C#

### Reading and writing to a console

**1.** Reading from the console  
**2.** Writing to the console  
**3.** Two ways to write to console  
     **a)** Concatenation  
     **b)** Place holder syntax – Most preferred

**Example:**  
using System;  
class Program  
{  
    static void Main()  
    {  
        // Prompt the user for his name  
        Console.WriteLine("Please enter your name");  
        // Read the name from console  
        string UserName = Console.ReadLine();  
        // Concatenate name with hello word and print  
        Console.WriteLine("Hello " + UserName);  
  
        //Placeholder syntax to print name with hello word   
        //Console.WriteLine("Hello {0}", UserName);  
    }  
}

### Built-in types & variables

### 2.1 Built- In types in C#:

**Defines proper use of identifier or an expression.**

**1.** Boolean type – Only true or false   
**2.** Integral Types - sbyte, byte, short, ushort, int, uint, long, ulong, char  
**3.** Floating Types – float and double  
**4.** Decimal Types   
**5.** String Type

6. Numeric – 1000, 3.14158

**Example:**

* Strings – “Hello”, “World”
* Characters – ‘a’, ‘t’
* Booleans – true/false
* Integer: int, long
* Float : holds 127 digits after decimal point
* Double: 1023 digits after decimal.
* Byte – stores small numbers from -128 to 127
* Short – stores -23768 to 32767
* Char – single quotes. - alphabetic character stored as numeric value ASCII(ascii can be explained as each key on keyboard is stored internally with a numeric)

### 2.2 Variables:

* A name for a memory location.
* Always has a type.
* Data stored in variable is value
* Variable must start with a letter.

### 2.3 Escape Sequences in C#:

**Escape Sequences in C#**  
<http://msdn.microsoft.com/en-us/library/h21280bw.aspx>

**Example:**

String Name = “One\n Two\nThree”; \n prints in a new line.

### 2.4 Verbatim in C#:

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.

### Common Operators

Assignment Operator **=**  
Arithmetic Operators like **+,-,\*,/,%**  
Comparison Operators like **==, !=,>, >=, <, <=**  
Conditional Operators like **&&, ||**  
Ternary Operator **?:**  
Null Coalescing Operator **??**

**Program without ternary Operator**  
using System;  
class Program  
{  
    static void Main()  
    {  
        int Number = 10;  
  
        bool IsNumber10;  
  
        if (Number == 10)  
        {  
            IsNumber10 = true;  
        }  
        else  
        {  
            IsNumber10 = false;  
        }  
  
        Console.WriteLine("i == 10 is {0}", IsNumber10);  
    }  
}  
  
**Same program with ternary Operator**  
using System;  
class Program  
{  
    static void Main()  
    {  
        int Number = 10;  
  
        bool IsNumber10 = Number == 10 ? true : false;  
          
        Console.WriteLine("i == 10 is {0}", IsNumber10);  
    }  
}

**Example2:**

Console.WriteLine(num1 + num2);

Console.WriteLine(num1 - num2);

Console.WriteLine(num1 \* num2);

Console.WriteLine(num1 / num2); //division

Console.WriteLine(num1 % num2); // reminder

Console.WriteLine(4 / 2);

Console.WriteLine(num3 / num4);

Console.WriteLine(num3 % num4);

Console.ReadKey();

**Arithmetic Operator order of precedence**:

1. Multiplication/ division and Exponentiation has the order of precedence when compared to Addition or substraction:

Eg: 2+3\*4 = 14 ,

if you want the addition to be performed first and the expression has to be in paranthesis. Otherwise the compiler will follow its rules which is multiply first.

### 4. Nullable Types

### 4.1 Nullable types in C #:

**Value Types:** Int, float, double, structs, enums etc

**Reference types** : interference, class, delegates , etc

**In C# types  are divided into 2 broad categories.**  
**Value Types**  - int, float, double, structs, enums etc  
**Reference Types** – Interface, Class, delegates, arrays etc

Value type by default is not Null.

Reference Types are by default Null.

**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**  
  
**Program without using NULL coalescing operator**  
 using System;  
 class Program  
 {  
    static void Main()  
    {  
        int AvailableTickets;  
        int? TicketsOnSale = null;  
  
        if (TicketsOnSale == null)  
        {  
            AvailableTickets = 0;  
        }  
        else  
        {  
            AvailableTickets = (int)TicketsOnSale; // Coverting nullable type to nonNUllable.

// AvailableTickets = TicketsOnSale.Value; // Can also write as a value if not using the type cast operator.  
        }  
  
        Console.WriteLine("Available Tickets={0}", AvailableTickets);    }}

### 4.2 Null Coalescing Operator ?? #:

**The above program is re-written using NULL coalescing operator**  
using System;  
class Program  
{  
    static void Main()  
    {  
        int AvailableTickets;  
        int? TicketsOnSale = null;  
  
        //Using null coalesce operator ??   
        AvailableTickets = TicketsOnSale ?? 0; // If the value is 0, print zero else whatever is the value in the tickets on Sale.  
  
