operators

Assignment Operator

=

Arithmetic Operators like

+,-,*,/,%

Comparison Operators like

==, !=,>, >=, <, <=

Conditional Operators like

&&, ||

Ternary Operator

?:

```
        int num = 11;
        int num1 = 11;
        bool isnum, isnum1;
        if (num == 10)
        {
            isnum = true;
        }
        else {
            isnum = false;
        }
       Console.WriteLine("Number == 10 is {0}", isnum);
        //Replaced by
        isnum1=(num1 == 11?true:false);
       Console.WriteLine("Number == 11 is {0}", isnum1);
```

Null Coalescing Operator

??

In C# types are divided into 2 broad categories.

Value Types - int, float, double, structs, enums etc

Reference Types - Interface, Class, delegates, arrays etc

By default, value types are non-nullable. To make them nullable use ?

int i = 0 (i is non nullable, so i cannot be set to null, i = null will generate compiler error)

int? j = 0 (j is nullable int, so j = null is legal)

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

string is a class reference type

```
        string s = null;
       // int i = null;   //error as  value type cannot be null

        int? i = null;  //this is not error.


       int? avl = null;
        int num;
        if (avl == null)
        {
            num = 0;
        }
        else
            {
            num = avl.Value;   //as null cannot implicitly convertible
            //or cast operator
            num =(int)avl;
        }
//----------or----Null Coalescing Operator--------
               int? avl1 = null;
               int num1= avl1??0; //if avl1 is null then the default value here 0 else
the value of avl1
```

Datatype Conversions

Implicit & Explicit conversion

Implicit conversion is done by the compiler:

1. When there is no loss of information if the conversion is done.

2. If there is no possibility of throwing exceptions during the conversion.

Example: Converting an int to a float will not lose any data and no exception will be thrown, hence an implicit conversion can be done.

```
        //Implicit Conversions
        int i = 10;
        float f = i;
        Console.WriteLine("i={0} and f={1}",i,f);  //Output   i=10 and f=10
```

Whereas when converting a float to an int, we lose the fractional part and also a possibility of overflow exception. Hence, in this case an explicit conversion is required. For explicit conversion we can use cast operator or the convert class in c#.

```
//Explicit Conversions

        float f1 = 10.52f;
        int i1=(int) f1;   //type cast operator
        Console.WriteLine("i1={0} and f1={1}", i1, f1); //Output   i1=10 and f1=10.52

        float f2= 112.52f;
        int i2 = Convert.ToInt16(f2);   //Convert class
        Console.WriteLine("i2={0} and f2={1}", i2, f2); //Output   i2=113 and
        f2=112.52
```

Difference between type cast and Convert

```
       float f3 = 1273733434333.45f;
        int i3 = (int)f3;   //type cast operator
        Console.WriteLine("i3={0} and f3={1}", i3, f3);  //Output    i3=-2147483648
and f3=1.2737334E+12
```

Here since the capacity of a float is greater than integer, the cast operator without throwing any exception converts the float into the int. But the int variable holds the lowest value that an int can hold.

```
        float f4 = 1273733434333.45f;
        //int i4 = Convert.ToInt16(f4);    //Convert class
        //int i5 = Convert.ToInt32(f4);    //Convert class
        long i6 = Convert.ToInt64(f4);    //Convert class
        Console.WriteLine("i6={0} and f4={1}", i6, f4);  //Output    i6=1273733447680
        and f4=1.2737334E+12
```

i4 and i5 will throw exception as the value of f4 is exceeds the range.

Difference between Parse and TryParse

If the number is in a string format we have 2 options-

Parse() and TryParse()

Parse() method throws an exception if it cannot parse the value, whereas TryParse() returns a bool indicating whether it succeeded or failed.

```
        string num = "100";
        int j = int.Parse( num);
        Console.WriteLine("String num={0} and int j={1}", num, j);

//Output    String    num=100 and int j=100

        string num2 = "100A";
        int j2 = int.Parse(num2);
        Console.WriteLine("String num2={0} and int j2={1}", num2, j2);

//Output    Error 'Input string was not in a correct format. In this case should use
TryParse'
```

Use Parse() if sure about the value will be valid, otherwise use TryParse()

```
        bool b = int.TryParse(num2, out j2);
        Console.WriteLine("String num2={0} and New int j2={1} and b={2}", num2, j2,b);

//Output    String num2=100A and New int j2=0 and b=False . Since the string could not
be converted to int

         b = int.TryParse(num, out j);
        Console.WriteLine("String num={0} and New int j={1} and b={2}", num, j, b);

//Output       String num=100 and New int j=100 and b=True
```

Array

A group of similar datatypes into a single variable i.e. strongly typed.

```
int[] evenNums =new int[3];
 evenNums[0] = 0;
 evenNums[1] = 2;
 evenNums[2] = 4;
```

Disadvantages:

- Arrays cannot grow in size once initialized.
- Have to rely on integral indices to store or retrieve items from the array.

Comment

```
//   Single Line Comment

/*  */  Multi line Comments

///   XML Documentation comment
```

IF Else

```
if(condition1 || condition2 || condition3)
```

If condition1 is true, condition 2 and 3 will NOT be checked.

```
if(condition1 | condition2 | condition3)
```

This will check conditions 2 and 3, even if 1 is already true.

```
if(condition1 && condition2 && condition3)
```

If condition1 is false, condition 2 and 3 will NOT be checked.

```
if(condition1 & condition2 & condition3)
```

This will check conditions 2 and 3, even if 1 is already false.

`|| and && are called Short-Circuit operators .`

While

1. While loop checks the condition first.

2. If the condition is true, statements within the loop are executed.

3. This process is repeated as long as the condition evaluates to true.

Note: Don't forget to update the variable participating in the condition, so the loop can end, at some point

Do While

1. A do loop checks its condition at the end of the loop.

2. This means that the do loop is guaranteed to execute at least one time.

3. Do loops are used to present a menu to the user

Difference - while & do while

1. While loop checks the condition at the beginning, whereas do while loop checks the condition at the end of the loop.

2. Do loop is guaranteed to execute at least once, whereas while loop is not.

FOR

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

ForEach loop

A foreach loop is used to iterate through the items in a collection. For example, foreach loop can be used to traverse through arrays or collections such as ArrayList, HashTable and Generics.

Break and Continue

The break statement can be used to jump out of a **loop**. A Break statement breaks out of the loop at the current point or we can say that it terminates the loop condition.

The continue statement breaks one iteration (in the loop), if a specified condition occurs, and continues with the next iteration in the loop. A Continue statement jumps out of the current loop condition and jumps back to the starting of the loop code.

Methods

- Methods are also called as functions. These terms are used interchangeably.
- Methods are extremely useful because they allow to define the logic once, and use it, at many places.
- Methods make the maintenance of the application easier.

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

1. Return type can be any valid data type or void.
2. Method name can be any meaningful name.
3. Parameters are optional

Instance vs Static Method

When a method declaration includes a **static** modifier, that method is said to be a **static** method. When **no static modifier** is present, the method is said to be an **instance** method.

Static method is invoked **using the class name**, where as **an instance method** is invoked **using an instance of the class**.

The difference between instance methods and static methods is that, **multiple instances of a class can be created (or instantiated) and each instance has its own separate method**. However, when a method is **static**, there **are no instances of that method**, and will invoke **only that one definition of the static method**.

Method parameter types

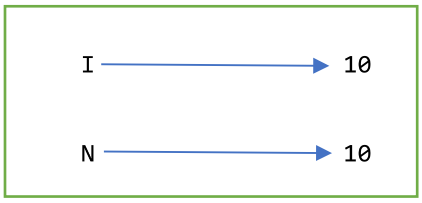
There are 4 different types of parameters a method can have

```
    static void Main()
    {
        int[] arr = new int[3];
        arr[0] = 98;
        arr[1] = 89;
        arr[2] = 79;

        int i = 10, j = 9, k = 1, s = 0;
        float f = 0.0f;
        valueParam(i);
        Console.WriteLine("Main 1 - {0}", i);     //   Output 10

        valueRef(ref i);
        Console.WriteLine("Main 2 - {0}", i);     //   Output 11

        f = outParameter(i, j, out k, out s);
        Console.WriteLine("outParameter from Main -Sum of {0} and {1} is {2} , Product
{3} and division {4}", i, j, s, k, f);     //   Sum of 11 and 9 is 20 , Product 99 and
division 1
i is here 11


        Console.WriteLine("paramsArray");

        Console.WriteLine("---------No parameter---------\n");
        paramsArray();
        Console.WriteLine("---------With parameter---------\n");
        paramsArray(arr);
        Console.WriteLine("---------With number of parameter---------\n");
        paramsArray(1, 2, 3, 4, 5, 67);
        paramsArray2("Swapnadip", 'A','B','C','D','E');

    }
```

Value Parameters

Creates a copy of the parameter passed, so modifications do not affect each other.

```
    public static void valueParam(int n)
    {
        n++;
        Console.WriteLine("valueParam - {0}",n);  //   Output 11
    }
```

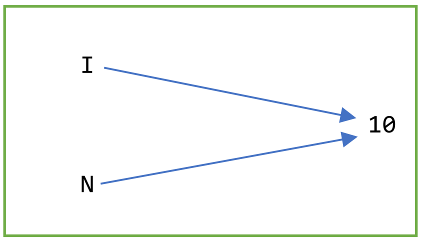


Here I and J are pointing to different memory locations. Operations on one variable will not affect the value of the other variable. In this case two different variables are created.

Reference Parameters

The ref method parameter keyword on a method parameter causes a method to refer to the same variable that was passed into the method. Any changes made to the parameter in the method will be reflected in that variable when control passes back to the calling method.

```
public static void valueRef(ref int n)
    {
        n++;
        Console.WriteLine("valueRef - {0}", n);  //   Output 11
    }
```



I and J are pointing to the same memory location. Operations on one variable will affect the value of the other variable. In this case no separate variables are created. Both I and n are pointing to same location.

Out Parameters

Use when a method to return more than one value.

```
    public static float outParameter(int n, int j, out int prod, out int sum)
    {
        prod = n * j;
        sum = n + j;
        return n / j;

    }
```

Parameter Arrays

The params keyword lets specify a method parameter, that takes **a variable number of arguments**. A **comma-separated list of arguments, or an array, or no arguments** can be sent to the method. params keyword **should be the last one in a method declaration**, and **only one params keyword is permitted** in a method declaration. Also, while calling the function, it may possible to call without any arguments. Calling a method with different techniques

```
valueParams();       or
valueParams(arr);    or
valueParams(1,2,3,4,5,67);
```

```
    public static void paramsArray(params int[] numbers)
    {
        int sum = 0;
        Console.WriteLine("paramsArray Number of parameter array - {0}",
numbers.Length);
        Console.Write("The numbers are ");
        for (int i = 0; i < numbers.Length; i++)
        {
            Console.Write(numbers[i]+" ");
        }
        foreach (int i in numbers)
        {
            sum = sum + i;
        }
        Console.WriteLine("\nSum is {0}", sum);
    }
```

This permit declaring

```
public static void paramsArray(char x, params int[] numbers)
```

but not

```
public static void paramsArray(params int[] numbers, char x)    OR
public static void paramsArray(params int[] numbers, params char[] x)
```

```
    public static void paramsArray2(string x, params char[] prop)
    {
        Console.WriteLine("paramsArray-2 {0} has the following properties ",x);
        for (int i = 0; i < prop.Length; i++)
        {
            Console.Write(prop[i] + " ");
        }
    }
```

Method Parameter Vs Method Argument

When declaring a method --- **Method Parameter**

```
public static void paramsArray2(string x, params char[] prop)
```

When calling or invoking a method --- **Method Argument**

```
valueParams(1,2,3,4,5,67);
```

Namespaces

- Namespaces are used to organize codes.
- Provide assistance in avoiding name clashes.
- We can use Alias for Namespaces like

```
using System;
using PATA= ProjectA.TeamA;
using PATB= ProjectA.TeamB;
using System.Text;
class NameSpacesDemo
{
    static void Main()
    {
        PATA.ClassA.Print();
        PATB.ClassA.Print();
    }
}


namespace ProjectA
{
    namespace TeamA
    {
        class ClassA
        {
            public static void Print()
            {
                Console.WriteLine("This is Class A Team A");
            }

        }
    }

    namespace TeamB
    {
        class ClassA
        {
            public static void Print()
            {
                Console.WriteLine("This is Class A Team B");
            }

        }
    }

}
```

Namespaces don't correspond to file, directory or assembly names. They could be written in separate files and/or separate assemblies and still belong to the same namespace.

Namespaces can be nested in 2 ways.

Namespace alias directives. Sometimes, may encounter a long namespace and wish to have it shorter. This could improve readability and still avoid name clashes with similarly named methods.

If all the namespaces and Classes are in a single file, then these will reside in the Bin folder of the main project with the **<mainProjectName>.exe**

But if separate Class Libraries are created then separate folders will be created and separate dlls will be created for each namespaces/Class Libraries

Class

There are simple data types like int, float, double etc. If a complex custom types data type required, then need to make use of classes.
A class consists of data and behaviour. Class data is represented by its fields and behaviour is represented by its methods.

```
class Customer
{
    string _firstName;
    string _lastName;

    public void PrintFullName()
    {
        Console.WriteLine("Full Name ={0} {1}", _firstName, _lastName);
        //or
        Console.WriteLine("Full Name ={0} {1}", this._firstName, this._lastName);
    }

    public Customer(string FName, string LName)
    {
        _firstName = FName;
        _lastName = LName;

        //or

        this._firstName = FName;
        this._lastName = LName;
    }



    ~Customer()
    {
        //Clean up code

    }
}
```

Constructor

Used to initialize the fields of a class. It may take parameters, but never returns anything. Constructs are called automatically while creating an object of a class.

Syntax:

```
public Customer(string FName, string LName)
    {
        _firstName = FName;
        _lastName = LName;

        //or

        this._firstName = FName;
        this._lastName = LName;
    }
```

`this` keyword actually refers to the instance /object of that class.

Creating/declaring a constructor every time is not necessary.

```
    static void Main()
    {
        Customer c1 = new Customer();     // Error as already a Constructor with a
                                          parameter is created.
        Customer cc = new Customer("Swapnadip", "Saha");
        cc.PrintFullName();
    }
```

If a constructor was not declared, .net framework automatically create a constructor, with no parameters for that class. This constructor walks through all the fields of the class and initialized them to default.

```
          Customer c1 = new Customer();
          Customer cc = new Customer("Swapnadip", "Saha");     // Error as no
                                   Constructor with a parameter is created.
```

Once a constructor is defined, the default constructor will not exist anymore. If want to use both, then declare a parameter less constructor.

```
       public Customer(): this("No First Name","No Last Name")
              {
```

```
                        //here this part is optional. If not provided, then the values
of class fields will be initialized to NULL, else these values, defined inside this
              }
```

Here `this` is referring to the Constructor with parameters.

Define the Constructors are like this with the same name but, different parameters or data types is called **Constructor Overloading.**

Static constructor

Static constructors are used to initialize static fields in a class. Declare a static constructor by using the keyword `static` in front of the constructor name.

Static Constructor is called only once, as it initiates once, and share across all the objects, no matter how many instances are being created.

Static constructors are called before instance constructors. In fact static constructors are called before anything of the class.

A constructor can also be a static member of a class. In that case , the static members of the class can be initialized using the static constructor. Access Modifier is not allowed for static Constructors. Static Constructors should be parameter less. By default, if there is no access modifier, then a field is private, and only accessed from that class.

A static constructor need not be called. Anytime, we refer to a static field, the static constructor will be automatically called even before that field, to initialize the static field. So, no need to call the static constructor explicitly. That's why, this do not have any access modifier, as some other class may able to call that constructor, and then it may call before any of the other fields may called.

Purpose of a class constructor

The purpose of a class constructor is to initialize class fields. A class constructor is automatically called when an instance of a class is created. Constructors do not have return values and always have the same name as the class.

Constructors are not mandatory. If we do not provide a constructor, a default parameter less constructor is automatically provided.

Constructors can be overloaded by the number and type of parameters.

Destructor

Destructors do not take parameters and never returns anything. Used to clean up the resources, the class was holding during its lifetime. Destructors are automatically called by garbage Collector. So, no need to declare it separately.