        Console.WriteLine("Available Tickets={0}", AvailableTickets);  
    }  
}

### 5. Datatype conversions

* Implicit conversions
* Explicit conversions
* Difference betwee Parse() and tryParse()

### 5.1 Implicit 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 loose any data and no exception will be thrown, hence an implicit conversion can be done.   
  
Where as when converting a **float** to an **int**, we loose 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#.

**Implicit Conversions: small Data type to large data type**

1. when a conversion happens without we explicitly do something.

2. Implicit conversion is done by the conversion automatically when there is no loss of information if the conversion is done.

3. And when there is no possibity of throwing exceptions during th conversion.

**Eg**: Converting an Int to float which is a very large number will not loose any data and no exception will be thrown, hence an implicit conversion can be done.

using System;

class DataImplicit

{

public static void Main()

{

int i = 100;

float f = i;

Console.WriteLine(f);

Console.ReadKey();

}

} O/P = 100;

### 5.2 Explicit conversion:

**Explicit Conversion: large data type to small data type**

On the other hand if you want to convert from float to int, this will not be implicitly converted by the complier. This needs to be explicitly converted.

There are 2 ways of explicit conversions

1. The cast method () -

Eg: using System;

class ExplicitConvrt

{

public static void Main()

{

float f = 123.45f;

int i = (int)f; // casting explicitly method

Console.WriteLine(i);

Console.ReadKey();

}

}

O/P: 123 ( we lost the fractional part)

2. dotnet explicit conversion convert.to

using System;

class ExplicitConvrt

{

public static void Main()

{

float f = 123.45f;

int i = Convert.ToInt32(f);

Console.WriteLine(i);

Console.ReadKey();

}

}

o/p: 123

Difference between type cast and convert :

Type cast operator – if I have a very long float, we dont get exception where as when we use convert class, we get the exception.

### 5.3 Type Cast Operator:

**Type Cast operator**:

using System;

class ExplicitConvrt

{

public static void Main()

{

float f = 14324343353454523.45f;

int i = (int)f;

Console.WriteLine(i);

Console.ReadKey();

}

}

O/p: -2147483648

### 5.4 Convert:

using System;

class ExplicitConvrt

{

public static void Main()

{

float f = 14324343353454523.45f;

int i = Convert.ToInt32(f);

Console.WriteLine(i);

Console.ReadKey();

}

}

o/p:An unhandled exception of type 'System.OverflowException' occurred in mscorlib.dll

Additional information: Value was either too large or too small for an Int32.

### 5.5 Parse and TryParse:

**Difference between Parse and TryParse**  
**1.** If the number is in a string format you have 2 options - Parse() and TryParse()   
**2.** Parse() method throws an exception if it cannot parse the value, whereas TryParse() returns a bool indicating whether it succeeded or failed.  
**3.** Use Parse() if you are sure the value will be valid, otherwise use TryParse()

**Parse()** : There is another way to convert the datatype.

1. Following converts a string to a integer.

class someclass

{

public static void Main()

{

String strnum = "122";

int i = int.Parse(strnum);

Console.WriteLine(i);

Console.ReadKey();

}

}

**tryParse()** : If you need to convert or extract an integer from a string use this.. Try parse returns a boolean as a result.

Eg: using System;

class someclass

{

public static void Main()

{

String strnum = "122TG";

int Res;

bool IsConversionSuccess = int.TryParse(strnum, out Res);

if (IsConversionSuccess)

{

Console.WriteLine(Res);

} else

{

Console.WriteLine("Enter a valid number");

Console.ReadKey();

}

}

}

Use Parse() if you know the value is correct else use tryparse().

namespace WindowsFormsApplication1

{

public partial class Form1 : Form

{

public Form1()

{

InitializeComponent();

}

private void button1\_Click(object sender, EventArgs e)

{

String input = tx.Text;

int res;

bool isTheNumint = int.TryParse(input, out res);

if (isTheNumint)

{

MessageBox.Show("Result is" + res);

}

else

{

MessageBox.Show("This field only accepts a number.");

}

}

}

}

### 6. Datatype conversions

As we defined data members and a constructor, the next logical thing to do is create a method that displays the data in a class object. We use ToString method to display data members.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace ToString

{

class userinfo

{

private String first, middle, last;

public userinfo (String fname, String fmiddle, String flast)

{

first = fname;

middle = fmiddle;

last = flast;

}

public override String ToString()

{

return first + " " + middle + " " + last;

}

}

class Program

{

static void Main(string[] args)

{

userinfo usehere = new userinfo("test", "test", "test");

Console.WriteLine(usehere.ToString());

Console.ReadKey();

}

}

}

### 7. Arrays in C#

:

Arrays is a collection of similar data types.

**Adv**: Arrays are strongly typed.

To store a group of similar data types. Such as collection of even numbers.

To declare a memory to store multiple values, we use the keyword use.

Arrays use indexing.

Eg:

class someclass

{

public static void Main()

{

int [] b = new int[4] ;

b[0] = 80;

b[1] = 89;

b[2] = 34;

b[3] = 08;

Console.WriteLine(b[3]);

Console.ReadKey();

}

}

**An array is a collection of similar data types.**  
  
using System;  
class Program  
{  
    public static void Main()  
    {  
        // Initialize and assign values in different lines  
        int[] EvenNumbers = new int[3];  
        EvenNumbers[0] = 0;  
        EvenNumbers[1] = 2;  
        EvenNumbers[2] = 4;  
  
        // Initialize and assign values in the same line  
        int[] OddNumbers = { 1, 3, 5};  
  
        Console.WriteLine("Printing EVEN Numbers");  
  
        // Retrieve and print even numbers from the array  
        for (int i = 0; i < EvenNumbers.Length; i++)  
        {  
            Console.WriteLine(EvenNumbers[i]);  
        }  
  
        Console.WriteLine("Printing ODD Numbers");  
  
        // Retrieve and print odd numbers from the array  
        for (int i = 0; i < OddNumbers.Length; i++)  
        {  
            Console.WriteLine(OddNumbers[i]);  
        }  
    }  
}

**Disadvantages**:

1.Once the size of array is initialised, it cannot grow overtime. We have other collection classes like list, hash table which grow in size.

2. Have t o rely on integral indices to store or retreive items unlike collections and lists.

**Advantages:** Arrays are strongly typed.  
  
**Disadvantages:** Arrays cannot grow in size once initialized. Have to rely on integral indices to store or retrieve items from the array.

### 8. For and ForEachLoop

**While loop example:**

class someclass

{

public static void Main()

{

int[] numbers = new int[3];

numbers[0] = 101;

numbers[1] = 102;

numbers[2] = 13;

int counter = 0; // initialise

while (counter < numbers.Length) // condition check

{

Console.WriteLine(numbers[counter]);

counter++; // increment variable

}

Console.ReadKey();

}

}

**If we have to do the same for Forloop:**

class someclass

{

public static void Main()

{

int[] numbers = new int[3];

numbers[0] = 101;

numbers[1] = 102;

numbers[2] = 13;

for( int j = 0; j<numbers.Length; j++)

{

Console.WriteLine(numbers[j]);

}

Console.ReadKey();

}

}

**If we have to do the same for For eachloop**:

If we have a collection of something, we can use each loop, we dont have to know how many elements are there in that collection.

Loop through the collection as long as there are elements inside that

class someclass

{

public static void Main()

{

int[] numbers = new int[3];

numbers[0] = 101;

numbers[1] = 102;

numbers[2] = 13;

foreach (int k in numbers)

{

Console.WriteLine(k);

}

}

}

### 9. Methods

There can be 2 types of methods:

A static method, an instance method.

A static method is called by its classname.methodname.

An instance method is called by creating an instance of the class.

Methods can be void or have a return type.

Methods can have no parameters or has parameters.

using System;

class methods

{

public static void callhardcode20()

{

int num = 0;

while(num <= 20)

{

Console.WriteLine("Evens are :" + num);

num = num + 2;

}

Console.ReadKey();

}

public void evenNumbers(int target)

{

int num = 0;

while (num <= target)

{

Console.WriteLine("Even Numers are: " + num);

num = num + 2;

}

Console.ReadKey();

}

public int add (int a, int b)

{

return a + b;

}

}

using System;

class thirtyevens

{

public static void Main()

{

methods.callhardcode20();

// methods different = new methods();

//different.evenNumbers(50);

methods addition = new methods();

int total = addition.add(10, 2);

Console.WriteLine("Total is " + total );

Console.ReadKey();

}

}

**Question**: how to execute mulitple methods of a class into another class's main method?

**Method Parameters**:There are 4 types of method parameters

1. Passing a parameter by a value - Value parameters

2. Passing a parameter by reference. - Reference parameters

3. Out Parameters

4. Parameter Arrays

### 9.1 Value Vs Reference:

**Value vs reference:**

**value: operations of one variable will not effect the operations on other variable.**

**ref: Operations of one variable will effect the operations on other variable**

I have 2 variables i = 0, and a method j =101. When a value is passed as a parameter to the method, the value of i is pointing to zero in the memory and the value of j is pointing to 101 in the memory.

where as if I use a ref parameter, both the variables point to the same value which is in the method in the memory.