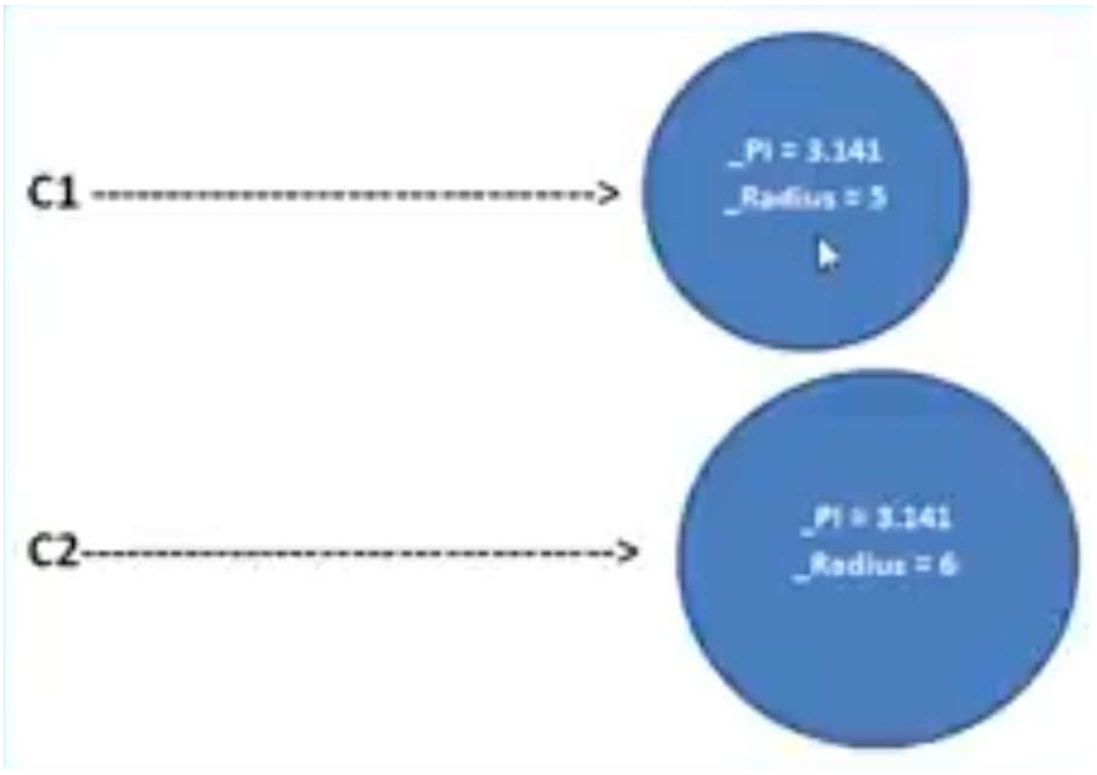
Syntax:

```
~Customer()
{
            //Clean up code
}
```

Static and Instance Class Members

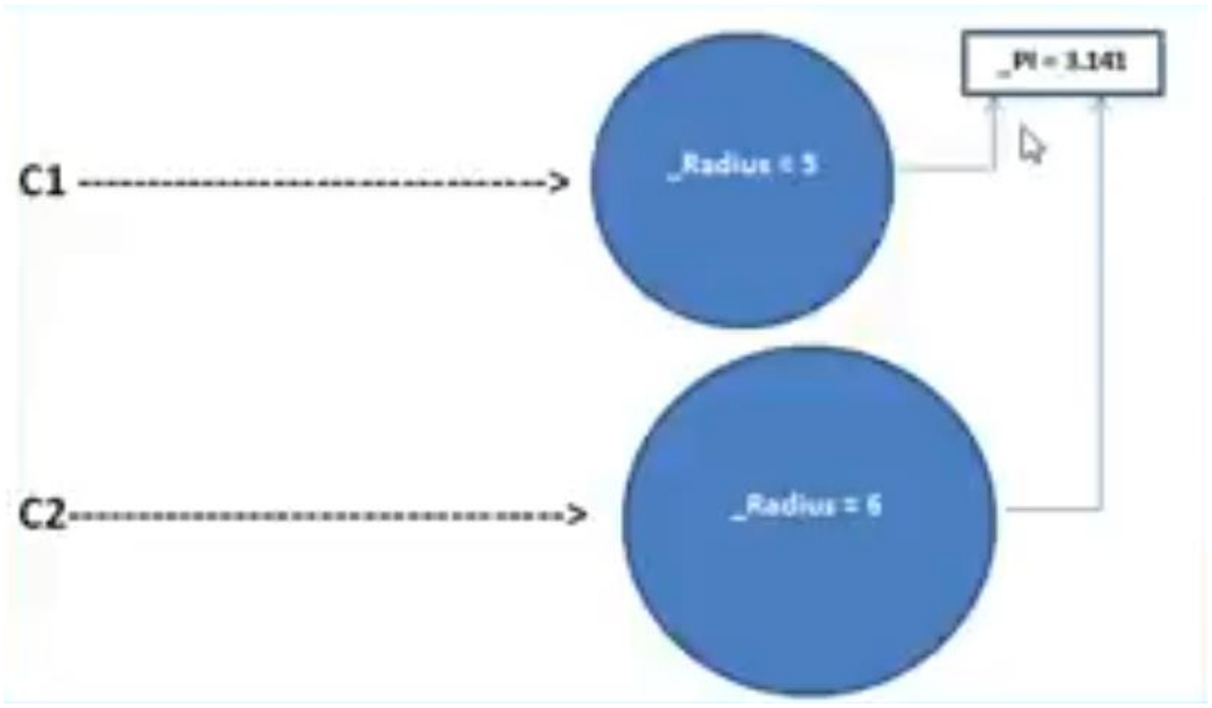
```
    static void Main()
    {
        Console.Write("Enter Radius");
        int radius = Convert.ToInt32(Console.ReadLine());
        Circle c1 = new Circle(radius);
        Console.WriteLine("The Area is {0}", c1.CalcArea());
        Circle c2 = new Circle(6);
        Console.WriteLine("The Area is {0}", c2.CalcArea());
    }

   class Circle
    {
      float _pi = 3.141f;
      int _radius;

      public Circle(int Radius)
       {
         this._radius = Radius;
       }

      public float CalcArea()
       {
        return this._pi * this._radius * this._radius;
       }
    }
```

In the above code, the value of radius is changing, but no matter, how many objects are created, the value of PI is always the same. The fields in the class are non-static instance fields.



In memory, for every instance creation, a copy of _Pi and _radius is created. This not needed, as _Pi is a constant.

But if _PI is static, then for every instance, _PI will be shared with the objects and a new radius will be created. A static field cannot be referring with `this` keyword. As `this` keyword refer to an object or instance of a class. It should be call as `<ClassName>.<fieldname>`



The same is true for static methods also.

```
class Circle
{
    static float _pi = 3.141f;
    int _radius;

    public Circle(int Radius)
    {
        this._radius = Radius;
    }

    public float CalcArea()
    {
        return _pi * this._radius * this._radius;  //or
        //return Circle._pi * this._radius * this._radius;
    }

       public static void Print(Circle c)
    {
        Console.WriteLine("Perimeter is {0}",2*Circle._pi* c._radius);
    }
}



    static void Main()
    {
        Console.Write("Enter Radius");
        int radius = Convert.ToInt32(Console.ReadLine());
        Circle c1 = new Circle(radius);
        Console.WriteLine("The Area is {0}", c1.CalcArea());
        Circle c2 = new Circle(6);
        Console.WriteLine("The Area is {0}", c2.CalcArea());

        Circle.Print(c1);
        Circle.Print(c2);
    }
```

```
class Circle
{
    static float _pi;
    int _radius;

    public Circle(int Radius)
    {
        this._radius = Radius;
    }

    static Circle()
    {
        Circle._pi = 3.141f;
    }
    public float CalcArea()
    {
        return _pi * this._radius * this._radius;  //or
        //return Circle._pi * this._radius * this._radius;

    }

    public static void Print(Circle c)
    {
        Console.WriteLine("Perimeter is {0}",2*Circle._pi* c._radius);
    }
```

- When a class member includes a `static` modifier, the member is called as static member. When no static modifier is present the member is called as non-static member or instance member.
- Static members are invoked using class name, whereas instance members are invoked using instances (objects) of the class.
- An instance member belongs to specific instance(object) of a class. If 3 objects of a class are created, then there would be 3 sets of instance members in the memory, whereas there will ever be only one copy of a static member, no matter how many instances of a class are created

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

Static constructor

- Static constructors are used to initialize static fields in a class. Declare a static constructor by using the keyword `static` in front of the constructor name.

- Static Constructor is called only once, as it initiates once, and share across all the objects, no matter how many instances you create.
- Static constructors are called before instance constructors

```
using System;

class Program
{
        Console.Write("Enter Radius");
        int radius = Convert.ToInt32(Console.ReadLine());
        Circle c1 = new Circle(radius);
        Console.WriteLine("The Area is {0}",c1.CalcArea());
        Circle c2 = new Circle(6);
        Console.WriteLine("The Area is {0}", c2.CalcArea());
        Circle c3 = new Circle();
        Console.WriteLine("The Area is {0}", c3.CalcArea());
        Circle.Print(c1);
        Circle.Print(c2);
        Circle.Print(c3);
}

class Circle
{
    static float _pi;
    int _radius;

    public Circle()
    {
        Console.WriteLine("ParameterLess Const");
    }

    public Circle(int Radius)
    {
        Console.WriteLine("Parameterwala Const");
        this._radius = Radius;
    }

    static Circle()
    {
        Console.WriteLine("Static Const");
        Circle._pi = 3.141f;
    }
    public float CalcArea()
    {
        Console.WriteLine("Float Method");
        return _pi * this._radius * this._radius;  //or
        //return Circle._pi * this._radius * this._radius;

    }

    public static void Print(Circle c)
    {
        Console.WriteLine("Static Method");
        Console.WriteLine("Perimeter is {0}",2*Circle._pi* c._radius);
    }
}

//OUTPUT
Static Const
Parameterwala Const
Float Method
```

```
The Area is 78.525
Parameterwala Const
Float Method
The Area is 113.076004
Parameterwala Const
ParameterLess Const
Float Method
The Area is 0
Static Method
Perimeter is 31.41
Static Method
Perimeter is 37.692
Static Method
Perimeter is 0
```

Pillars of Object-Oriented Programming

1. Inheritance
2. Encapsulation
3. Abstraction
4. Polymorphism

Inheritance

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

Note: All the common fields, properties, methods can be specified in the base class, which allows reusability. In the derived class only have fields, properties and methods, that are specific to them.

Inheritance Concepts

Use the `new` keyword to hide a base class member. Will get a compiler warning, if missed the `new` keyword.
Different ways to invoke a hidden base class member from derived class
1. Use `base` keyword
2. Cast child type to parent type and invoke the hidden member
3. ParentClass PC = new ChildClass()
PC.HiddenMethod()

Inheritance Syntax

```
public class ParentClass
{
    // Parent Class Implementation
}
public class DerivedClass : ParentClass
{
    // Child Class Implementation
}
```

1. In this example DerivedClass inherits from ParentClass.
2. C# supports only single class inheritance. A child class must have only one Base/Parent class.

```
public class ParentClass
{
    // Parent Class Implementation
}

public class NewParentClass
{
    //New Parent Class Implementation
}

public class DerivedClass : ParentClass , NewParentClass
{
    // Child Class Implementation
    // This is not possible
}
```

3. C# supports multiple level inheritance. i.e.

```
public class ParentClass
{
    // Parent Class Implementation
}
public class DerivedClass : ParentClass
{
    // Child Class Implementation
}
public class NewChildClass : DerivedClass
{
    //New Child Class Implementation
    // This is possible

}
```

4. C# supports multiple interface inheritance.
5. Derived class is a specialization of base class.
6. Base classes are automatically instantiated before derived classes. i.e. Any time an instance of a child class is created/executed, the constructor of base class is automatically created/executed.
7. Parent Class constructor executes before Child Class constructor.
8. If a parent class do have two constructors, a Parameter Less constructor and a constructor with parameters, then the parameter less constructor will be called first. But it can be controlled by the child class, that which one should call first.

Case 1:

```
using System;

class InherittedConstructorCalling
{
    static void Main()
    {
        SonClass c1 = new SonClass();
    }
}

public class FatherClass
{
    public FatherClass()
    {
        Console.WriteLine("Father Class Parameter Less Constructor.");
    }
}
public class SonClass : FatherClass
{
    public SonClass()
    {
        Console.WriteLine("Son Class Parameter Less Constructor.");
    }
}

//Output

Father Class Parameter Less Constructor.
Son Class Parameter Less Constructor.
```

Case 2:

```
using System;

class ConstructorCallingInheritance
{
    static void Main()
    {
        DaughterClass c1 = new DaughterClass();
    }
}

public class MotherClass
{
    public MotherClass()
    {
        Console.WriteLine("Mother Class Parameter Less Constructor.");
    }
    public MotherClass(string Message)
    {
        Console.WriteLine("Mother Class Constructor with Parameter .");
        Console.WriteLine(Message);
    }
}
public class DaughterClass : MotherClass
{
    public DaughterClass():base("Daughter controlling Mother")
    {
        Console.WriteLine("Daughter Class Parameter Less Constructor.");
    }
}

//Output

Mother Class Constructor with Parameter .
Daughter controlling Mother
Daughter Class Parameter Less Constructor.
```

The outputs of case 1 and case 2 will be same in case 3 and case 4 respectively.

```
        Console.WriteLine("-----Case 3-----");
        FatherClass  S2 =new SonClass();
        Console.WriteLine("-----Case 4-----");
        MotherClass  c2 = new DaughterClass();
```

Method Hiding

Method hiding is used, to hide a base class method from derived class by defining the method with same name, but may have different functionality. To make acknowledged that this hiding is intentional in derived class, an optional new keyword needs to be introduced in derived class, to avoid the warnings.

Case 1:

```
using System;

class MethodHiding
{
    static void Main()
    {
        FullTimeEmployee fte = new FullTimeEmployee();
        fte.FirstName = "Swapnadip";
        fte.LastName = "Saha";
        fte.PrintFullName();

        PartTimeEmployee pte = new PartTimeEmployee();
        pte.FirstName = "Saha";
        pte.LastName = "Swapnadip";
        pte.PrintFullName();
    }
}

public class Employee
{
    public string FirstName;
    public string LastName;
    public void PrintFullName()
    {
        Console.WriteLine("Name of the Employee is {0} {1}",  FirstName, LastName);
    }

}

public class PartTimeEmployee : Employee
{
    float hourleyWage;
    public new void PrintFullName()
    {
        Console.WriteLine("Name of the Employee is {0} {1} - Contract",  FirstName, 
LastName);   //Case 1:  Method hiding - Creating own method in derived class
    }
}


public class FullTimeEmployee : Employee
{
    float monthlySalary;
}

// OUTPUT

Name of the Employee is Swapnadip Saha
Name of the Employee is Saha Swapnadip – Contract
```

Case 2:

After hiding the base class method, if still want to call the base class method.

```
public class PartTimeEmployee : Employee
{
    float hourleyWage;
    public new void PrintFullName()
    {
        base.PrintFullName();   //Case 2: Calling base class in spite of Method hiding
    }
}

// OUTPUT

Name of the Employee is Swapnadip Saha
Name of the Employee is Saha Swapnadip
```

Case 3:

```
((Employee)pte).PrintFullName(); // Case 3 Calling Base method from derived method, in
spite of Method hiding

public class PartTimeEmployee : Employee
{
    float hourleyWage;

    public new void PrintFullName()
    {
        Console.WriteLine("Name of the Employee is {0} {1} - Contract", FirstName,
LastName);

    }
}

// OUTPUT

Name of the Employee is Swapnadip Saha
Name of the Employee is Saha Swapnadip
```

Case 4:

```
Calling Base method from derived method, in spite of Method hiding


        Employee pte1 = new PartTimeEmployee();    // Vice versa is not possible
        pte1.FirstName = "Saha";
        pte1.LastName = "Swapnadip";
        pte1.PrintFullName();

// OUTPUT

Name of the Employee is Swapnadip Saha
Name of the Employee is Saha Swapnadip
```

Sealed Class:

A sealed class have the keyword `sealed`, it cannot be inherited by other classes.

The following code is not allowed

```
public sealed class SealBasedClass
{
    static void Print()
    {

    }
}

public class SealDerivedClass : SealBasedClass //'SealDerivedClass': cannot derive
from sealed type 'SealBasedClass'
{
    static void Print()
    {

    }
}
```

Polymorphism

Polymorphism is one of the primary pillars of object-oriented programming.

Polymorphism allows to invoke derived class methods through a base class reference during runtime.

Dynamic/Runtime

Virtual/Override

Let consider the following code

```
using System;

public class Employees
{
    public string FName = "Swapnadip";
    public string LName = "Saha";
    public void PrintFullName()
    {
        Console.WriteLine("Name of the Employee is {0} {1}", FName, LName);
    }

}

public class PartTimeEmployees : Employees
{
    public new void PrintFullName()
    {
        Console.WriteLine("Name of the Employee is {0} {1} - Part Time", FName,
LName);
    }
}
public class FullTimeEmployees : Employees
{
    public new void PrintFullName()
    {
        Console.WriteLine("Name of the Employee is {0} {1} - Full Time", FName,
LName);
    }
}

public class TemporaryEmployees : Employees
{
    public new void PrintFullName()
    {
        Console.WriteLine("Name of the Employee is {0} {1} - Tempo", FName, LName);
    }
}

public class OtherEmployees : Employees
{
    public new void PrintFullName()
    {
        Console.WriteLine("Name of the Employee is {0} {1} - Other", FName, LName);
    }
}

public class ContractEmployees : Employees
{
    public new void PrintFullName()
    {
        Console.WriteLine("Name of the Employee is {0} {1} - Contract", FName, LName);
    }
}


class Polymorphism
{
    static void Main()
    {
        Employees[] emp = new Employees[6];
        emp[0] = new Employees();
        emp[1] = new PartTimeEmployees();
        emp[2] = new FullTimeEmployees();
        emp[3] = new TemporaryEmployees();
        emp[4] = new OtherEmployees();
        emp[5] = new ContractEmployees();

        foreach (var e in emp)
        {
            e.PrintFullName();
        }

    }
}

//OUTPUT
Name of the Employee is Swapnadip Saha
Name of the Employee is Swapnadip Saha
Name of the Employee is Swapnadip Saha
Name of the Employee is Swapnadip Saha
Name of the Employee is Swapnadip Saha
Name of the Employee is Swapnadip Saha
```

Here we have created base class object with child class references, the method hiding will not work, and the objects will call the base class method.

To make the derived class function to be worked, we need to make the base class function virtual and override the derived class methods.
To do so, the base class method is declared `virtual`, and the derived class method is declared as `override`.
The `virtual` keyword indicates, the method can be overridden in any derived class.

```
using System;
```

```
public class Employees
{
    public string FName = "Swapnadip";
    public string LName = "Saha";
    public virtual void PrintFullName()
    {
        Console.WriteLine("Name of the Employee is {0} {1}", FName, LName);
    }

}

public class PartTimeEmployees : Employees
{
    public override void PrintFullName()
    {
        Console.WriteLine("Name of the Employee is {0} {1} - Part Time", FName,
LName);
    }
}
public class FullTimeEmployees : Employees
{
    public override void PrintFullName()
    {
        Console.WriteLine("Name of the Employee is {0} {1} - Full Time", FName,
LName);
    }
}

public class TemporaryEmployees : Employees
{
    public override void PrintFullName()
    {
        Console.WriteLine("Name of the Employee is {0} {1} - Tempo", FName, LName);
    }
}

public class OtherEmployees : Employees
{
    public new void PrintFullName()
    {
        Console.WriteLine("Name of the Employee is {0} {1} - Other", FName, LName);
    }
}

public class ContractEmployees : Employees
{
    public new void PrintFullName()
    {
        Console.WriteLine("Name of the Employee is {0} {1} - Contract", FName, LName);
    }
}


class Polymorphism
{
    static void Main()
    {
        Employees[] emp = new Employees[5];
        emp[0] = new Employees();
        emp[1] = new PartTimeEmployees();
        emp[2] = new FullTimeEmployees();
        emp[3] = new TemporaryEmployees();
        emp[4] = new OtherEmployees();

        foreach (var e in emp)
        {
            e.PrintFullName();
        }

        ContractEmployees ce = new ContractEmployees();
        ce.PrintFullName();
    }
}


// OUTPUT

Name of the Employee is Swapnadip Saha
Name of the Employee is Swapnadip Saha - Part Time // Polymorphism  Virtual/Override
Name of the Employee is Swapnadip Saha - Full Time // Polymorphism  Virtual/Override
Name of the Employee is Swapnadip Saha – Tempo // Polymorphism  Virtual/Override
Name of the Employee is Swapnadip Saha //calling base class as the object has child
ref
Name of the Employee is Swapnadip Saha – Contract  //method hiding
```

The Virtual Base class method can still be accessed through a Derived class, where overridden methods are not present, or are available for method hiding also.
In the above code

```
public class OtherEmployees : Employees
{
    public new void PrintFullName()
    {
        Console.WriteLine("Name of the Employee is {0} {1} - Other", FName, LName);
    }
}

public class ContractEmployees : Employees
{
    public new void PrintFullName()
    {
        Console.WriteLine("Name of the Employee is {0} {1} - Contract", FName, LName);
    }
}
```

If a derived class does not have the implementation of the Base class method, then it will call the base class.

Difference between Method hiding and Method overriding

```
using System;
class OverridingVsHiding
{
    public class BaseClass
    {
        public void Print()   // here we can also include the virtual keyword
        {
            Console.WriteLine("I am base Class ");
        }

    }
    public class DerivedClass: BaseClass
    {
        Public new void Print()
        {
            Console.WriteLine("I am Derived Class");
        }

    }
    static void Main()
    {
        BaseClass B = new DerivedClass();
        B.Print();

        DerivedClass C = new DerivedClass();
        C.Print();

    }
}

// OUTPUT
I am base Class
I am Derived Class
```

In method overriding a base class reference variable pointing to a child case object, will **invoke the hiding method in base class**

```
using System;
class OverridingVsHiding
{
    public class BaseClass
    {
        public virtual void Print()
        {
            Console.WriteLine("I am base Class");
        }

    }

    public class DerivedClass: BaseClass
    {
        public override void Print()
        {
            Console.WriteLine("I am Derived Class");
        }

    }
    static void Main()
    {
        BaseClass B = new DerivedClass();
        B.Print();

        DerivedClass C = new DerivedClass();
        C.Print();
    }
}

// OUTPUT
```

```
I am Derived Class
I am Derived Class
```

In method overriding a base class reference variable pointing to a child case object, will **invoke the overridden method in child class**

Static or compile-time polymorphism

Method Overloading

Function overloading and Method overloading terms are used interchangeably.

Method overloading allows a class to have multiple methods with the same name, but, with a different signature. So, in C# functions can be overloaded based on the number, type(int, float etc) and kind(Value, Ref or Out) of parameters.

The signature of a method consists of the name of the method and the type, kind (value, reference, or output) and the number of its formal parameters. The signature of a method does not include the return type and the params modifier. So, it is not possible to overload a function, just based on the return type or params modifier.

Operator Overloading

Encapsulation

Encapsulation is defined as the wrapping up of data and information under a single unit. It is the mechanism that binds together the data and the functions that manipulate them. In a different way, encapsulation is a protective shield that prevents the data from being accessed by the code outside this shield.

- Technically in encapsulation, the variables or data of a class are hidden from any other class and can be accessed only through any member function of its own class in which they are declared.
- As in encapsulation, the data in a class is hidden from other classes, so it is also known as data-hiding.
- Encapsulation can be achieved by: Declaring all the variables in the class as private and using C# Properties in the class to set and get the values of variables.

To avoid marking the class fields public and exposing to the external world, as you will not have control over what gets assigned and returned.

Encapsulation applied for two reasons:

- If the members of a class are private then how another class in C# will be able to read, write, or compute the value of that field.
- If the members of the class are public then another class may misuse that member.

```
using System;
class Property
{
    static void Main()
    {
        Student C1 = new Student();
        C1.ID = -101;
        C1.Name = null;
        C1.PassMarks = -100;
        Console.WriteLine("ID {0} & Name {1} PassMarks {2}", C1.ID, C1.Name,
C1.PassMarks);
    }
}

public class Student
{
    public int ID;
    public string Name;
    public int PassMarks = 35;

}
//Output
ID -101 & Name  PasaMarks -100
```

1. Problems with Public Fields
2. ID should always be non-negative number
3. Name cannot be set to NULL
4. If Student Name is missing "No Name" should be returned
5. PassMark should be read only

Getter and Setter

The programming languages, which do not have the Property concept, are use getter and setter methods to encapsulate and protect fields.

```
using System;
class Property
{
    static void Main()
    {
        Student C1 = new Student();
        C1.SetId(-100);
        Console.WriteLine("ID {0}", C1.GetId());
    }
}

public class Student
{
    private int _ID;
    private string _Name;
    private int _PassMarks = 35;

    public void SetId(int ID)
    {
        if (ID <= 0)
        {
            throw new Exception("Student ID should be greater than 0");
        }
        this._ID = ID;
    }

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

Now in this example, ID should not be 0, and will throw the exception

In this example we use the `SetId(int ID)`and `GetId()` methods to encapsulate _id class field. As a result, we have better control on what gets assigned and returned from the _id field.

Note: Encapsulation is one of the primary pillars of object-oriented programming.

```
using System;
class GetterSetter
{
    static void Main()
    {
        Student C1 = new Student();
        C1.SetId(100);
        C1.SetName("Swapnadip Saha");
        Console.WriteLine("ID {0}, Name {1}, PassMarks {2} ",
C1.GetId(),C1.GetName(),C1.GetPassMarks());
    }
}

public class Student
{
    private int _ID;
    private string _Name;
    private int _PassMarks = 35;

    public void SetId(int ID)
    {
        if (ID <= 0)
        {
            throw new Exception("Student ID should be greater than 0");
        }
        this._ID = ID;
    }

    public int GetId()
    {
        return this._ID;
    }

    public void SetName(String Name)
    {
        if (string.IsNullOrEmpty(Name))
        {
            throw new Exception("name should not be Null");
        }
        this._Name = Name;
    }

    public string GetName()
    {
        return this._Name;
    }

    public int GetPassMarks()
    {
        return this._PassMarks;
    }
}
```

Now in this example ID is set to 100 and Name is set to a valid string. But the pass mark is always fixed, and nothing should be assigned to it. So only a GetPassMarks method or a getter is defined here. This are called **ReadOnly**.

Property

In C# to encapsulate and protect fields we use properties

1. We use get and set accessors to implement properties
2. A property with both get and set accessor is a Read/Write property
3. A property with only get accessor is a Read only property
4. A property with only set accessor is a Write only property

Note: The advantage of properties over traditional Get() and Set() methods is that, you can access them as if they were public fields

```
using System;
class Property
{
    static void Main()
    {
        Students C1 = new Students();
        C1.Id = 100;
        C1.Name = "Swapnadip Saha";
        Console.WriteLine("ID {0}, Name {1}, PassMarks {2} ", C1.Id, C1.Name,
C1.Passmrks);
    }
}

public class Students
{
    private int _ID;
    private string _Name;
    private int _PassMarks = 35;

    public int Id
    {
        set
        {
            if (value <= 0)
            {
                throw new Exception("Student ID should be greater than 0");
            }
            this._ID = value;
        }
        get
        {
            return this._ID;
        }
    }


    public string Name
    {
        set
        {
            if (string.IsNullOrEmpty(value))
            {
                throw new Exception("name should not be Null");
            }
            this._Name = value;
        }
        get
        {
            return (string.IsNullOrEmpty(this._Name)) ? "No Name" : this._Name;
        }

    }
    public int PassMarks
    {
        get
        {
            return this._PassMarks;
        }

    }
}
```

In Getter Setter we used two methods,

```
   public void SetId(int ID)
    {
        if (ID <= 0)
        {
            throw new Exception("Student ID should be greater than 0");
        }
        this._ID = ID;
    }

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

And in Main we called them as methods, one is for assigning value and another one is for reading the value.

```
        Student C1 = new Student();
        C1.SetId(100);
        Console.WriteLine("ID {0  ", C1.GetId())
```

But while using property, this can be declared as a variable, and can be accessed like a variable only,

```
public int Id
    {
        set
        {
            if (value <= 0)
            {
                throw new Exception("Student ID should be greater than 0");
            }
            this._ID = value;
        }
        get
        {
            return this._ID;
        }
    }

       Students C1 = new Students();
```

```
        C1.Id = 100;
        Console.WriteLine("ID {0} ", C1.Id);
```

Auto Implemented Properties

If there is no additional logic in the property accessors, then we can make use of auto- implemented properties .
Auto-implemented properties reduce the amount of code that we have to write.
When you use auto-implemented properties, the compiler creates a private, anonymous field that can only be accessed through the property's get and set accessors
For example;

```
       private string _city;
       private string _email;

    public string City
    {
        set
        {
            this._city = value;
        }
        get
        {
            return this._city;
        }
    }

    public string Email
    {
        set
        {
            this._email = value;
        }
        get
        {
            return this._email;
        }
    }
```

This is a lot of code.

This can be replaced by

```
    public string City { get; set; }
    public string Email { get; set; }
```

Complier will automatically create a private field, and the properties will be use to read and write.

Another syntax

```
public string LastName { get => _lastName; set => _lastName = value; }
/*Property syntax-- write click refactor*/
```

Structure

Just like classes structs can have

1. Private Fields
2. Public Properties

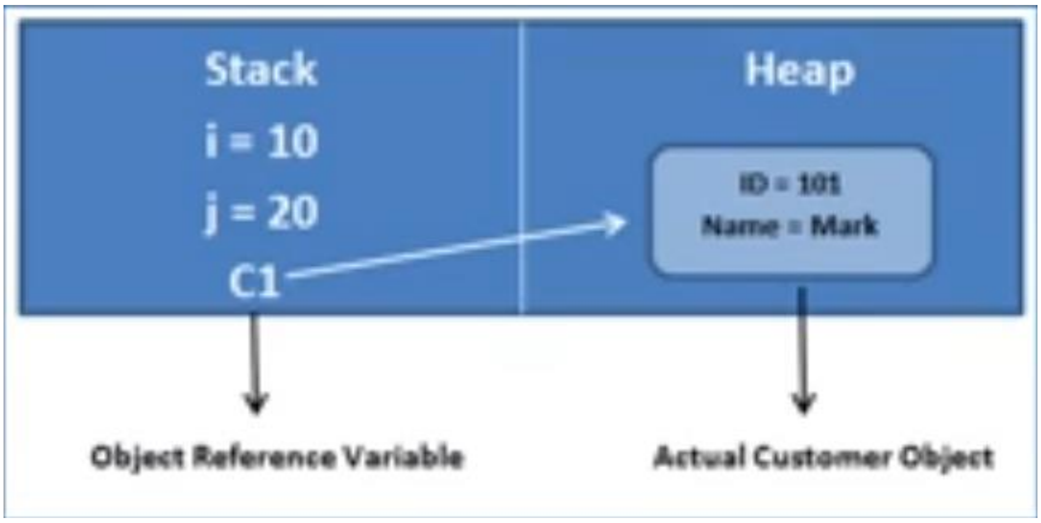
3. Constructors
4. Methods

Object initializer syntax, can be used to initialize either a struct or a class.

```
using System;

// Structure syntax --  Same like Class


public struct Customer
{
    private int _id;
    private string _name;
    public int id
    {
        get
        {
            return _id;
        }
        set
        {
            this._id = value;
        }
    }

    public string name
    {
        get
        {
            return _name;
        }
        set
        {
            this._name = value;
        }
    }

    public Customer(int Id, string Name)
    {
        this._id = Id;
        this._name = Name;

    }

    public void PrintCustomer()
    {
        Console.WriteLine("Id {0} and Name {1}", this._id, this._name);
    }
}


class Struct
{
    static void Main()
    {
        Customer c1 = new Customer();
        c1.PrintCustomer(); // The default Constructor is still working even after
defining the Constructor with Parameter

        Customer c2 = new Customer(1, "SS");
        c2.PrintCustomer(); // Initializing the values using the Constructor with
Parameter

        Customer c3 = new Customer();
        c3.id = 2;
        c3.name = "EC";
        c3.PrintCustomer();  // Calling the Properties (Encapsulation)


       Customer c4 = new Customer()
        {
            id = 3,
            name = "SC"
        };
// Object initializer syntax

        c4.PrintCustomer();
    }
}

// OUTPUT
Id 0 and Name                      default Constructor
Id 1 and Name SS                   Constructor with Parameter
Id 2 and Name EC                   Calling the Properties (Encapsulation)
Id 3 and Name SC                   Object initializer syntax
```

Difference between struct and class

- A **struct is a value type** where as a **class is a reference type**.
- All the differences that are applicable to value types and reference types are also applicable to classes and structs.
- **Structs are stored on stack**, whereas **classes are stored on the heap**.



```
    static void Main()
      {
        int i = 10;
        if (i == 10)
        {
            int j = 20;
            Customer C1 = new Customer();
            C1.id = 101;
            C1.Name = "Mark";
         }
```

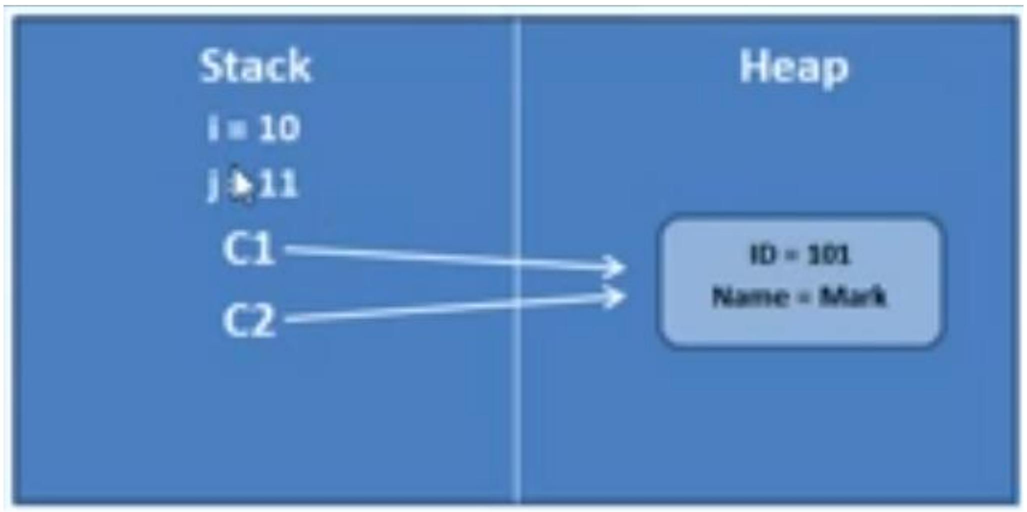
```
}
```

The `Customer` The object reference variable C1 is created in stack, but it refers to the Customer Object, this object is in Heap. The actual Customer Object is in Heap.

- Value types hold their value in memory where they are declared, but reference types hold a reference to an object in memory.
- **Value types are destroyed immediately after the scope is lost**, whereas for reference types only the **reference variable is destroyed after the scope is lost**. The **object is later destroyed by garbage collector.**

In the above code, the scope of `int i` is the entire `Main()` function. But `int j` is available only inside the `if` block. As soon as the execution of `if` block is done `int j` will be destroyed immediately and the execution of `Main()` is done `int i` will be destroyed immediately .
Now As soon as the execution of `if` block is done, the scope of object reference variable `Customer C1` ended. i.e. C1 will be destroyed, but the object will be still be there in heap.

- When you copy a struct into another struct, a new copy of that struct gets created and modifications on one struct will not affect the values contained by the other struct.
  But When you copy a class into another class, we only get a copy of the reference variable. Both the reference variables point to the same object on the heap. So, operations on one variable will affect the values contained by the other reference variable.



```
int i = 10;
int j = i;
Console.WriteLine("i {0} and j {1}", i, j);   // i=10 j=10
j++;
Console.WriteLine("i {0} and j {1}", i, j);   // i=10 j=11
```

In this case two different variables will be created, changing the value of j will not affect the value of i. But

```
Customer cc1= new Customer();
cc1.id = 102;
cc1.Name = "EC";
Customer cc2= cc1;
```

```
        Console.WriteLine("cc1.id {0} and cc1.Name {1} cc2.id {0} and cc2.Name {1}",
cc1.id, cc1.Name, cc2.id, cc2.Name);   //cc1.id 102 and cc1.Name EC cc2.id 102 and
cc2.Name EC
        cc2.id = 103;
        Cc2.Name = "SC";
        Console.WriteLine("cc1.id {0} and cc1.Name {1} cc2.id {0} and cc2.Name {1}",
cc1.id, cc1.Name, cc2.id, cc2.Name);   //cc1.id 103 and cc1.Name SC  cc2.id 103 and
cc2.Name SC
```

In this case the object references variables CC1 and CC2 are pointing to a single object of Customer Class. So, changing one is affecting the other.

- Structs can't have destructors, but classes can have destructors. Since structs are value type, they will be destroyed, once its current scope is ended, so need of destructors
- Structs cannot have explicit parameter less constructor where as a class can.
- Struct can't inherit from another class where as a class can, **both structs and classes can inherit from an interface. structure cannot be a parent/Base in case of inheritance**

Examples of structs in the.NET Framework- int (System.Int32), double(System.Double) etc.

Note: A class or a struct cannot inherit from another struct. Struct are sealed types.

Interface

- We create interfaces using `interface` keyword. Just like classes interfaces also contains properties, methods, delegates or events, **but only declarations and no implementations.** It is a compile time error to provide implementations for any interface member.
- **Interface members are public by default**, and they don't allow explicit access modifiers, even the `public` keyword.
- Interfaces **cannot contain fields**.
- If a `class` or a `struct` inherits from an interface, it must provide **implementation for all interface members, and they should have same signatures also.** Otherwise, we get a compiler error.
- A `class` or a `struct` **can inherit from more than one interface at the same time** and in that case ,the `class` or the `struct` should have the method implementations from all the interfaces. But whereas, a class cannot inherit from more than once class at the same time.
- **Interfaces can inherit from other interfaces**. A class that inherits this interface must provide implementation for all interface members in the entire interface inheritance chain.
- We **cannot create an instance of an interface**, but **an interface reference variable can point to a derived class object.**

Explicit Interface Implementation

As a class can inherit multiple interfaces, in the following code:

```
using System;


interface IInterfaceCustomer
{
    void Print();
}
interface IInterfaceCustomer2
{
    void Print();
}
class Interfaces : IInterfaceCustomer, IInterfaceCustomer2
{
    public void Print()
    {
        Console.WriteLine("Print");
    }
    static void Main()
    {
        Interfaces I1 = new Interfaces();
        I1.Print();
    }
}
```

Now in this code, as the both interfaces do have same method declaration, it is not sure, the class `Interfaces` will invoke which one. In this case a conversion may needed. We are using Explicit Interface Implementation technique to solve this problem.