**Eg of a value Parameter:**

using System;

class Diffparameters

{

public static void Main()

{

int i = 0;

Diffparameters inst = new Diffparameters();

inst.simplemethod(i);

Console.WriteLine(i);

Console.ReadKey();

}

public void simplemethod(int j)

{

j = 101;

}

}

**o/p = 0**

### 9.2 Ref Parameter:

**Eg of a Ref Parameter:**

using System;

class Diffparameters

{

public static void Main()

{

int i = 0;

Diffparameters inst = new Diffparameters();

inst.simplemethod(ref i);

Console.WriteLine(i);

Console.ReadKey();

}

public void simplemethod(ref int j)

{

j = 101;

}

}

**O/P : 101**

### 9.3 Output Parameter:

**Output Parameters**: A single method returning more than 1 value from the function.

A normal method that can return only 1 output

using System;

class outParameters

{

public static void Main()

{

outParameters inst1 = new outParameters();

int res = inst1.outParamsMethod(10, 20);

Console.WriteLine(res);

Console.ReadKey();

}

public int outParamsMethod(int i, int j)

{

return i + j;

}

}

### 9.4 Param Method:

A param method that returns more than 1 output.

using System;

class outParameters

{

public static void Main()

{

outParameters inst1 = new outParameters();

int sumRes;

int productRes;

inst1.outParamsMethod(10, 20, out sumRes, out productRes );

Console.WriteLine("Sum is {0} && Multiplication is {1}", sumRes, productRes);

Console.ReadKey();

}

public void outParamsMethod(int i, int j, out int sum, out int product)

{

sum = i + j;

product = i \* j;

}

}

### 9.5 Param Arrays:

**Parameter Arrays**: decorted with params keyword to create parameter arrays. makes the argument optional.

Params paramenter has to be the last arg in a multiple args.

Cannot use multiple params.

o/p: List of Elements are 0 - in this case, including the KW param made the arg optional.

using System;

class someclass

{

public static void Main()

{

int[] numbers = new int[3];

numbers[0] = 101;

numbers[1] = 102;

numbers[2] = 13;

paramethod();

}

public static void paramethod(params int[] num)

{

Console.WriteLine("List of Elements are {0}", num.Length);

foreach (int k in num)

{

Console.WriteLine(k);

}

Console.ReadKey();

}

}

o/p: List of Elements are 3

101

102

13

using System;

class someclass

{

public static void Main()

{

int[] numbers = new int[3];

numbers[0] = 101;

numbers[1] = 102;

numbers[2] = 13;

paramethod(numbers);

}

public static void paramethod(params int[] num)

{

Console.WriteLine("List of Elements are {0}", num.Length);

foreach (int k in num)

{

Console.WriteLine(k);

}

Console.ReadKey();

}

}

o/p: List of Elements are 4

1

2

3

4

using System;

class someclass

{

public static void Main()

{

int[] numbers = new int[3];

numbers[0] = 101;

numbers[1] = 102;

numbers[2] = 13;

paramethod(1,2,3,4);

}

public static void paramethod(params int[] num)

{

Console.WriteLine("List of Elements are {0}", num.Length);

foreach (int k in num)

{

Console.WriteLine(k);

}

Console.ReadKey();

}

}

o/p: List of Elements are 3

2

3

4

using System;

class someclass

{

public static void Main()

{

int[] numbers = new int[3];

numbers[0] = 101;

numbers[1] = 102;

numbers[2] = 13;

paramethod(1,2,3,4);

}

public static void paramethod(int x, params int[] num)

{

Console.WriteLine("List of Elements are {0}", num.Length);

foreach (int k in num)

{

Console.WriteLine(k);

}

Console.ReadKey();

}

}

### 10. Namespaces

Project A

TeamA

TeamB

I want all the teamA files to be in projectA.teamA namespace and teamB in projectA.TeamB namespace.

we can resolve the class names by using a fqdn or a alliases.

Eg: using System;

using System.Collections.Generic;

using PATA = ProjectA.TeamA; // alliases

class Program

{

static void Main(string[] args)

{

PATA.ClassA.Print();

ProjectA.TeamB.ClassA.Print();

}

}

namespace ProjectA

{

namespace TeamA

{

class ClassA

{

public static void Print()

{

Console.WriteLine("Print team A");

}

}

}

}

namespace ProjectA

{

namespace TeamB

{

class ClassA

{

public static void Print()

{

Console.WriteLine("Print team B");

}

}

}

}

Namespaces dont correspond to file directory or assembly names.

We can create project for team A and Team B as class libraries

Example: Namespace helps to organise

Job portal - all classes of jobportal.jobseekers namespace

job providers - in jobportal.jobproviders namespace.

ProjectA.TeamA

ProjectA.TeamB

my main method has only main method.