Note: When a class explicitly implements, an interface member, the interface member can no longer be accessed thru class reference variable, but only thru the interface reference variable.

`Access modifiers` are not allowed on explicitly implemented interface members.

Type Cast

```
class Interfaces : IInterfaceCustomer, IInterfaceCustomer2
{
    public void Print()
    {
        Console.WriteLine("Print");
    }
    static void Main()
    {
        Interfaces I1 = new Interfaces();
         I1.Print();
        ((IInterfaceCustomer)I1).Print();
        ((IInterfaceCustomer2)I1).Print();
    }
}
```

In this case, while instantiate the interface,

`I1.Print();`

This will also work.

Explicit Conversion

```
class Interfaces : IInterfaceCustomer, IInterfaceCustomer2
{
     void IInterfaceCustomer.Print()
    {
        Console.WriteLine("Print 1");
    }

     void IInterfaceCustomer2.Print()
    {
        Console.WriteLine("Print 2");
    }
    static void Main()
    {
         Interfaces I1 = new Interfaces();
        ((IInterfaceCustomer)I1).Print();
        ((IInterfaceCustomer2)I1).Print();

    }
}
```

In this case, while implementing the methods, `access modifiers` are not required. In this case, while instantiate the interface,

```
I1.Print();
```

This will not work, for the ambiguity issue.

Inheritance

```
              IInterfaceCustomer i2 =new Interfaces();
             i2.Print();


              IInterfaceCustomer2 i3 = new Interfaces();
             i3.Print();
```

Default

 If you want to make one of the interface methods, the default, then implement that method normally and the other interface method explicitly. This makes the default method to be accessible thru class instance.

```
using System;


interface IInterfaceCustomer
{
    void Print();
}
interface IInterfaceCustomer2
{
    void Print();
}
class Interfaces : IInterfaceCustomer, IInterfaceCustomer2
{
    public void Print()
    {
        Console.WriteLine("Print 1");  //default
    }

    void IInterfaceCustomer2. Print()
    {
        Console.WriteLine("Print 2");
    }
    static void Main()
    {

        Interfaces I4 = new Interfaces();
        I4.Print();
       ((IInterfaceCustomer2)I4). Print();

    }
}
```

In this case as `IInterfaceCustomer2.Print()` is defined explicitly,
```
 I4.Print();
```
will call the method from `IInterfaceCustomer1`. To call the method from `IInterfaceCustomer2` need to call the method using explicit conversion

```
        ((IInterfaceCustomer2)I4). Print();
```

So, the method which is defined normally, will be the default method.

Abstract Class

- The `abstract` keyword is used to create `abstract` classes.
- An `abstract` class is incomplete and hence cannot be instantiated.
- An `abstract` class can only be used as base class. **A non-abstract class derived from an abstract class must provide implementations for all inherited abstract members. The functions should be** `override` **function.** Since an `abstract` class can be only a base class, its `abstract` members will be always public.
- An `abstract` class cannot be `sealed`. Since `abstract` class can be only a base class and `sealed` classes cannot be inherited. So, they are contradictory properties.
- An `abstract` class may contain `abstract` members(methods, properties, indexers, and events), but not mandatory or **non-abstract members.**
- If a class inherits an `abstract` class, there are 2 options available for that class
    1. Option 1: Provide Implementation for all the `abstract` members inherited from the base abstract class.

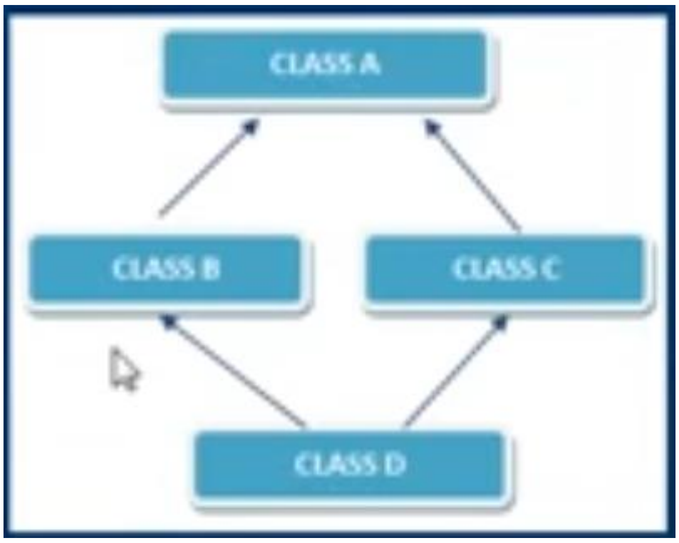
    2. Option 2: If the class does not wish to provide Implementation for all the abstract members inherited from the abstract class, then the class has to be marked as abstract.

Abstract Class vs Interface

- `abstract` classes **can have implementations** for some of its members (Methods), but the `interface` **can't have implementation** for any of its members.
- `Interfaces` cannot have **fields** where as an `abstract` class can have **fields**.
- An `interface` can inherit from another `interface` only and **cannot inherit from an** `abstract` class, whereas an `abstract` class **can inherit from another** `abstract` class or **another** `interface`.
- A class can **inherit** from multiple `interfaces` at the same time, whereas a class **cannot inherit from multiple classes** at the same time.
- `abstract` class members **can have access modifiers** , for the **non-abstract** fields whereas `interface` members **cannot have access modifiers**.

Problems in multiple inheritance

Class



```
using System;

class MultipleInheritance
{
    static void Main182()
    {

    }
}

public class A
{
    public virtual void Print()
    {
        Console.WriteLine("A Class");
    }
}


public class B : A
{
    public override void Print()
    {
        Console.WriteLine("B Class");
    }
}

public class C : A
{
    public override void Print()
    {
        Console.WriteLine("C Class");
    }
}

public class D : B,C   .
{
    public override void Print()
    {
        Console.WriteLine("D Class");
    }
}
```

1. Class B and Class C inherit from Class A.
2. Class D inherits from both B and C.
3. If a method in D calls a method defined in A (and does not override the method), and B and C have overridden that method differently, then from which class does it inherit: B, or C?

This ambiguity is called as Diamond problem

Inheritance

```
using System;

 interface IA
{
   void PrintA();

}
class ClassA : IA
{
    public void PrintA()
    {
        Console.WriteLine("A Class");
    }
}

interface IB
{
    void PrintB();

}
class ClassB : IB
{
    public void PrintB()
    {
        Console.WriteLine("B Class");
    }
}

//One way
class ClassAB : IA, IB
{
    public void PrintA()
    {
        Console.WriteLine("AB A Class");
    }

    public void PrintB()
    {
        Console.WriteLine("AB B Class");
    }
}
//Another way  -- Recommended
class ClassBA : IA, IB
{
    ClassA a = new ClassA();
    ClassB b = new ClassB();
    public void PrintA()
    {
        a.PrintA();
    }

    public void PrintB()
    {
        b.PrintB();
    }
}


class MultipleInterfaceInheritance
{
    static void Main()
    {
        ClassBA ba = new ClassBA();
        ClassAB ab = new ClassAB();

        ba.PrintA();   // OutPut    A class
        ba.PrintB();   // OutPut    B class

        ab.PrintA();   // OutPut    AB A class
        ab.PrintB();   // OutPut    AB B class

    }
}
```

Delegates

A Delegate is a **type safe function pointer**. Delegates are reference type. Type

Syntax:

Think of it as a method with a `delegate` keyword. The `delegate` can point to a method which should have **a similar signature.**
To point a `delegate` to a method, an instance of the delegate needs to be created, in this way, `delegates` are similar to `classes.` In the constructor of the declaration the method name should be passed. Then as the `delegates` is more like the function, while calling the `delegate` object, the parameter needs to be passed, which is/are defined in the function definition.

```
using System;

class Delegates
{
    public delegate void HelloFunctionDelegate(string message); // Definition of the
delegate

    public static void Main()
    {
        HelloFunctionDelegate hfd = new HelloFunctionDelegate(Hello); // Creating the
delegate object

        hfd("Delegates");    // Calling the delegate, with the parameter
    }

    public static void Hello(string message)  // This is the method, to which the
delegate will point. The delegate Should have same signature as the method. i.e.
return type void and a string parameter.

    {
        Console.WriteLine(message);
    }
}
```

A delegate is a type safe function pointer. That is, it holds a reference (Pointer) to a function.

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

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

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

Usage

Let us consider the following code. Here the Class Employee the method PromoteEmployee will print the ID and name of the employees who have experience more than or equal to 5, and are eligible for employees. The Class method is taking Employee List (Generic) as a parameter.

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


class Delegates
{
    public static void Main()
    {
        List<Employee> te = new List<Employee>();
        te.Add(new Employee() { ID=101,Name="Swapnadip Saha",Salary=10000,
Experience=5});
        te.Add(new Employee() { ID=102,Name="Arnab Neogy",Salary=8000, Experience=4});
        te.Add(new Employee() { ID=103,Name="Rohit Singh",Salary=12000,
Experience=6});
        te.Add(new Employee() { ID=104,Name="Koushik Roy",Salary=6000, Experience=8});
        te.Add(new Employee() { ID=105,Name="Antara Dey",Salary=7000, Experience=3});
        te.Add(new Employee() { ID=106,Name="Shantanu Bakshi",Salary=9000,
Experience=2});

        Employee.PromoteEmployee(te);
    }
}


class Employee
{
    public int ID { get; set; }
    public string Name { get; set; }
    public int Salary { get; set; }
    public int Experience { get; set; }

    public static void PromoteEmployee(List<Employee> employeeList)
    {
        foreach (Employee emp in employeeList)
        {
            if (emp.Experience >= 5)
            {
                Console.WriteLine("Eligible " + emp.ID + "-" + emp.Name);
            }
        }
    }
}
```

Now in this case, it may happen that the promotion eligible condition is changed to some other conditions. In that case the change will have to make in the main `Employee` class. This is not advisable, as making a framework type structures, like reusable class and not to hardcode, sometimes making changes into a class is not permittable. In that case we may use the concept of `delegates.`

Here the hard-coded logic is

```
if (emp.Experience >= 5)
```

this can be replaced by `delegate.`
The condition is return a `Boolean true or false.` So, the `delegate` should return a `bool.`

So, the delegate will be

```
delegate bool IsPromotableEmployee(Employee emp);
```

Now need to pass this delegate to the function `PromoteEmployee` as

```
    public static void PromoteEmployee(List<Employee> employeeList,
IsPromotableEmployee objIsEligible)
    {
        foreach (Employee emp in employeeList)
        {
            if (objIsEligible(emp))
            {
                Console.WriteLine("Eligible " + emp.ID + "-" + emp.Name);
            }
        }
    }
```

Since `delegates` are function pointer, so **effectively we are passing a function as a function parameter**.

Now need to pass this `delegate` parameter in the function calling in the Main function.

```
Employee.PromoteEmployee(te, objIsEligible);
```

Where `objIsEligible` is a `delegate` object. Now while declaring a `delegate` object, need to pass, to the constructor, a method , which has the same signature as of the `delegate`.

So, the function will be

```
    public static bool IsElligible(Employee emp)
    {
        if (emp.Experience >= 5)
        {
            return true;
        }
        else
        {
            return false;
        }

    }
```

And now the instance of the `delegate` will be as

```
IsPromotableEmployee objIsEligible = new IsPromotableEmployee(IsElligible);
```

Hence here is the full code,

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


class Delegates
{
    public static void Main()
    {
        List<Employee> te = new List<Employee>();  // Object of the Employee Class

       IsPromotableEmployee objIsEligible = new IsPromotableEmployee(IsElligible);
// Object of the delegate where the constructor has the method name as same signature
i.e. IsElligible

        te.Add(new Employee() { ID=101,Name="Swapnadip Saha",Salary=10000,
Experience=5});
        te.Add(new Employee() { ID=102,Name="Arnab Neogy",Salary=8000, Experience=4});
        te.Add(new Employee() { ID=103,Name="Rohit Singh",Salary=12000,
Experience=6});
        te.Add(new Employee() { ID=104,Name="Koushik Roy",Salary=6000, Experience=8});
        te.Add(new Employee() { ID=105,Name="Antara Dey",Salary=7000, Experience=3});
        te.Add(new Employee() { ID=106,Name="Shantanu Bakshi",Salary=9000,
Experience=2});

        Employee.PromoteEmployee(te, objIsEligible);   // Passing the Employee list
and the delegate
    }

    public static bool IsElligible(Employee emp)   // the delegate is pointing to this
method
    {
        if (emp.Experience >= 5)
        {
            return true;
        }
        else
        {
            return false;
        }

    }
}

delegate bool IsPromotableEmployee(Employee emp);   // the delegate
```

```
class Employee
{
    public int ID { get; set; }
    public string Name { get; set; }
    public int Salary { get; set; }
    public int Experience { get; set; }

    public static void PromoteEmployee(List<Employee> employeeList ,
IsPromotableEmployee IsEligible)   // The delegate object is passes
    {
        foreach (Employee emp in employeeList)
        {
              if (IsEligible(emp))   //Replaced the condition with delegate
            {
               Console.WriteLine("Eligible " + emp.ID + "-" + emp.Name);
            }
        }
    }
}
```

Lambda Function

Lambda functions are also, `delegates`. Instead of writing the method, to which the `delegate` is pointing and creating the `delegate` instance in the Main function we can use the Lambda function.

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

delegate bool IsPromotableEmployee(Employee emp);

class Employee
{
    public int ID { get; set; }
    public string Name { get; set; }
    public int Salary { get; set; }
    public int Experience { get; set; }

    public static void PromoteEmployee(List<Employee> employeeList,
IsPromotableEmployee IsEligible)
    {
        foreach (Employee emp in employeeList)
        {
            if (IsEligible(emp))
            {
                Console.WriteLine("Eligible " + emp.ID + "-" + emp.Name);
            }
        }
    }
}

class Delegates
{
    public static void Main()
    {
        List<Employee> oe = new List<Employee>();

        oe.Add(new Employee() { ID = 101, Name = "Swapnadip Saha", Salary = 10000,
Experience = 5 });
        oe.Add(new Employee() { ID = 102, Name = "Arnab Neogy", Salary = 8000,
Experience = 4 });
        oe.Add(new Employee() { ID = 103, Name = "Rohit Singh", Salary = 12000,
Experience = 6 });
```

```
        oe.Add(new Employee() { ID = 104, Name = "Koushik Roy", Salary = 6000,
Experience = 8 });
        oe.Add(new Employee() { ID = 105, Name = "Antara Dey", Salary = 7000,
Experience = 3 });
        oe.Add(new Employee() { ID = 106, Name = "Shantanu Bakshi", Salary = 9000,
Experience = 2 });

            Employee.PromoteEmployee(oe, emp => emp.Experience
>= 5);
    }
}
```

MultiCast Delegates

A Multicast `delegate` is a `delegate` that has **references to more than one function**. When you **invoke** a `multicast delegate`, **all the functions the delegate is pointing to, are invoked.**
There are 2 approaches to create a multicast `delegate`. Depending on the approach you use
+ or += to register a method with the `delegate`
– or -= to un-register a method with the `delegate`

Using +

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

namespace MulticastDelegates
{
    public delegate void MultiCastDelegate();
    class MulticastDelegatesClass
    {
        public static void Main()
        {
            MulticastDelegatesClass mcd = new MulticastDelegatesClass();

            MultiCastDelegate  del1,del2,del3,del4;

            del1 =new MultiCastDelegate(mcd.SampleDelegateMethodOne);   // the method
is not static, so calling using an instance of the class

            del2 = new MultiCastDelegate(SampleDelegateMethodTwo);      // the method
is  static, so calling directly

            del3 = new
MultiCastDelegate(MulticastDelegatesClass.SampleDelegateMethodThree);      // the
method is  static, so calling <Classname>.<Methodname>


            del4 = del1 + del2 + del3;    //Making del4 MultiCast

            del4();
        }
        public void SampleDelegateMethodOne()
        {
            Console.WriteLine("Sample Delegate Method One");
        }
        public static void SampleDelegateMethodTwo()
        {
            Console.WriteLine("Sample Delegate Method Two");
        }
        public static void SampleDelegateMethodThree()
        {
            Console.WriteLine("Sample Delegate Method Three");
        }

    }

}
```

```
//OUTPUT

Sample Delegate Method One

Sample Delegate Method Two

Sample Delegate Method Three
```

Using –

```
            del4 = del1 + del2 + del3 -del2;    //Making del4 MultiCast, unregistering
del2

            del4();
```

```
//OUTPUT

Sample Delegate Method One

Sample Delegate Method Three
```

Using + =

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

namespace MulticastDelegates
{
    public delegate void MultiCastDelegate();
    class MulticastDelegatesClass
    {
        public static void Main()
        {
            MulticastDelegatesClass mcd = new MulticastDelegatesClass();

            MultiCastDelegate del = new MultiCastDelegate(SampleDelegateMethodTwo);
            del += mcd.SampleDelegateMethodOne;
            del += SampleDelegateMethodThree;  // or
MulticastDelegatesClass.SampleDelegateMethodThree

            del();
        }
        public void SampleDelegateMethodOne()
        {
            Console.WriteLine("Sample Delegate Method One");
        }
        public static void SampleDelegateMethodTwo()
        {
            Console.WriteLine("Sample Delegate Method Two");
        }
        public static void SampleDelegateMethodThree()
        {
            Console.WriteLine("Sample Delegate Method Three");
        }

    }

}
//OUTPUT
```

```
Sample Delegate Method Two

Sample Delegate Method One

Sample Delegate Method Three
```

Using -=

```
            MultiCastDelegate del = new MultiCastDelegate(SampleDelegateMethodTwo);
            del += mcd.SampleDelegateMethodOne;
            del += SampleDelegateMethodThree;  // or
MulticastDelegatesClass.SampleDelegateMethodThree
            del -= MulticastDelegatesClass.SampleDelegateMethodThree;  // or
SampleDelegateMethodThree

            del();


//OUTPUT
```

```
Sample Delegate Method Two

Sample Delegate Method One
```

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

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

Delegate with return other than void

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

namespace MulticastDelegates
{
    public delegate void MultiCastDelegate();
    public delegate int MultiCastDelegateInt();
    class MulticastDelegatesClass
    {
        public static void Main()
        {
            MulticastDelegatesClass mcd = new MulticastDelegatesClass();

            MultiCastDelegateInt delInt, delInt1, delInt2;
            delInt1 = new MultiCastDelegateInt(mcd.SampleDelegateMethodIntOne);
            delInt2 = new MultiCastDelegateInt(SampleDelegateMethodIntTwo);
            delInt = delInt1 + delInt2;

            int delegateOutput=delInt();
            Console.WriteLine(delegateOutput);
        }

        public int SampleDelegateMethodIntOne()
        {
            return 1;
        }
        public static int SampleDelegateMethodIntTwo()
        {
            return 2;
        }
    }

}
```

//OUTPUT

2

Delegate without parameter

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

namespace MulticastDelegates
{
    public delegate void MultiCastDelegate();
    public delegate int MultiCastDelegateInt();
    public delegate int MultiCastDelegateOutParameter(out int delParam);
    class MulticastDelegatesClass
    {
        public static void Main()
        {
            MulticastDelegatesClass mcd = new MulticastDelegatesClass();

           MultiCastDelegateOutParameter delOutInt, delOutInt1, delOutInt2;
            delOutInt1 = new
MultiCastDelegateOutParameter(mcd.SampleDelegateMethodOutOne);
            delOutInt2 = new
MultiCastDelegateOutParameter(SampleDelegateMethodOutTwo);
            delOutInt = delOutInt2 + delOutInt1;


            int delegateOutOutPut, delegateOutParam = -1;
            delegateOutOutPut = delOutInt(out delegateOutParam);
            Console.WriteLine("Return the Output: " + delegateOutOutPut + " Output
Param: " + delegateOutParam);

 }
```

```
        public int SampleDelegateMethodOutOne(out int intParame)
        {
            intParame= 1;
            return intParame;
        }
        public static int SampleDelegateMethodOutTwo(out int intParame)
        {
            intParame= 2;
            return intParame;
        }
    }

}
```

//OUTPUT

1

Where do you use multicast delegates?

Multicast `delegate` makes implementation of **observer design pattern** very simple. Observer pattern is also called as **publish/subscribe pattern.**

Exception Handling

An exception is an unforeseen error that occurs when a program is running.

Examples:

Trying to read from a file that does not exist, throws FileNotFoundException. Trying to read from a database table that does not exist, throws a SqlException.

Showing actual unhandled exceptions to the end user is bad for two reasons

1. Users will be annoyed as they are cryptic and does not make much sense to the end users.

2. Exceptions contain information, that can be used for hacking into your application

An exception is actually a class that derives from System.Exception class. The System.Exception class has several useful properties, that provide valuable information about the exception.

Message: Gets a message that describes the current exception

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

Cannot catch or handle compile-time errors programmatically because they prevent the application from being compiled in the first place.

Examples of Compile-Time Errors:

Syntax errors (e.g., missing semicolon, wrong variable declaration)

Type errors (e.g., trying to assign a string to an integer)

Missing references (e.g., forgetting to include a using statement)

Releasing System Resources

We use try, catch and finally blocks for exception handling:

try - The code that can possibly cause an exception will be in the try block.

catch - Handles the exception.

finally - Clean and free resources that the class was holding onto during the program execution. Finally, block is optional. But it is a good practice to keep the finally block. It may sometime happen that, there may some exceptions in one of the specific catch blocks, in that case other catch blocks will not be executed. In that case only the finally blocks would execute.

Note: It is a good practice to always release resources in the finally block, because finally block is guaranteed to execute, irrespective of whether there is an exception or not.

Specific exceptions will be caught before the base general exception, so specific exception blocks should always be on top of the base exception block. Otherwise, you will encounter a compiler error.

```
using System;


namespace ExceptionHandiling
{
    class ExceptionHandiling
    {
        public static void Main()
        {
            try
            {
                string[] newArray = new string[3];

                newArray[0] = "Element 1";
                newArray[1] = "Element 2";
                newArray[2] = "Element 3";
                newArray[3] = "Element 4";
            }

            //Specific Exception

            catch (IndexOutOfRangeException ex)
            {
                Console.WriteLine("The exception is "+ex.Message);
                Console.WriteLine(ex.Message);
                Console.WriteLine();
                Console.WriteLine();
                Console.WriteLine(ex.StackTrace);

            }
```

```
          //General Exception

            catch (Exception ex)
// Here it throws "Index was outside the bounds of the array."  This should be the
last one if there are more than exception catch blocks. All the exceptions will be
catch here. If there are more than one catch blocks, and the exception is already
catching by one of the catch blocks, then the general exception block will not work.
            {
                Console.WriteLine("The exception is " + ex.Message);
                Console.WriteLine(ex.Message);
                Console.WriteLine();
                Console.WriteLine();
                Console.WriteLine(ex.StackTrace);
            }

        }

    }

}
```

Inner Exceptions

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

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

Always check if inner exception is not null before accessing any property of the inner exception object, else, you may get Null Reference Exception

To get the type of InnerException use GetType() method

For example, let us consider the following example—

```
using System;
using System.IO;
using System.Net;
using System.Net.Sockets;
using System.Linq;

namespace ExceptionHandiling
{

class InnerException
    {
        public static void Main()
        {

            try
            {
                Console.WriteLine("Enter First Number");
                int fn = Convert.ToInt32(Console.ReadLine());
                Console.WriteLine("Enter Second Number");
                int sn = Convert.ToInt32(Console.ReadLine());
                Console.WriteLine("Sum is {0} ", fn + sn);
            }
            catch (Exception ex)
            {

                string hostName = Dns.GetHostName();   //for HostName
                IPAddress ipAddresses =
Dns.GetHostAddresses(hostName).FirstOrDefault(ip => ip.AddressFamily ==
AddressFamily.InterNetwork); ;    //for IP Address

                string filePath = @"D:\472124\T Factor New\PragimTech\Module 2\1. C#
Tutorial for Beginners\PragimTech1 csharp\PragimTech csharp\Contents\LogFile.txt";
                if (File.Exists(filePath))
                {
                    StreamWriter sw = new StreamWriter(filePath, true);   // true will
append. Only filepath will rewrite
                    sw.WriteLine();
                    sw.WriteLine(DateTime.Now + " - " + ex.GetType().Name + " - " +
hostName + " - " + ipAddresses);
                    sw.WriteLine(DateTime.Now + " - " + ex.Message);
                    sw.Close();
                }
                else
                {
                    throw new FileNotFoundException(filePath + " is not present", ex);
                }

            }
        }


    }
}
```

In the above code, user inputs two integers fn and sn. Now if the user inputs a string or a number which out of range of integer, then it will throw an exception and write it into the log file. If the log file does not exist, it will throw the

```
              throw new FileNotFoundException(filePath + " is not present", ex);
```

and probably the file code will struct. This is actually the exception inside the exception. Now to solve this issue, the entire code can be written inside another try catch block.

```
using System;
using System.IO;
using System.Net;
using System.Net.Sockets;
using System.Linq;

namespace ExceptionHandiling
{

    class InnerException
    {
        public static void Main()
        {
            try
            {
                try
                {
                     …………………
                     else
                     {
                            throw new FileNotFoundException(filePath + " is not
                     present", ex);
                     }

                   }
            catch (Exception ex1)
            {

                Console.WriteLine("Current Exception:"+ex1.GetType().Name);
                Console.WriteLine("Inner
Exception:"+ex1.InnerException.GetType().Name);
            }

        }
    }
}
```

Now the exception throws from the main Try catch block, i.e. related to the two integers and their calculations, **this will be the inner exception, as this is from inner try catch block**. And the file not found will be the **current exception as this throws from the current try catch block**.
Now if we write like this

```
                               throw new FileNotFoundException(filePath + " is not
                        present");
```

i.e. **if the exception variable is not passed to the current exception** then a `null` will pass to the current exception from inner exception, it will be an error as null dose not have
`.GetType().Name`
property.

Custom Exceptions

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

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

Example:

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

Within the.NET framework we don't have any exception, that adequately describes this problem. So, this scenario is one of the examples where you want to create a custom exception.

List of Exceptions

Debug → Windows → Exception Settings **Ctrl + Alt + E**

Example

```
using System;
using System.IO;
using System.Net;
using System.Net.Sockets;
using System.Linq;


namespace ExceptionHandiling
{
    class CustomException
    {
        public static void Main()
        {
            try
            {
                Exception ex = new Exception();
                throw new UserDefinedCustomException();
                throw new UserDefinedCustomException("User defined Exceptions");
                throw new UserDefinedCustomException("User defined Exceptions and 
Inner exceptions", ex);
            }

            catch (UserDefinedCustomException ex)
            {
                Console.WriteLine(ex.Message);
            }

        }
    }
    class UserDefinedCustomException : Exception
    {
        public UserDefinedCustomException() : base()
        {
            //the default constructor. Calling from the Base
        }
        public UserDefinedCustomException(string message) : base(message)
        {
            //the parameterized constructor. Calling from the Base
        }
        public UserDefinedCustomException(string message, Exception InnerException) : 
base(message, InnerException)
        {
            //the parameterized constructor, with InnerException. Calling from the 
Base
        }

    }
}
```

Now the Custom exception **class** can be applicable in current Application domain only. Let say there are two applications App A and App B. Now the classes and objects of App A cannot be directly accessed by App B. To move objects across application boundaries his we need **Serialization**. **Serialization** is like breaking down the objects into packets that can be transmitted over the network.

Here to make work `UserDefinedCustomException` **Serializable** we need to this,

```
using System.Runtime.Serialization;

        public UserDefinedCustomException(SerializationInfo info, StreamingContext 
context) : base(info, context)
        {

        }
```

Steps to create Custom Exception

1. Create a class that derives from System.Exception class. As a convention, end the class name with Exception suffix. All.NET exceptions end with, exception suffix.
2. Provide a public constructor, that takes in a string parameter. This constructor simply passes the string parameter, to the base exception class constructor.
3. Using InnerExceptions, you can also track back the original exception. If you want to provide this capability for your custom exception class, then overload the constructor accordingly.
4. If you want your Exception class object to work across application domains, then the object must be serializable. To make your exception class serializable mark it with Serializable attribute and provide a constructor that invokes the base Exception class constructor that takes in SerializationInfo and StreamingContext objects as parameters.

Exception Handling Abuse

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

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

Example

```
using System;
using System.IO;
using System.Net;
using System.Net.Sockets;
using System.Linq;
using System.Runtime.Serialization;


    class ExceptionAbuse
    {
        public static void  Main()
        {
            try
            {
                Console.WriteLine("Enter Numerator");
                int fn = Convert.ToInt32(Console.ReadLine());
                Console.WriteLine("Enter Denominator");
                int sn = Convert.ToInt32(Console.ReadLine());
                Console.WriteLine("Sum is {0} ", fn / sn);
            }
            catch (DivideByZeroException)
            {
                Console.WriteLine("Division by zero is not accepted");
            }
            catch (FormatException)
            {
                Console.WriteLine("Enter a number");
            }
            catch (OverflowException)
            {
                Console.WriteLine("Numbers should be between {0} and {1}.",
Int32.MinValue, Int32.MaxValue);
            }
            catch(Exception ex)
            {
                Console.WriteLine(ex.Message);
            }
        }
    }
```

Now instead doing so many exceptions (This is the exception abuse), try to make implement the logic as many as possible in the code it self

```
class ExceptionAbuseSolved
    {
        public static void Main()
        {
            try
            {
                int fn, sn;
                Console.WriteLine("Enter Numerator");
                fn = Convert.ToInt32(Console.ReadLine());
                bool isNumerator = Int32.TryParse(Console.ReadLine(), out fn);

                if (isNumerator)
                {
                    Console.WriteLine("Enter Denominator");
                    sn = Convert.ToInt32(Console.ReadLine());
                    bool isDenominator = Int32.TryParse(Console.ReadLine(), out sn);
                    if (isDenominator && sn != 0)
                    {
                        Console.WriteLine("Quotient {0}", fn / sn);
                    }
                    else
                    {
                        if (sn == 0)
                        {
                            Console.WriteLine("Division by zero is not accepted");
                        }
                        else
                        {
                            Console.WriteLine("Enter a number for Denominator between
{0} and {1}", Int32.MinValue, Int32.MaxValue);
                        }

                    }
                }
                else
                {
                    Console.WriteLine("Enter a number for Numerator between {0} and
{1}", Int32.MinValue, Int32.MaxValue);
                }


            }

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

Enums

Enums are strongly typed constants.
If a program uses set of integral numbers, consider replacing them with enums. Otherwise, the program becomes less

- Readable
- Maintainable

Let us consider the following code, without enums

Without enums

If a program uses set of integral numbers, consider replacing them with enums. Otherwise, the program becomes less

- Readable
- Maintainable

```
using System;

namespace Enums
{
    class WithoutEnums
    {
        public static void Main()
        {
            Customer[] customer = new Customer[3];
            customer[0] = new Customer { Name="SS", Gender=0 };
            customer[1] = new Customer { Name="EC", Gender=1 };
            customer[2] = new Customer { Name="AS", Gender=2 };
            foreach (Customer cc in customer)
            {
                Console.WriteLine("Name-{0} and Gender {1}.",cc.Name,cc.Gender);
            }
            ///output
            ///Name-SS and Gender 0.
            ///Name - EC and Gender 1.
            ///Name - AS and Gender 2.
            /// ----------------------------- ----------------------------
            /// ----------------------------- ----------------------------
            /// ----------------------------- ----------------------------
            ///This does not make sense, as every time we have to check the
documentation, for the definition of 0,1,2
            ///or
            ///We may have a function, which converts the int into string
like............

            foreach (Customer cc1 in customer)
            {
                Console.WriteLine("Name-{0} and Gender {1}.", cc1.Name,
WithoutEnums.getGender(cc1.Gender));
            }
        }
        public static string getGender(int gender)
        {
            switch (gender)
            {
                case 0:
                    return "Unknown";
                case 1:
                    return "Male";
                case 2:
                    return "Female";
                default:
                    return "No Idea";
            }
        }
    }
    class Customer
    {
        /// <summary>
        /// 0 for Unknown
        /// 1 for MALE
        /// 2 for FEMALE
        /// </summary>
        public string Name { get; set; }
        public int Gender  { get; set; }
    }
}
```

This is a lot of code and less readable.

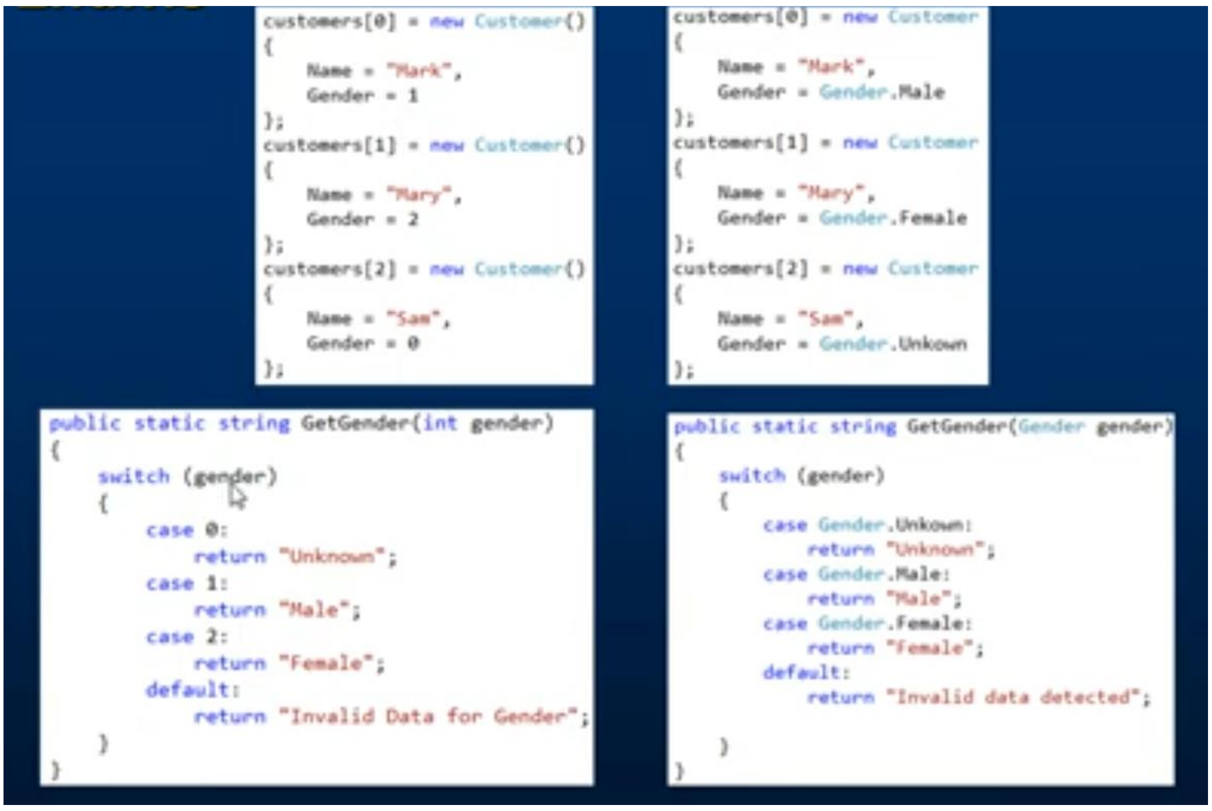
With enums

Enums are strongly typed constants.