Created 2 projects of type class library for ProjectA.TeamA and ProjectA.TeamB and build the solution. Copied all the code. They are in thier respecive assemblies. the DLL has the class A and the print method.

There is a tool IL DASM - open that dll file from the , you can see what's present in what in a structure.

Now I want to use both the assemblies in my main project. to do that right click on project - click add reference - I can add a reference of the both projects. so I can start using them with the using directive.

**FInally My project looks like this** :

using System;

using ProjectA.TeamA;

using PATB = ProjectA.TeamB;

class Program

{

static void Main(string[] args)

{

ProjectA.TeamA.ClassA.Print();

PATB.ClassA.Print();

}

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace ProjectA.TeamA

{

public class ClassA

{

public static void Print()

{

Console.WriteLine("Print team A");

Console.ReadKey();

}

}

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace ProjectA.TeamB

{

public class ClassA

{

public static void Print()

{

Console.WriteLine("Print team B");

Console.ReadKey();

}

}

}

A namespace can contain another namespace, a class, an interface, a struct, a enum, a delegate.

### 12.Classes

Main class is the entry point where the code executes. In a project, there can be multiple classes but can have only 1 main method.

Static classes - client code cannot modify any data.

Non static classes - client code can create instances of a class.

### 12.1 Introduction to Classes

**What is a class?**

Example, I want to store a number, I can use integer data type. If I want to store customer information such as name, info, db etc, I want to use classes.

A class can not only store data but also perform certain actions like store customer info into database and print customer name etc.

**A class consists of state( Data) and behavior.**

**Data is represented by its fields**

**Behavior by its methods.**

using System;

class Customer

{

// State of a class: this class has some state now representing these 2 fields.

String \_Fname;

String \_Lname;

// To initiaise fields, a class can have a contructor. A constructor will have the same name as the classes.

//\*\*\*\*\* A constructor does not return a value like a method. A method can have a return type but not a constructor.

// The way to identify a constructor is that it has the same name as a class and it takes parameters and IT HAS NO RETURN TYPE

public Customer (String firstname, String lastname)

{

// \_Fname = firstname;

// \_Lname = lastname;

// you can also write as this.

// This actually refers to an instance of a class.

this.\_Fname = firstname;

this.\_Lname = lastname;

// This constructor will get called automatically when we call an instance of this class

}

// Now I want to print the customer infomation, for that I want to have a method

public void PrintFullname()

{

Console.WriteLine("The Full name = {0}", this.\_Fname + " " + this.\_Lname);

}

// A class can also have a destructors:

// Destructors can have the same name of your class, but they dont take parameters

// They dont have a return type

// They have a ~ before it

// We use destructors to clean up any resourced your class was holding for life time.

// These destcurtors we dont have to call it, it is austomatically called by the garbage collector.

~Customer()

{

//clean up code goes here.

}

}

class Program

{

static void Main(string[] args)

{

Customer C1 = new Customer("Priya " , "WIll make C# in the name of JESUS :). Thank you JESUS "); // C1 is called as an instance or object

C1.PrintFullname();

Console.ReadKey();

}

}

Is it mandatory that a constructor needs to be created everytime you make a class? NO

If we dont have a constructor, in the class, we can use the default parameter less constructor which is like calling an instance of the class

customer C1 = new customer()

When I run this, the string fields were not initialised, so some one has to go an input the details.

Let's say we want to provide the capability where somebody is able to create a custom objects, passing in any parameters to the constructor? You can also write a consructor that does not take any parameters.

**Overloading constructor**:

using System;

class customer

{

String \_firstname;

String \_lastname;

public customer() : this("default first" , "default last") // this here is sending the default values to firstname, lastname, these are then initialised // This is C#’s default constructor which is calling the customised constructor below.

{

}

public customer(String firstname, String lastname)

{

this.\_firstname = firstname;

this.\_lastname = lastname;

}

public void printinfo()

{

Console.WriteLine("Firstname = {0}", this.\_firstname + " " + this.\_lastname);

}

}

class Program

{

static void Main(string[] args)

{

customer c1 = new customer("test", "test"); // this is going to the constructor with default ex one

c1.printinfo();

customer C2 = new customer(); // this is using the constructor which doesnt take any parameters and other takes the parameters. this is called overloading constructor which can be overloaded by number or data types..

}

}

### 13.Static and Instance Members

Static members are invoked by the name of the class

non static or Instance members are invoked by the instance of the class.

Static will have only 1 copy of memory

Instace will have copy in memory for every instance.

Access modifiers are not allowed on static constructors.

On the other hand, if an access modifier is not public for a non static constructor, by default the constructor becomes private which means it can only be used in this class but not anywhere else.

Static constuctors initialise oe copy of static field in memory.

Static constructrs are called only once and are called before instance constructor and even before you refer to any static field of that class.