If a program uses set of integral numbers, consider replacing them with enums, which makes the program more

- Readable
- Maintainable

```
using System;

namespace Enums
{
    class Enums
    {
        public static void Main()
        {
            Customer[] customer = new Customer[3];
            customer[0] = new Customer { Name = "SS", Gender = Gender.Unknown };
            customer[1] = new Customer { Name = "EC", Gender = Gender.Male };
            customer[2] = new Customer { Name = "AS", Gender = Gender.Female };
            foreach (Customer cc1 in customer)
            {
                Console.WriteLine("Name-{0} and Gender {1}.", cc1.Name, cc1.Gender);
            }
        }

        /// <summary>
        /// Optional In case of enums
        /// </summary>
        /// <param name="gender"></param>
        /// <returns></returns>
        public static string getGender1(Gender gender)
        {
            switch (gender)
            {
                case Gender.Unknown:
                    return "Unknown";
                case Gender.Male:
                    return "Male";
                case Gender.Female:
                    return "Female";
                default:
                    return "No Idea";
            }
        }
    }

    public enum Gender
    {
        Unknown,
        Male,
        Female
    }

    class Customer
    {
        public string Name { get; set; }
        public Gender Gender { get; set; }
    }
}
```



```
customers[0] = new Customer()
{
    Name = "Mark",
    Gender = 1
};
customers[1] = new Customer()
{
    Name = "Mary",
    Gender = 2
};
customers[2] = new Customer()
{
    Name = "Sam",
    Gender = 0
};
```

```
customers[0] = new Customer
{
    Name = "Mark",
    Gender = Gender.Male
};
customers[1] = new Customer
{
    Name = "Mary",
    Gender = Gender.Female
};
customers[2] = new Customer
{
    Name = "Sam",
    Gender = Gender.Unkown
};
```

```
public static string GetGender(int gender)
{
    switch (gender)
    {
        case 0:
            return "Unknown";
        case 1:
            return "Male";
        case 2:
            return "Female";
        default:
            return "Invalid Data for Gender";
    }
}
```

```
public static string GetGender(Gender gender)
{
    switch (gender)
    {
        case Gender.Unkown:
            return "Unknown";
        case Gender.Male:
            return "Male";
        case Gender.Female:
            return "Female";
        default:
            return "Invalid data detected";
    }
}
```

1. Enums are enumerations.
2. Enums are strongly typed constants. **Hence, an explicit cast is needed to convert from enum type to an integral type and vice versa**. **Also, an enum of one type cannot be implicitly assigned to an enum of another type even though the underlying value of their members are the same**.

```
            Gender gender = (Gender)3;
             int n=(int) Gender.Unknown;
```

Cannot implicitly convert. Need to perform explicit conversion. Again

```
public enum Gender
{
    Unknown = 1,
    Male = 2,
    Female = 3
}

public enum Season
{
    Summer = 1,
    Winter = 2,
    Spring = 3
}

public static void Main()
 {
    Gender g = new Gender();
     Season g1 = new Season();

   g=g1;    //or
   Gender g2 = Season.Summer;

   //These are errors. We need explicit Conversion like

        g = (Gender)g1;
        //or
        Gender g2 = (Gender)Season.Summer;
 }
```

3. The default underlying type of an enum is int.
4. The default value for first element is ZERO and gets incremented by 1.
5. It is possible to customize the underlying type and values.
6. Enums are value types.
7. enum keyword (all small letters) is used to create enumerations, whereas **Enum** class, contains static **GetValues()** and **GetNames()** methods which can be used to list Enum underlying type values and Names.

```
public enum Gender
{
    Unknown,
    Male,
    Female
}

    public static void Main()
    {

        Console.WriteLine(typeof(Gender));
        int[] vals = (int[])Enum.GetValues(typeof(Gender));// the GetValues()
method is expecting a type to be passed in. A type can be a class, struct, enum etc.
Here we are passing the name of the enum i.e. Gender. But as the parameter is type we
are passing typeof(Gender)=Enums.Gender. As the return type of GetValues() array of
values, hence int[]
            foreach (int v in vals)
            {
                Console.WriteLine(v.ToString());
            }

            string[] name = Enum.GetNames(typeof(Gender));
            foreach (string n in name)
            {
                Console.WriteLine(n.ToString());
            }
        }
              //OUTPUT

              Enums.Gender
              0
              1
              2
              Unknown
              Male
              Female
```

The GetValues() method is expecting a `type` to be passed in. A type can be a class, struct, enum etc. Here we are passing the name of the enum i.e. Gender. But as the parameter is type we are passing `typeof(Gender)=` Enums.Gender. As the return type of GetValues() array of values, hence `int[]`.

In the `enum`, since nothing is specified, hence the underlying values are integer and starting from 0. But the default type can be changed.

```
    public enum Gender :short
    {
        Unknown,
        Male,
        Female
    }
```

Now the underlying values are short type. In this case.
```
short[] vals = (short[])Enum.GetValues(typeof(Gender));

     foreach (short v in vals)
      {
         Console.WriteLine(v.ToString());
      }
```

Also, it is possible to change the default values like

```
    public enum Gender :short
    {
        Unknown=1,
        Male=7,
        Female=99
    }

              //OUTPUT

              Enums.Gender
              1
              7
```

```
99
Unknown
Male
Female
```

Difference between Types and type numbers

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

Type members can have all the access modifiers, whereas types can have only 2 (internal, public) of the 5 access modifiers

Note: Using regions you can expand and collapse sections of your code either manually, or using visual studio Edit -> Outlining-> Toggle All Outlining

Summary of Differences:

Concept	Definition	Purpose	Output/Usage
Namespace	Logical container to organize classes and types	Avoids naming conflicts and structures code	`using MyNamespace`
Assembly	Compiled code library in the form of a DLL or EXE	Holds compiled code and metadata	`.dll` or `.exe` file
DLL	Dynamic Link Library	Contains reusable code for multiple apps	Cannot be executed directly; shared by apps
EXE	Executable file	A standalone application	Directly executable by the OS

Access Modifiers

In C# there are 5 different access modifiers:

1. Private
2. Protected
3. Internal
4. Protected Internal
5. Public
6. Private protected

Private members are available only with in the containing type. Not even from the child classes. **Only for types but not for type members**.

Public members are available anywhere. There is no restriction. **For both types and type members**

Protected Members are available, with in the containing type and to the types that derive from the containing type. Like. the child classes of the parent class. **Only for types but not for type members**

Internal available anywhere within the containing assembly. It's a compile time error to access, an internal member from outside the containing assembly. **For both types and type members**

Protected Internal can be accessed by any code in the assembly in which it is declared, or from within a derived class in another assembly. It is a combination of protected and internal. If you have understood protected and internal, this should be very easy to follow. **Only for types but not for type members**

private protected Only code in the same assembly and in the same class or a derived class can access the type or member. **Only for types but not for type members**

	Containing class	Derived types within current assembly	Current assembly	Derived types	Entire program
public	Yes	Yes	Yes	Yes	Yes
private	Yes	No	No	No	No
internal	Yes	Yes	Yes	No	No
protected	Yes	Yes	No	Yes	No
protected internal	Yes	Yes	Yes	Yes	No
private protected	Yes	Yes	No	No	No

Modifier	Description
public	There are no restrictions on accessing public members.
private	Access is limited to within the class definition. This is the default access modifier type if none is formally specified
internal	Access is limited exclusively to classes defined within the current project assembly
protected	Access is limited to within the class definition and any class that inherits from the class
protected internal	Access is limited to the current assembly and types derived from the containing class. All members in the current project and all members in the derived class can access the variables.
private protected	Access is limited to the containing class or types derived from the containing class within the current assembly.

Attributes

Attributes allow you to add declarative information to your programs. This information can then be queried at runtime using reflection.

There are several Pre-defined Attributes provided by.NET. It is also possible to create your own Custom Attributes.

A few pre-defined attributes with in the.NET framework:

- **Obsolete** -Marks types and type members outdated
- **WebMethod** - to expose a method as an XML Web service method
- **Serializable** - Indicates that a class can be serialized

It is possible to **customize the attribute using parameters.**

An attribute is a class that inherits from **System.Attribute** base class.

Now let’s assume for the below code

```
using System;

namespace Attributes
{
    class Attributes
    {
        public static void Main()
        {

            Console.WriteLine(Calculator.Add(5, 6));
            Console.WriteLine(Calculator.Add(5, 6,7));
        }
    }

    class Calculator
    {
        public static int Add(int a, int b)
        {
            return a + b;
        }
              public static int Add(int a, int b, int c)
        {
            return a + b + c;
        }


    }

}
```

To add multiple number, if the number of parameters passing to the

```
          public static int Add()
```

is not fixed, then we need to overload the method with number of increasing parameters. But that can be updated by passing `params` or List of integers. By doing this the classical methods will not in use for any longer. But it may happen that some methods are still pointing to this method. In that case, instead removing the method, mark it as `[Obsolete]`. Hence the old method will be still in use, and new developers can easily understand to use which methods. These `[Obsolete]`, `[WebMethod]` are called Attributes.

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

namespace Attributes
{
    class Attributes
    {
        public static void Main()
        {
            List<int> la = new List<int>() { 9,10};



            int[] a =new int[2];
            a[0] = 7;
            a[1] = 8;
            Console.WriteLine(Calculator.Add(a));
            Console.WriteLine(Calculator.Add(5, 6));
            Console.WriteLine(Calculator.Add(la));

        }
    }

    class Calculator
    {
        [Obsolete]
        public static int Add(int a, int b)
        {
            return a + b;
        }
        [Obsolete]
        public static int Add(params int[] a)
        {
            int sum = 0;
            foreach (int i in a)
            {
                sum += i;
            }
            return sum;
        }

        public static int Add(List<int> a)
        {
            int sum = 0;
            foreach (int i in a)
            {
                sum += i;
            }
            return sum;
        }
    }

}
```

Parametrized Attributes

Now while building the solution, it will give warnings that,

```
'Calculator.Add(params int[])' is obsolete
'Calculator.Add(int, int)' is obsolete
```

So, we need to tell the developer, which method needs to be used. Bu=y adding some message to the methods as

```
[Obsolete("Use public static int Add(List<int> a)")]
```

Now the tooltip, where the methods called in Main() will show this message. And now we can change these methods accordingly.

```
        public static void Main()
        {
            List<int> la = new List<int>() { 9,10};
            int[] a =new int[2];
            a[0] = 7;
            a[1] = 8;
            Console.WriteLine(Calculator.Add(new List<int>() { 5, 6 }));
            Console.WriteLine(Calculator.Add(new List<int>() { a[0], a[1]}));
            Console.WriteLine(Calculator.Add(la));

        }
```

Note: Still the old can be used. To prevent from using the `[Obsolete]` method, we need to use,

```
[Obsolete("Use public static int Add(List<int> a)"),true]
```

`Now, using the old methods will give an error. Bu default it is false.`

Reflection -- Need Again

Reflection is the ability of inspecting an assembly metadata at runtime. It is used to find all types in an assembly and/or dynamically invoke methods in an assembly.

```
using System;

namespace Reflections
{
    /// <summary>
    /// There are two classes Reflections and Customer. While building this console
application, these classes will be compiled into an IL
    /// and packaged into an assembly (.exe or  .dll). An assembly has two parts one
is the IL and the other one is Metadata, which contains the name of the class
    /// the types, members etc of the class(or structs, enums)
    /// </summary>
    class Reflections
    {
        public static void Main()
        {

        }
    }
    class Customer
    {
        public int id { get; set; }
        public string name { get; set; }
        public Customer()
        {
            this.id = -1;
            this.name = string.Empty;
        }

        public Customer(int Id, string Name)
        {
            this.id = Id;
            this.name = Name;
        }

        public void PrintID()
        {
            Console.WriteLine("Id is {0}", this.id);
        }
        public void PrintName()
        {
            Console.WriteLine("Name is {0}", this.name);
        }
    }
}
```

Uses of reflection:

1. When you drag and drop a button on a win forms or an asp.net application. The properties window uses reflection to show all the properties of the Button class. So, reflection is extensively used by IDE or a UI designer.

```
  public static void Main()
        {
            // We want write the list of elements (properties, methods, constructors, 
fields etc)

            Type T = Type.GetType("NSReflections.Customer"); //GetType() is a static 
method of Type class. this method will get the type of the class i.e. 
<Namespace>.<Class>

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

Console.WriteLine("Full Name of Type:" + T.FullName
Console.WriteLine("Name of Type:" + T.Name);
Console.WriteLine("Assembly Qualified Name of Type:" + T.AssemblyQualifiedName);
Console.WriteLine("Namespace of Type:" + T.Namespace);
Console.WriteLine("BaseType of Type:" + T.BaseType);


//OUTPUT
------------Type Description---------------
Full Name of Type:NSReflections.Customer
Name of Type:Customer
Assembly Qualified Name of Type:NSReflections.Customer, PragimTech csharp, 
Version=1.0.0.0, Culture=neutral, PublicKeyToken=null
Namespace of Type:NSReflections
BaseType of Type:System.Object


       Console.WriteLine("------------Properties of the Types---------------");

PropertyInfo[] props = T.GetProperties(); // The GetProperty returns n array of 
PropertyInfo from using System.Reflection;
   foreach (PropertyInfo pro in props)
            {
                Console.WriteLine("Property Name -" + pro.Name + ". Property Type  -" 
+ pro.PropertyType + ". Property Can Read  -" + pro.CanRead);
            }


//OUTPUT
------------Properties of the Types---------------
Property Name -id. Property Type  -System.Int32. Property Can Read  -True
Property Name -name. Property Type  -System.String. Property Can Read  -True


            Console.WriteLine("------------Methods of the types---------------");
            MethodInfo[] meth = T.GetMethods(); // The GetMethods returns n array of 
MethodInfo from using System.Reflection;
            foreach (MethodInfo met in meth)
            {
                Console.WriteLine("Method Name -" + met.Name + ". Return Types -" + 
met.ReturnType + ". Parameters- " + met.GetParameters() + ". Is Static -" + 
met.IsStatic + ". DeclaringType -" + met.DeclaringType);
                Console.WriteLine();
            }

//OUTPUT
------------Methods of the types---------------
Method Name -get_id. Return Types -System.Int32. Parameters-
System.Reflection.ParameterInfo[]. Is Static -False. DeclaringType -
NSReflections.Customer
```

```
Method Name -set_id. Return Types -System.Void. Parameters-
System.Reflection.ParameterInfo[]. Is Static -False. DeclaringType -
NSReflections.Customer


Method Name -get_name. Return Types -System.String. Parameters-
System.Reflection.ParameterInfo[]. Is Static -False. DeclaringType -
NSReflections.Customer

Method Name -set_name. Return Types -System.Void. Parameters-
System.Reflection.ParameterInfo[]. Is Static -False. DeclaringType -
NSReflections.Customer

Method Name -PrintID. Return Types -System.Void. Parameters-
System.Reflection.ParameterInfo[]. Is Static -False. DeclaringType -
NSReflections.Customer

Method Name -PrintName. Return Types -System.Void. Parameters-
System.Reflection.ParameterInfo[]. Is Static -False. DeclaringType -
NSReflections.Customer

Method Name -GetType. Return Types -System.Type. Parameters-
System.Reflection.ParameterInfo[]. Is Static -False. DeclaringType -
System.Object

Method Name -ToString. Return Types -System.String. Parameters-
System.Reflection.ParameterInfo[]. Is Static -False. DeclaringType -
System.Object

Method Name -Equals. Return Types -System.Boolean. Parameters-
System.Reflection.ParameterInfo[]. Is Static -False. DeclaringType -
System.Object

Method Name -GetHashCode. Return Types -System.Int32. Parameters-
System.Reflection.ParameterInfo[]. Is Static -False. DeclaringType -
System.Object
```

Every `type` in .net are directly or indirectly inherits `System. Object.` `System. Object` has four methods

- Equals
- GetHashCode
- GetType
- ToString

These methods are also returning by the `MethodInfo[] meth = T.GetMethods();`

```
    Console.WriteLine("------------Fields of the types---------------");

FieldInfo[] field = T.GetFields(BindingFlags.Public | BindingFlags.NonPublic |
BindingFlags.Instance | BindingFlags.Static);

// The GetFields returns n array of FieldInfo from using System.Reflection; By
default, GetFields() only returns public fields. If the class T has no public fields,
nothing will be displayed. You can retrieve all fields (public and non-public) by
modifying the GetFields call like this.
```

```
            foreach (FieldInfo fld in field)
            {
                Console.WriteLine("Attributes Name -" + fld.Attributes + ". Field Name 
-" + fld.Name + ". fld.ToString()- " + fld.ToString() + ". Declaring Type -" + 
fld.DeclaringType);
            }
```

```
//OUTPUT
------------Fields of the types---------------
Attributes Name -Private. Field Name -<id>k__BackingField. fld.ToString()- 
Int32 <id>k__BackingField. Declaring Type -NSReflections.Customer

Attributes Name -Private. Field Name -<name>k__BackingField. fld.ToString()- 
System.String <name>k__BackingField. Declaring Type -NSReflections.Customer
```

```
            Console.WriteLine("------------Constructors of the types---------------");
            ConstructorInfo[] ctor = T.GetConstructors(); // The GetMethods returns n 
array of MethodInfo from using System.Reflection;
            foreach (ConstructorInfo cons in ctor)
            {
                Console.WriteLine("Is Public -" + cons.IsPublic + ". Parameters- " + 
cons.GetParameters() + ". DeclaringType -" + cons.DeclaringType + ". Name);
            }
        }
```

```
//OUTPUT
------------Constructors of the types---------------
Is Public -True. Parameters- System.Reflection.ParameterInfo[]. DeclaringType -
NSReflections.Customer
Is Public -True. Parameters- System.Reflection.ParameterInfo[]. DeclaringType -
NSReflections.Customer
Here Nothing is clear about the constructor. Hence use
```

```
                Console.WriteLine(cons.ToString());
```

This will return

```
//OUTPUT
Void .ctor()
Void .ctor(Int32, System.String)
```

Since the constructor name is same as Class name, hence the number and type of parameters are important.

Note: Instead of using

```
                            Type T = Type.GetType("NSReflections.Customer");
```

we can also use

```
                            Type T = typeof(Customer);
```

Or

```
                            Customer c1 = new Customer();
                            Type T = c1.GetType();
```

Since all the objects do have the access to `GetType()` **from there** `base` **i.e.** `System.Objects`**;**

```
                            public Type GetType();
```

2. Late binding can be achieved by using reflection. You can use reflection to dynamically create an instance of a type, about which we don't have any information at compile time. **So, reflection enables you to use code that is not available at compile time.**

   Let say,

```
Customer c1 = new Customer();
```

this instantiation of `Customer` was done while compiling the code, hence this is an early binding. Late binding refers to , if the instantiation is done at run time/dynamically. When maybe we done have any information about the class, which needs to be instantiation.

3. Consider an example where we have two alternate implementations of an interface. You want to allow the user to pick one or the other using a config file. With reflection, you can simply read the name of the class whose implementation you want to use from the config file, and instantiate an instance of that class. This is another example for **late binding using reflection.**

Generics

- Generics are introduced in C# 2.0. Generics allow us to design classes and methods decoupled from the data types.

Let say for the following Code

```
class Generics
{
    public static void Main()
    {
        bool Equal = Calculator.AreEqual(1, 2);
        if (Equal)
        {
            Console.WriteLine("Equal");
        }
        else
        {
            Console.WriteLine("Not Equal");
        }
    }
}
public class Calculator
{
    public static bool AreEqual(int val1, int val2)
    {
        return val1 == val2;
    }
}
```

Here `AreEqual()` takes two integer parameters. So, while calling the function from Main(), providing anything other than integer, it will throw an error. As the method `AreEqual()` is strongly coupled with the datatypes.

One way to solve this issue is

```
public class Calculator
{
    public static bool AreEqual(object val1, object val2)
    {
        return val1 == val2;
    }
}
```

As integer, string etc all are directly or indirectly inherits the property of System.Object. So the method can be reused for any type of datatypes. But in this case there are two problems
First

```
bool Equal = Calculator.AreEqual("A", 2);
```

This will not throw any error as this the both are `object` is reference type.

Second

```
bool Equal = Calculator.AreEqual(1, 2);
```

Here the integers are structure that is value types and the `object` are reference type. So a runtime conversion or boxing/unboxing happens Which degrade the execution.

So the best way to make independent of datatypes is to use **Generics.**

We can use

```
public static bool AreEqual <T>(T val1, T val2)
{
        return val1.Equals(val2);
}
```

Here `<T>` indicates type. While calling the function we can use any types and reuse the function. But the arguments should be all of same type. So, the Boxing/unboxing will never happen.

```
bool Equal = Calculator.AreEqual<string>("A", "B");

if (Equal)
{
    Console.WriteLine("Equal");
}
else
{
    Console.WriteLine("Not Equal");
}
```

Or

```
bool Equal = Calculator. AreEqual <int>(1, 2);

if (Equal)
{
    Console.WriteLine("Equal");
}
else
{
    Console.WriteLine("Not Equal");
}
```

Also ,we can make the Class as Generic.

```
public class Calculator<T>
{
    public static bool AreEqual(T val1, T val2)
    {
        return val1.Equals(val2);
    }
}
```

```
bool Equal = Calculator<int>.AreEqual(1, 2);
```

- Generic classes are extensively used by collection classes available in `using System.Collections.Generic;` namespace.

- One way of making `AreEqual` () method reusable, is to use object type parameters. Since, every type in .NET directly or indirectly inherit from `System.Object` type, `AreEqual` () method works with any data type, but the problem is performance degradation due to boxing and unboxing happening.

- Also, `AreEqual` () method is no longer type safe. It is now possible to pass integer for the first parameter, and a string for the second parameter. It doesn't really make sense to compare strings with integers.

So, the problem with using System.Object type is that

1. `AreEqual` () method is not type safe

2. Performance degradation due to boxing and unboxing

ToString() Override

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

namespace ToStringOverride
{
    class ToStringOverride
    {
        public static void Main()
        {
            int p = 10;
            Console.WriteLine(p.ToString());

            Customer c1 = new Customer();
            c1.FirstName = "Swapnadip";
            c1.LastName = "Saha";
            Console.WriteLine(c1.ToString());
        }
    }
    public class Customer
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
    }

}
```

In this code the first Output will be 10 as, it is a simple datatype and can be converted to `string`. But the second Output will be the type name of `Customer` i.e. `ToStringOverride.Customer (<NameSpaceName.ClassName>)`

So, we can `override` the `ToString()` Function as we required.

```
public string ToString()
{
    return base.ToString();
}
```

This will return the base type, i.e. the original `ToString()`

```
public override string ToString()
{
    return this.FirstName + " " + this.LastName;
}
```

This will `override` the `ToString()` and output will be `Swapnadip Saha`. Or it can be customized as required. Like `return this.LastName + " " + this.FirstName;`

Again `Console.WriteLine(Convert.ToString(c1));` this also can be written instead of `Console.WriteLine(c1.ToString());`

Equals() Override

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

namespace ToStringEqualsOverride
{
    class ToStringEqualsOverride
    {
        public static void Main()
        {

        int i = 10;
        int j = 10;
        Console.WriteLine(i == j);             //Output True
        Console.WriteLine(i.Equals(j));        //Output True


        Direction direction1 = Direction.east;
        Direction direction2 = Direction.west;

        Console.WriteLine(direction1 == direction2);              //Output False
        Console.WriteLine(direction1.Equals(direction2));         //Output False

        Customer c2 = c1;//Reference and values equality

        Console.WriteLine(c1 == c2);           //Output True
        Console.WriteLine(c1.Equals(c2));      //Output True

        Customer c3 = new Customer();
        c3.FirstName = "Swapnadip";
        c3.LastName = "Saha";
        //Reference inequality but values equality

        Console.WriteLine(c1 == c3);          //Output False
        Console.WriteLine(c1.Equals(c3));     //Output False for base, true after
override

        //If Reference equality is true then value equality is obviously true but
value equality does not guaranty Reference equality.
```

```
        //Here == operand gives reference equality, but .Equals gives value
equality.

    }
}



    public class Customer
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }

        public override bool Equals(object obj)
        {
            if (obj == null) return false; //  the provided obj is null. If it
is, Equals returns false, meaning the current Customer instance is not equal to
null.

            if (!(obj is Customer)) return false; //if the provided obj is not of
type Customer (using the is keyword). If obj is not a Customer, the method
returns false, as objects of different types cannot be considered equal.


            return this.FirstName == ((Customer)obj).FirstName
            && this.LastName == ((Customer)obj).LastName;

            //if the FirstName and LastName properties of the current instance
(this) match those of obj, which is cast to Customer. If both the FirstName and
LastName properties match, Equals returns true, indicating that the two Customer
objects are equal. Otherwise, it returns false
        }

        public override int GetHashCode()
        {
            return this.FirstName.GetHashCode() ^ this.LastName.GetHashCode();
        }
    }

    public enum Direction
    {
        north = 1,
        east = 2,
        west = 3,
        south = 4
    }
```

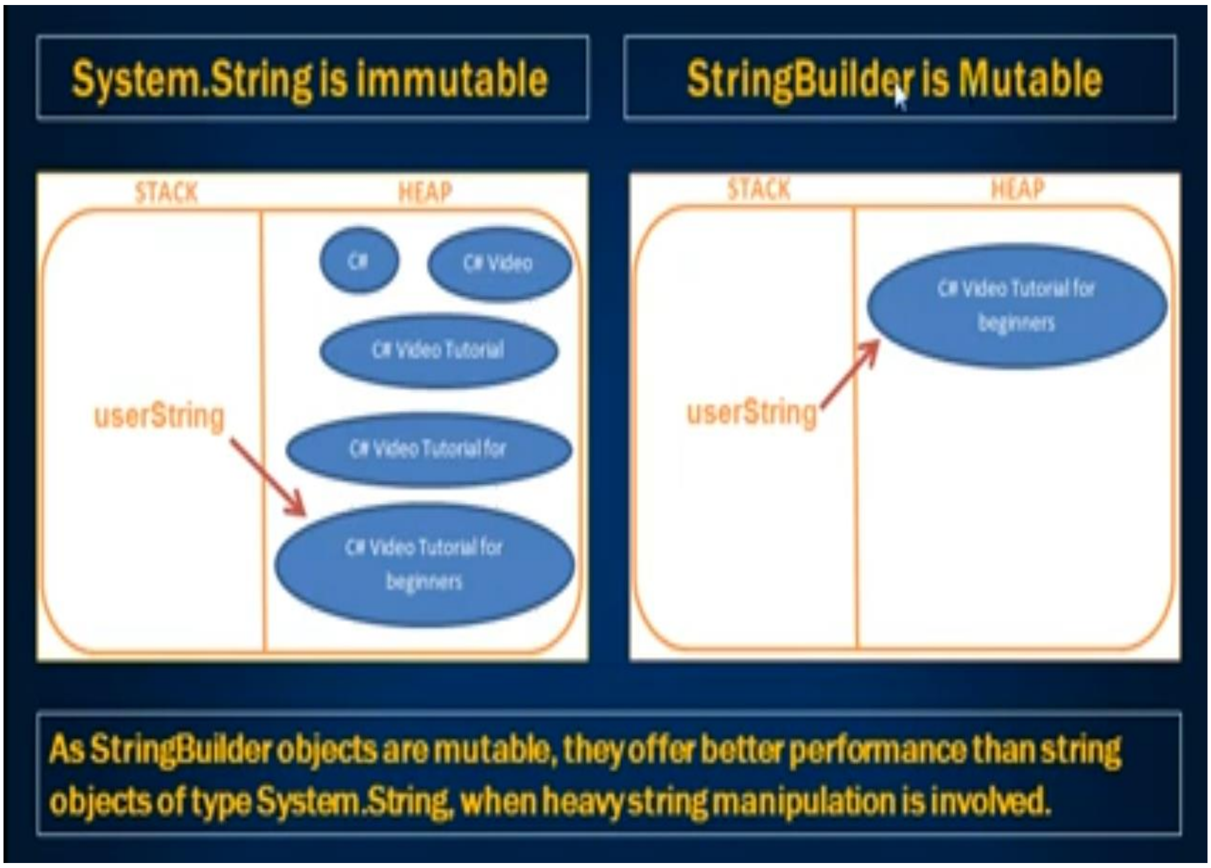
For `Equals` `override` there should be an `override` `GetHashCode()` else compiler will show a warning.

Convert.ToString() vs ToString()

- Convert.ToString() handles null, while ToString() doesn't, and throws a NULL Reference exception.
- ToString() is from System.Object, whereas Convert.ToString() is a static class from Convert class

Depending on the type of the application, architecture and what you are trying to achieve, you choose one over the other.

System.String() vs System.Text.StringBuilder



As StringBuilder objects are mutable, they offer better performance than string objects of type System.String, when heavy string manipulation is involved.

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

namespace StringvsStringBuilder
{
    class StringvsStringBuilder
    {
        public static void Main()
        {
            string userString = "C#";
            userString += " Video";
            userString += " Tutorial";
            userString += " for";
            userString += " beginners";
           userString += " -- String";

            Console.WriteLine(userString);

        }
    }

}
```

In this case for each merging a new object will be created and the reference variable will point to the latest object. The previous objects or the orphan objects (objects which do not have any reference variable) will be still in memory and will be cleared by garbage collector. So, **System.String cannot be changed once created. To change a new object needs to be created in memory.**

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

namespace StringvsStringBuilder
{
    class StringvsStringBuilder
    {
        public static void Main()
        {
            StringBuilder sb = new StringBuilder("C#");
            sb.Append(" Video");
            sb.Append(" Tutorial");
            sb.Append(" for");
            sb.Append(" beginners");
            sb.Append(" -- StringBuilder");

            Console.WriteLine(sb);

        }
    }

}
```

In this case for each merging does not create a new object and the previous string will be concatenate itself. So, **System.Text can be changed.**

Partial Class

Partial classes allow us to split a class into 2 or more files. All these parts are then combined into a single class, when the application is compiled. The partial keyword can also be used to split a struct or an interface over two or more files.

Advantages of partial classes

1. The main advantage is that, visual studio uses partial classes to separate, automatically generated system code from the developer's code. For example, when you add a webform, two .CS files are generated

    a. **WebForm1.aspx.cs** - Contains the developer code
    b. **WebForm1.aspx.designer.cs** - Contains the system generated code. For example, declarations for the controls that you drag and drop on the webform.

2. When working on large projects, spreading a class over separate files allows multiple programmers to work on it simultaneously.

Though, Microsoft claims this as an advantage, not seen anywhere, people using partial classes, just to work on them simultaneously.

Example:

The following is a normal class

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

namespace PartialClass
{
    public class Customer
    {
        private string _firstName;
        private string _lastName;

        public string FirstName
        {
            get
            {
                return _firstName;
            }
            set
            { _firstName = value;
            }
        }
        public string LastName { get => _lastName; set => _lastName = value; }
/*Property syntax-- right click refactor*/

        public string GetFullName()
        {
            return _firstName + ' ' + _lastName;
        }
    }
}
```

Now let us create two partial Classes:

Let the first one is `PartialCustomerOne.cs.` This consists only the private elements:

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

namespace PartialClass
{
    public partial class PartialCustomer
    {
        private string _firstName;
        private string _lastName;
    }
}
```

Let the second one is `PartialCustomerTwo.cs.` This consists only the public elements:

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

namespace PartialClass
```

```
{
    public partial class PartialCustomer
    {
        public string FirstName
        {
            get
            {
                return _firstName;
            }
            set
            {
                _firstName = value;
            }
        }
        public string LastName { get => _lastName; set => _lastName = value; }
/*Property syntax-- right click refactor*/

        public string GetFullName()
        {
            return _firstName + ' ' + _lastName;
        }
    }
}
```

The `partial class` **name should be same in both cases.** While compiling, these two partial class will be compiled to a single one. These files are physically may be different but logically same. In the second file there is **no declarations** for

```
        private string _firstName;
        private string _lastName;
```

But there are still accessible. The instantaneous for the partial classes and normal classes are the same.

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

namespace PartialClass
{
    public class PartialClassDemo
    {
        public static void Main()
        {
            Customer c1 = new Customer();
            c1.FirstName = "Swapnadip";
            c1.LastName = "Saha";
            Console.WriteLine("Full Name is :{0}",c1.GetFullName());


            PartialCustomer pc1 = new PartialCustomer();
            pc1.FirstName = "Saha";
            pc1.LastName = "Swapnadip";
            Console.WriteLine("Full Name is :{0}", pc1.GetFullName());
        }
    }

}
```

Creating Partial Classes

1. All the parts spread across different files, must use the `partial` keyword.
2. All the parts spread across different files, must have the **same access modifiers**.
3. If any of the parts are declared `abstract`, **then the entire type is considered** `abstract`.
4. If any of the parts are declared `sealed`**, then the entire type is considered** `sealed`.
5. If any of the parts **inherit** a class, **then the entire type inherits that class**.
6. C# does not support multiple class inheritance. **Different parts of the partial class, must not specify different base classes.**
7. Different parts of the `partial` class **can specify different base interfaces**, and the final type implements all of the interfaces listed by all of the partial declarations.

For example

----------------Interface 1----------------

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

namespace PartialClass
{
    public interface IPartialInterfaceOne
    {
        void PartialInterfaceOneMethod();
    }
}
```

----------------Interface 2----------------

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

namespace PartialClass
{
    public interface IPartialInterfaceTwo
    {
        void PartialInterfaceTwoMethod();
    }
}
```

----------------------Partial Class 1----------------------------

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

namespace PartialClass
{
    public partial class PartialCustomer : IPartialInterfaceOne
    {
        private string _firstName;
        private string _lastName;

        void IPartialInterfaceOne.PartialInterfaceOneMethod()
        {
            throw new NotImplementedException();
        }
    }
}
```

----------------------Partial Class 2-----------------------------

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

namespace PartialClass
{
    public partial class PartialCustomer : IPartialInterfaceTwo
    {
        public string FirstName
        {
            get
            {
                return _firstName;
            }
            set
            {
                _firstName = value;
            }
        }
        public string LastName { get => _lastName; set => _lastName = value; }
/*Property syntax-- right click refactor*/

        public string GetFullName()
        {
            return _firstName + ' ' + _lastName;
        }

        void IPartialInterfaceTwo.PartialInterfaceTwoMethod()
        {
            throw new NotImplementedException();
        }
    }
}
```

This is allowed.

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

namespace PartialClass
{
    public partial class PartialCustomer : IPartialInterfaceOne
    {
        private string _firstName;
        private string _lastName;

        void IPartialInterfaceTwo.PartialInterfaceTwoMethod()
        {
            throw new NotImplementedException();
        }

        void IPartialInterfaceOne.PartialInterfaceOneMethod()
        {
            throw new NotImplementedException();
        }
    }
}
```

This is also allowed. But if an interface member is implemented in one of the partial classes, then it should not be implemented in other partial classes.

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

namespace PartialClass
{
    public partial class PartialCustomer : IPartialInterfaceOne
    {
        private string _firstName;
        private string _lastName;

        void IPartialInterfaceTwo.PartialInterfaceTwoMethod()
        {
            throw new NotImplementedException();
        }

    }
}
```

This is also allowed. Implemented method is not the member of Base interface. But the member of the interfaces should be in one of the other partial classes.

8. Any members that are declared in a partial definition are available to all of the other parts of the partial class.

Partial Method

1. A **partial class or a struct** can contain **partial methods**.
2. A **partial method** is created using the **partial** keyword.
3. A **partial method** declaration **consists of two parts**.
    i) The definition (only the method signature)
    ii) The implementation.

These may be in separate parts of a partial class, or in the same part.
4. **The implementation for a partial method is optional**. If we don't provide the implementation, the compiler removes the signature and all calls to the method.
5. **Partial methods are private by default**, **and it is a compile time error to include any access modifiers, including private.**
6. **It is a compile time error, to include declaration and implementation at the same time for a partial method.**
7**. A partial method return type must be void**. Including any other return type is a compile time error. As the partial methods are private and it may not have the definition, from other class calling a partial method or returning a value can manipulate or harm the flow of the program.
8. **Signature of the partial method declaration**, must **match with the signature of the implementation**.
9. **A partial method must be declared within a partial class or partial struct**. A **non-partial class or struct cannot include partial methods**.
10. **A partial method can be implemented only once**. Trying to implement a partial method more than once, raises a compile time error

----------------------Partial Class 1 -----------------------------

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

namespace PartialMethodDemo
{
    public partial class PartialMethod
    {
        partial void SamplePartialMethod();
        public void PublicMethod()
        {
            Console.WriteLine("PublicMethod Invoked");
            SamplePartialMethod();
        }
    }
}
```

----------------------Main Class -----------------------------

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

namespace PartialMethodDemo
{
    public class PartialMethodDemo
    {
        public static void Main()
        {
            PartialMethod pm = new PartialMethod();
            pm.PublicMethod();

        }
    }
}
```

In the `partial` class `PartialMethod`, only the declaration of the method `SamplePartialMethod()` was made. In main method only

```
        pm.PublicMethod();
```

will be called. As there is no definition for `SamplePartialMethod()`, it will be ignored by the interpreter/compiler. **Also Note, declaration of a partial method can only be in one partial class.**

----------------------Partial Class 2 -----------------------------

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

namespace PartialMethodDemo
```

```
{
    public partial class PartialMethod
    {
        partial void SamplePartialMethod()
        {
            Console.WriteLine("SamplePartialMethod Invoked");
        }
    }
}
```

Now, in this case, both the methods will be called from Main(). **Note, method declaration and definition can be in a single partial class also.**

Indexers

Where are indexers used in .NET

To store or retrieve data from session state or application state variables, we use indexers.

```
using System;

namespace WebApplications
{
    public partial class Indexers : System.Web.UI.Page
    {
        protected void Page_Load(object sender, EventArgs e)
        {
            // Using the string indexer to store session data
            Session["Session1"] = "Session 1 Data";

            // Using the string indexer to store session data
            Session["Session2"] = "Session 2 Data";

            // Using the integral indexer to retrieve data
            Response.Write("Session 1 Data = " + Session[0].ToString());
            Response.Write("<br/>");

            // Using the string indexer to retrieve data
            Response.Write("Session 2 Data = " +Session["Session2"].ToString());
        }
    }
}
```

If you view the metadata of `HttpSessionState` from
`(public virtual HttpSessionState Session)`
class, you can see that there is an integral and string indexer defined. We use `this` keyword to create indexers in c#.

```
public object this[string name]
 {
     get
     {
         return _container[name];
     }
     set
     {
         _container[name] = value;
     }
 }
 public object this[int index]
 {
     get
     {
         return _container[index];
     }
     set
     {
         _container[index] = value;
     }
 }
```

Another example of indexers usage in .NET. To retrieve data from a specific column when looping thru `SqlDataReader` object, we can use either the integral indexer or string indexer.

```
string CS = ConfigurationManager.ConnectionStrings["DBCS"].ConnectionString;
using (SqlConnection con = new SqlConnection(CS))
{
    SqlCommand cmd = new SqlCommand("Select * from tblEmployee", con);
    con.Open();

    SqlDataReader rdr = cmd.ExecuteReader();

    while (rdr.Read())

    // Using integral indexer to retrieve Id column value
    Response.Write("Id= " + rdr[0].ToString() + "");

    // Using string indexer to retrieve Id column value
    Response.Write("Name= " + rdr["Name"].ToString());

    Response.Write("<br/>");
}
```

Right click on `SqlDataReader` class, to view its metadata. Notice that, there is an integral and string indexer defined.