Instance constructors are automatically called every time a new object is created.

using System;

class circle

{

// static float \_pi = 3.14f; // can use this static field

static float \_pi;

int \_radius;

static circle()// or a static constructor . Static constructor does not take parameters.

{

\_pi = 3.14f;

}

public circle(int radius)

{

\_radius = radius;

}

public float calcArea()

{

return circle.\_pi \* this.\_radius \* this.\_radius;

}

}

class Program

{

static void Main(string[] args)

{

circle c1 = new circle(6);

float area = c1.calcArea();

Console.WriteLine("something = {0}", area);

circle C2 = new circle(5);

float area2 = C2.calcArea();

Console.WriteLine("somethinf = {0}", area2);

Console.ReadKey();

}

}

### 14. Inheritence

Inheritence is used to re-use the code. Inheritance is the main pillar of the OOPS

There are 4 pillars of inheritance in OOPS:

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

Inheritence Eg:

using System;

class Employee

{

public String \_Firstname;

public String \_Lastname;

public String \_Email;

public void fullnameprint()

{

Console.WriteLine( this.\_Firstname + " " + this.\_Lastname);

}

}

class FullTimeEmployee : Employee

{

public float yearlyrate;

}

class PartTimeEmployee : Employee

{

public float hourlyrate;

}

class Program

{

static void Main(string[] args)

{

FullTimeEmployee fte = new FullTimeEmployee();

fte.\_Firstname = "hey fn";

fte.\_Lastname = "Hey ln";

fte.yearlyrate = 5000;

fte.fullnameprint();

PartTimeEmployee pte = new PartTimeEmployee();

pte.\_Firstname = "part time";

pte.\_Lastname = " last name";

pte.hourlyrate = 40;

pte.fullnameprint();

Console.ReadKey();

}

}

In the base class, you specify all the re-usable code. In the derived class you will have only the unique fields, properties and methods

Child class is a specialization of parent class.

C# only allows single class inheritance

C# does not support multiple inheritance but multilevel inheritance is possible.

Eg: class A : PartTimeEmployee

{

}

If I create an instance of A, A will have the inheritance from PartTime employee as well as Employee.

C# allows multiple interface inheritance.

Base classes are automatically instantiated before derived classes

Parent class constructor executes before child class constructor.

Eg:

using System;

public class ParentClassConstructor

{

public ParentClassConstructor()

{

Console.WriteLine("parent Class Constructor is called");

}

}

public class childclass : ParentClassConstructor

{

public childclass()

{

Console.WriteLine("The Child class constructor is called is called");

}

}

class Program

{

static void Main(string[] args)

{

childclass cc = new childclass();

Console.ReadKey();

}

}

O/p : parent class constructor is called.

The Child class constructor is called is called

If I have 2 constructors (overloading constructor)one without parameter and other other with parameters in the parent class. And when the child class which inherits parent call is called, by default the parameterless constructor is called. If I want to call the constructor with parameters, then in the derived class you can add a keyword base and pass the parameter. Example below.

using System;

public class ParentClassConstructor

{

public ParentClassConstructor()

{

Console.WriteLine("parent Class Constructor is called");

}

public ParentClassConstructor(String Message)

{

Console.WriteLine(Message);

}

}

public class childclass : ParentClassConstructor

{

public childclass() : base("Derived called child class :This is the one from parameter")

{

Console.WriteLine("The Child class constructor is called is called");

}

}

class Program

{

static void Main(string[] args)

{

childclass cc = new childclass();

Console.ReadKey();

}

}

### 15. Method Hiding

Method Hiding

Invoking hidden base class members.

**If I want to hide the base class member, hide the method.**

Eg: if I have a parent class EMPLOYEE with a print method and a child classes Parttime Employee and Fulltime Employee and the manager says, for all the part time employees, I want an extra – appended. Then we add the same print method in the child class that existed in the parent class and add a new keyword , this hides the actual method in the parent class.

Eg:

using System;

public class Employee

{

public String \_Firstname;

public String \_Lastname;

public String Email;

public void printEmpDetails()

{

Console.WriteLine("EMP full name is {0}, {1}", this.\_Firstname, this.\_Lastname);

}

}

public class FullTimers : Employee

{

}

public class PartTimers : Employee

{

public new void printEmpDetails() // The new keyword in this line hides the print method in the parent method

{

Console.WriteLine("EMP full name is {0}, {1}", this.\_Firstname, this.\_Lastname + "- contracter");

}

}

class res

{

static void Main(string[] args)

{

FullTimers FTE = new FullTimers();

FTE.\_Firstname = "Full";

FTE.\_Lastname = "Timer";

FTE.printEmpDetails();

PartTimers PTE = new PartTimers();

PTE.\_Firstname = "Part";

PTE.\_Lastname = "Timer";

PTE.printEmpDetails();

Console.ReadKey();

}

}

Now if I want to call the hidden method from the parent class, you do it by calling the keyword base.the methodname and comment the content within the derived.

Eg:

using System;

public class Employee

{

public String \_Firstname;

public String \_Lastname;

public String Email;

public void printEmpDetails()

{

Console.WriteLine("EMP full name is {0}, {1}", this.\_Firstname, this.\_Lastname);

}

}

public class FullTimers : Employee

{

}

public class PartTimers : Employee

{

public new void printEmpDetails() // The new keyword in this line hides the print method in the parent method

{

base.printEmpDetails();

//Console.WriteLine("EMP full name is {0}, {1}", this.\_Firstname, this.\_Lastname + "- contracter");

}

}

class res

{

static void Main(string[] args)

{

FullTimers FTE = new FullTimers();

FTE.\_Firstname = "Full";

FTE.\_Lastname = "Timer";

FTE.printEmpDetails();

PartTimers PTE = new PartTimers();

PTE.\_Firstname = "Part";

PTE.\_Lastname = "Timer";

PTE.printEmpDetails();

Console.ReadKey();

}

}

The other way of using the method within the parent class is to use the type cast converter.

Eg:

using System;

public class Employee

{

public String \_Firstname;

public String \_Lastname;

public String Email;

public void printEmpDetails()

{

Console.WriteLine("EMP full name is {0}, {1}", this.\_Firstname, this.\_Lastname);

}

}

public class FullTimers : Employee

{

}

public class PartTimers : Employee

{

public new void printEmpDetails() // The new keyword in this line hides the print method in the parent method

{

// base.printEmpDetails();

Console.WriteLine("EMP full name is {0}, {1}", this.\_Firstname, this.\_Lastname + "- contracter");

}

}

class res

{

static void Main(string[] args)

{

FullTimers FTE = new FullTimers();

FTE.\_Firstname = "Full";

FTE.\_Lastname = "Timer";

FTE.printEmpDetails();

PartTimers PTE = new PartTimers();

PTE.\_Firstname = "Part";

PTE.\_Lastname = "Timer";

((Employee)PTE).printEmpDetails(); // This is the type caste method of calling the base class method.

Console.ReadKey();

}

}

The other way to call the base method is : Instead of using A CHILD class reference variable, I can use a parent class reference variable. Employee PTE = new PartTimers();

Eg:

using System;

public class Employee

{

public String \_Firstname;

public String \_Lastname;

public String Email;

public void printEmpDetails()

{

Console.WriteLine("EMP full name is {0}, {1}", this.\_Firstname, this.\_Lastname);

}

}

public class FullTimers : Employee

{

}

public class PartTimers : Employee

{

public new void printEmpDetails() // The new keyword in this line hides the print method in the parent method

{

// base.printEmpDetails();

Console.WriteLine("EMP full name is {0}, {1}", this.\_Firstname, this.\_Lastname + "- contracter");

}

}

class res

{

static void Main(string[] args)

{

FullTimers FTE = new FullTimers();

FTE.\_Firstname = "Full";

FTE.\_Lastname = "Timer";

FTE.printEmpDetails();

Employee PTE = new PartTimers();

PTE.\_Firstname = "Part";

PTE.\_Lastname = "Timer";

PTE.printEmpDetails();

Console.ReadKey();

}

}

### 16. Polymorphism

The base classes are virtual and the derived are override is Polymorphism. Polymorphism enables you to invoke derived classes methods through base class variables at run time.

Polymorphism enables us To invoke derived class methods using a base class reference variable at run time.

In base class – we make any method as virtual and in the child classes, we change the methods to overridden. If any childclass does not have a method or override, the base class method will be used.

using System;

public class Employee

{

public String \_Firstname = "FN";

public String \_Lastname = "LN";

public virtual void printname()

{

Console.WriteLine("The First and last name is {0}, {1}", \_Firstname, \_Lastname);

}

}

public class PartTimeEmp : Employee

{

public override void printname()

{

Console.WriteLine("The First and last name is {0}, {1}", \_Firstname, \_Lastname + "-Partimer");

}

}

public class FullTimeEmp : Employee

{

public override void printname()

{

Console.WriteLine("The First and last name is {0}, {1}", \_Firstname, \_Lastname + "-FullTime");

}

}

public class TempEmp : Employee

{

public void printname()

{

Console.WriteLine("The First and last name is {0}, {1}", \_Firstname, \_Lastname + "-Temp");

}

}

class Program

{

static void Main(string[] args)

{

Employee[] E = new Employee[4]; // Declare base class array

// Since this is Array, the number of objects can declare be declared once Array is created.