What are indexers in c#?

From the above examples, it should be clear that, Indexers allow instances of a class to be indexed just like arrays.

```
public override object this[int i] => GetValue(i);

public override object this[string name] => GetValue(GetOrdinal(name));
```

Indexers in C#

Points to remember

1. Use `this` keyword to create an indexer
2. Just like properties indexers have get and set accessors
3. Indexers can also be overloaded

Example:

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


namespace Indexers
{
    public class Employee
    {
        public int EmployeeId { get; set; }
        public string EmployeeName { get; set; }
        public string Gender { get; set; }
    }

    public class Company
    {
        private List<Employee> listEmployees;
        public Company()
        {
            listEmployees=new List<Employee>();
            listEmployees.Add(new Employee() {
EmployeeId=1,EmployeeName="Swapno",Gender="Male" });
            listEmployees.Add(new Employee() { EmployeeId=2,EmployeeName="Dip
",Gender="Female" });
            listEmployees.Add(new Employee() { EmployeeId=3,EmployeeName= "Saha",
Gender="Male" });
            listEmployees.Add(new Employee() {
EmployeeId=4,EmployeeName="Hms",Gender= "Female" });
            listEmployees.Add(new Employee() {
EmployeeId=5,EmployeeName="Appl",Gender= "Female" });
            listEmployees.Add(new Employee() {
EmployeeId=6,EmployeeName="Support",Gender="Male" });
            listEmployees.Add(new Employee() {
EmployeeId=7,EmployeeName="Tmc",Gender="Male" });
            listEmployees.Add(new Employee() {
EmployeeId=8,EmployeeName="Kolkata",Gender="Male" });

        }
        // Use "this" keyword to create an indexer

        // This Indexer takes employeeld as parameter

        // and returns employee name


        //--------The indexer -----------------
        public string this[int employeeld]
        {
            //Just like properties indexers have get and set accessors

            get
            {
                // for the given employee id. It will return the name of the
employee
                return listEmployees.FirstOrDefault(x => x.EmployeeId ==
employeeld).EmployeeName;  //find an employee x such that, that employee's
EmployeeID should match the parameter passing into the indexer. From this object
Pick EmployeeName


                //-----------------OR-------------------------
                foreach (var employee in listEmployees)
                {
                    if (employee.EmployeeId == employeeId)
                    {
                        return employee.EmployeeName;
                    }
                }
                return null; // or throw an exception, if desired             }
            set
            {
                listEmployees.FirstOrDefault(x => x.EmployeeId ==
employeeld).EmployeeName = value;  //for an employee x such that, that employee's
EmployeeID should match the parameter passing into the indexer. Set the value
i.e. name for the EmployeeId

               // i.e. comp[2]="Swapnadip Saha"    Here the name of the employee
whose id is 2 will set to Swapnadip Saha

                //-----------------OR-------------------------


                foreach (var employee in listEmployees)
                {
                    if (employee.EmployeeId == employeeId)
                    {
                        employee.EmployeeName = value;
                        return;
                    }
                }
                // Optionally handle the case where the employeeId is not found
                throw new ArgumentException($"Employee with ID {employeeId} not
found.");              }


        }





    }
}




using Indexers;
using System;


public partial class WebApplications_38__Indexers : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {


        Company company = new Company();

        Response.Write("Name of the employees with id =2 " + company[2]);

        company[2] = "Swapnadip Saha";
        Response.Write("Now Name of the employees with id =2 " + company[2]);
    }
```

```
}
```

Overloading Indexers

```
    public string this[ string gender]
    {
        get
        {
            return listEmployees.Count(x => x.Gender == gender).ToString();
        }

        set
        {
            foreach (Employee x in listEmployees)
            {
                if (x.Gender == gender)
                {
                    x.Gender = value.ToString();
                }
            }
        }


}




using Indexers;
using System;


public partial class WebApplications_38__Indexers : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {


        Company company = new Company();

        Response.Write("Name of the employees with id =2 " + company[2]);

        company[2] = "Swapnadip Saha";
        Response.Write("Now Name of the employees with id =2 " + company[2]);
```

```
            Response.Write("Number of Male " + company["Male"]);
            company["Female"] = "Male";

             Response.Write("Number of Male " + company["Male"]);
    }

}
```

Method parameter optional

There are 4 ways that can be used to make method parameters optional

Use parameter arrays

```
namespace OptionalMethodPrama
{
    public class OptionalMethodPrama
    {
        public static void Main()
        {

//-----------------------1.Use parameter arrays------------------------------
            int[] n = new int[10] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
            AddNumbers(0, 0, n);                                        // Output 55
            AddNumbers(11, 12, 13, 14, 15, 16, 17, 18, 19, 20);         // Output 155
            AddNumbers(55, 101, null);                                  // Output 156
            AddNumbers(55, 155);                                        // Output 210
        }

//-----------------------1.Use parameter arrays------------------------------


        public static void AddNumbers(int FirstNumber,int SecondNumber,params
int[] numbers)
        {
            int? sum = FirstNumber + SecondNumber;
            if (sum != null)
            {
                foreach (int i in numbers)
                {
                    sum = sum + i;
                }
                Console.WriteLine("\nSum is {0}", sum);
            }
        }

    }
}
```

Method overloading

```
 //-----------------------2.Method Overloading---------------------------------

            int[] n = new int[10] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
            AddNumbers(0, 0, n);                                        // Output 55
            //AddNumbers(11, 12, 13, 14, 15, 16, 17, 18, 19, 20);     // Output
155
            AddNumbers(55, 101, null);                              // Output 156
            AddNumbers(55, 155);                                    // Output 210


//-----------------------2.     Method Overloading-----------------------------

 public static void AddNumbers(int FirstNumber, int SecondNumber, int[] numbers)
 {
     int? sum = FirstNumber+ SecondNumber;
     if (sum != null)
     {
         if (numbers != null)
         {
             foreach (int i in numbers)
             {
                 sum = sum + i;
             }

         }
         Console.WriteLine("\nSum is {0}", sum);
     }
 }

 public static void AddNumbers(int FirstNumber, int SecondNumber)
 {
     int? sum = FirstNumber + SecondNumber;
     if (sum != null)
     {
         Console.WriteLine("\nSum is {0}", sum);
     }
 }
```

Specify parameter defaults

```
//-----------------------3. Specifying parameter defaults------------------------

            int[] n = new int[10] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
            AddNumbers(0, 0, n);                                    // Output 55
            //AddNumbers(11, 12, 13, 14, 15, 16, 17, 18, 19, 20);   // Output 155
            AddNumbers(55, 101, null);                              // Output 156
            AddNumbers(55, 155);                                   // Output 210
```

Like params optional parameter must appear after all the required parameter

Named parameters

In the following method, parameters "b" & "c" are optional.

```
public static void Test(int a, int b = 10, int c = 20)
{
    Console.WriteLine("a =" + a);
    Console.WriteLine("b =" + b);
    Console.WriteLine("c =" + c);
}
```

When we invoke this method as shown below, "1" is passed as the argument for parameter "a" and "2" is passed as the argument for parameter "b" by default.

```
Test(1, 2);

// Output

       a =1
       b =2
       c =20
```

If the intention is to pass "2" as the argument for parameter "c". We can make use of named parameters, as shown below. Notice that, the name of the parameter for which value "2" is being passed.

```
Test(1, c: 2);

// Output

       a =1
       b =10
       c =2
```

Optional Attribute

Use Optional Attribute that is present in **System.Runtime.InteropServices namespace**

```
using System.Runtime.InteropServices;

namespace OptionalMethodPrama
{
    public class OptionalMethodPrama
    {
        public static void Main()
        {
            //-----------------------4.Optional Keyword--------------------------
-----------

            int[] n = new int[10] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
            AddNumbers(0, 0, n);                                    // Output 55
            //AddNumbers(11, 12, 13, 14, 15, 16, 17, 18, 19, 20);     // Output
155
            AddNumbers(55, 101, null);                              // Output 156
            AddNumbers(55, 155);                                    // Output 210
            AddNumbers(55, 155, new int[] { 15, 15, 15 });          // Output 255
        }
}

        //-----------------------4.    Optional Keyword--------------------------
----

        public static void AddNumbers(int FirstNumber, int
SecondNumber,[Optional] int[] FourthNumbers)
        {
            int? sum = FirstNumber + SecondNumber;
            if (sum != null)
            {
                if (FourthNumbers != null)
                {
                    foreach (int i in FourthNumbers)
                    {
```

```
                    sum = sum + i;
                }

            }
            Console.WriteLine("\nSum is {0}", sum);
        }
    }
}
```

In all the above examples

```
AddNumbers(55, 155);
```

are the examples of optional parameters.

Code Snippets

Code snippets are ready-made snippets of code you can quickly insert into your code.

1. Keyboard shortcut: CTRL K+X
2. Right click and select "Insert Snippet...", from the context menu
3. Click on Edit-Intellisence-Insert Snippet
4. Use code snippets short cut. For example, to use "for loop" code snippet, type "for" and press TAB key twice

Once a code snippet is inserted, the editable fields are highlighted in yellow, and the first editable field is selected automatically. Upon changing the first editable field, press TAB to move to the next editable field. To come to the previous editable field use SHIFT+ TAB. Press ENTER or ESC keys to cancel field editing and return the Code Editor to normal.

Code Snippet Types:

Expansion: These snippets allows the code snippet to be inserted at the cursor.

Surrounds With: These snippets allow the code snippet to be placed around a selected piece of code.

Refactoring: These snippets are used during code refactoring.

Surround-with Code Snippets

Surround-with snippets surrounds the selected code, with the code snippets code.

1. Select the code to surround, and use keyboard shortcut CTRL K+S
2. Select the code to surround, right click and select "Surround with.." option from the context menu
3. Select the code to surround, then click on Edit menu, select "IntelliSense" and then select the "Surround With" command.

Code snippets can be used with any type of applications that you create with visual studio. For example, you can use them with

1. Console applications
2. ASP.NET web applications
3. ASP.NET MVC applications etc...

Code snippets are available for the following languages.

1. 1.C#
2. Visual Basic
3. XML
4. HTML
5. Jscript
6. SQL

Code Snippet Manager

Code Snippet Manager can be used to Add or remove code snippets. You can also find the following information about a code snippet.

1. Description
2. Shortcut
3. Snippet Type
4. Author

To access code snippet manager, click on "Tools" and then select "Code Snippet Manager". Code snippets are xml files and have snippet extension.

List Collection Classes

Dictionary

1. A dictionary is a collection of (key, value) pairs.
2. Dictionary class is present in System.Collections.Generic namespace.
3. When creating a dictionary, we need to specify the type for key and value.
4. Dictionary provides fast lookups for values using keys.
5. Keys in the dictionary must be unique.

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

namespace Dictionary
{
    public class Dictionary
    {
        public class Customer
        {
            public int ID { get; set; }
            public string Name { get; set; }
            public int Salary { get; set; }

        }
        public static void Main()
        {
            Customer cust001 = new Customer() { ID = 1001, Name = "Swapnadip", 
Salary = 500 };
            Customer cust002 = new Customer() { ID = 1002, Name = "Saha", Salary 
= 550 };
```

```
            Customer cust003 = new Customer() { ID = 1003, Name = "Pidanpaws",
Salary = 450 };

            Dictionary<int, Customer> custDict = new Dictionary<int, Customer>();
            custDict.Add(cust001.ID, cust001);
            custDict.Add(cust002.ID, cust002);
            custDict.Add(cust003.ID, cust003);

             // To check whether the key is already exists in the dictionary as
      the keys are unique in Dictionary

            if (!custDict.ContainsKey(cust003.ID))
            {
                custDict.Add(cust003.ID, cust003);
            }


            if (custDict.ContainsKey(1004))
            {
                Customer cust = custDict[1004];   // Will give a runtime error as
there is no ID exists. To sole the issue, add a check (if condition)
            }
            // To  Retrieve   Method 1
            Console.WriteLine("To  Retrieve   Method 1");
            Customer cust004 = custDict[cust001.ID];
            Console.WriteLine("Name of ID {0} is {1}", cust001.ID, cust004.Name);

            // To  Retrieve   Method 2
            Console.WriteLine("To  Retrieve   Method 2");
            Customer cust005 = custDict[1003];
            Console.WriteLine("Name of ID 1003 is {0}", cust005.Name);

            // To  Retrieve   Method 3  foreach - KeyValuePair
            Console.WriteLine("To  Retrieve   Method 3 foreach - KeyValuePair");
            foreach (KeyValuePair<int, Customer> Dict in custDict)
//foreach (var Dict in custDict) can be used   For better readability use
KeyValuePair
            {
                Console.WriteLine("Key is {0}", Dict.Key);
                Customer customer = Dict.Value;
                Console.WriteLine("ID={0},Name={1} and salary={2}", customer.ID,
customer.Name, customer.Salary);
            }

            // To  Retrieve   Method 4  foreach - Key
            Console.WriteLine("To  Retrieve   Method 4 foreach - Key");
            foreach (int Dict in custDict.Keys)
            {
                Console.WriteLine("Key is {0}", Dict);
            }

            // To  Retrieve   Method 5  foreach - value
            Console.WriteLine("To  Retrieve   Method 5 foreach - value");
            foreach (Customer Dict in custDict.Values)
            {
                Console.WriteLine("Values are ID={0},Name={1} and salary={2}",
Dict.ID, Dict.Name, Dict.Salary);
            }
        }
    }

}
```

TryGetValue()

When we are not sure that a dictionary is contains a specific key. It returns a value associated with a specific key. Other wise we may get a runtime exception.

```
    //----------------TRYGETVALUE()--------------------
            Customer cc = new Customer();
            if (custDict.TryGetValue(100001, out cc))
            {
                Console.WriteLine("Values are ID={0},Name={1} and salary={2}", 
cc.ID + cc.Name + cc.Salary);
            }
            else
            {
                Console.WriteLine("Key not Found");
            }
```

Count()

To found the total number of elements of a Dictionary. There are two methods to get the count

1. The Count property

```
 Console.WriteLine("Number of elements {0}", custDict.Count);
```

2. The Count() function

```
Console.WriteLine("Number of elements {0}", custDict.Count())
```

Here the `Count()` function is coming from `Enumerable` Class and it is a from `System.Linq` namespace i.e. Linq extension method on dictionary objects.

The `Count()` function has an overloaded option which takes a predicate (LINQ) and outputs the count accordingly.

```
Console.WriteLine("Number of elements {0}", custDict.Count(KeyValuePair=>
KeyValuePair.Value.Salary>500));  //Provides the number of elements whose Salary
is greater than 500

 Console.WriteLine("Number of elements {0}", custDict.Count(KeyValuePair=>
KeyValuePair.Value.Name.StartsWith("S")));  //Provides the number of elements
whose Name Starts with S
```

Remove()

To remove an item from a dictionary.

```
custDict.Remove(cust001.ID);
```

If the key is not existing in the dictionary, then there will be no compilation or runtime errors.

Clear()

To clear the dictionary, i.e. to remove all the key and values at once.

```
custDict.Clear();
```

Using LINQ extension methods with Dictionary

The count(predicate) example

Different ways to convert an array into a dictionary

```
            Customer[] arrCustomer= new Customer[3];
            arrCustomer[0] = cust001;
            arrCustomer[1] = cust002;
            arrCustomer[2] = cust003;

            Dictionary<int, Customer> custDict01 = arrCustomer.ToDictionary(cust
=> cust.ID, cust => cust);
            foreach (KeyValuePair<int, Customer> Dict in custDict01)
            {
                Console.WriteLine("Key is {0}", Dict.Key);
                Customer customer = Dict.Value;
                Console.WriteLine("ID={0},Name={1} and salary={2}", customer.ID,
customer.Name, customer.Salary);
```

Different ways to convert a list into a dictionary

```
            List<Customer> lstCustomer = new List<Customer>() { };
            lstCustomer.Add(cust002);
            lstCustomer.Add(cust001);
            lstCustomer.Add(cust003);

            Dictionary<int, Customer> custDict02 = lstCustomer.ToDictionary(cust
=> cust.ID, cust => cust);
            foreach (KeyValuePair<int, Customer> Dict in custDict02)
            {
                Console.WriteLine("Key is {0}", Dict.Key);
                Customer customer = Dict.Value;
                Console.WriteLine("ID={0},Name={1} and salary={2}", customer.ID,
customer.Name, customer.Salary);

            }
```