E[0] = new Employee(); // first object

E[1] = new PartTimeEmp();

E[2] = new FullTimeEmp();

E[3] = new TempEmp();

foreach (Employee emp in E)

{

emp.printname();

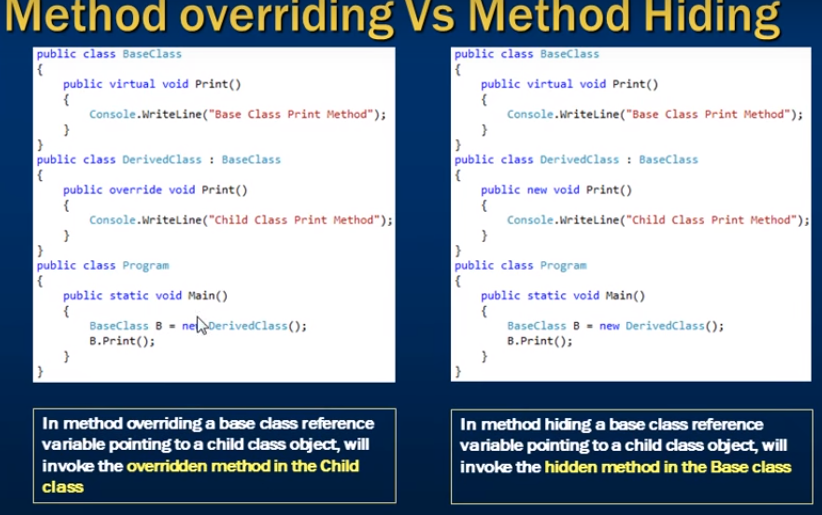
}

Console.ReadKey();

}

}

### 17. Method Overriding Vs Method Hiding



using System;

public class BaseClass

{

public virtual void printname()

{

Console.WriteLine("Im a base class");

}

}

public class DerivedClass: BaseClass

{

public override void printname()

{

Console.WriteLine("Im a derived class");

}

}

class Program

{

static void Main(string[] args)

{

// to call the overriden method, point the ref of base class to child class

BaseClass B = new DerivedClass();

B.printname();

Console.ReadKey();

DerivedClass D = new DerivedClass();

D.printname();

}

}

### 18. Method Overloading

We can overload a function based on number of parameters differing from each method on the same method name.

Methods can also be overloaded based on the data type. Such as same function name with different data types. Such as one INT and one Float.

A method can also be overloaded based on the kind of parameters such as output, ref, value and parameter arrays.

class Program

{

static void Main(string[] args)

{

}

public static void add(int fn, int sn)

{

Console.WriteLine("Sum = {0}", fn, sn);

}

public static void add(int fn, int sn, int tn)

{

Console.WriteLine("Sum = {0}", fn, sn);

}

}

Different type parameters:

using System;

class Program

{

static void Main(string[] args)

{

}

public static void add(int fn, int sn)

{

Console.WriteLine("Sum = {0}", fn, sn);

}

public static void add(float fn, float sn)

{

Console.WriteLine("Sum = {0}", fn, sn);

}

}

Different types parameters:

using System;

class Program

{

static void Main(string[] args)

{

}

public static void add(int fn, int sn)

{

Console.WriteLine("Sum = {0}", fn, sn);

}

public static void add(float fn, int sn)

{

Console.WriteLine("Sum = {0}", fn, sn);

}

}

Different kinds of parameters

using System;

class Program

{

static void Main(string[] args)

{

}

public static void add(int fn, int sn, int tn)

{

Console.WriteLine("Sum = {0}", fn, sn);

}

public static void add(int fn, int sn, out int sum)

{

Console.WriteLine("Sum = {0}", fn, sn);

sum = fn + sn;

}

}

The signature of a method is formed by name, type, kind and number but not by the return type and params keyword (value, ref, etc)

You cannot overload a function just based on a return type of a function.

A method can also be not build just based on params keyword.

### 19. Why Properties

**Marking the class fields public and exposing to the external world is bad as you will not have control over what gets assigned and returned.**

Eg: A student ID cannot be negative. If I give public, it allows negative, similarly a student name cannot be empty and if empty cannot return a name. If it is set to public, it allows empty and does return some name. Change a value of a student passmark. If it is set to public, I can change the value. Like below:

using System;

public class Student

{

public int ID;

public String name;

public int marks;

}

public class program

{

public static void Main()

{

Student C1 = new Student();

C1.ID = -101;

C1.name = null;

C1.marks = 0;

Console.WriteLine("the ID is {0}, && the name is {1}, && the marks is {2}"

, C1.ID, C1.name, C1.marks);

Console.ReadKey();

}

}

So all violations of the business is occurred. So you want to turn your public fields into private and have a control what gets executed in these fields and what gets out of these fields. In general in the programming languages where there is no properties, it uses a getter and setter methods to encapsulate and protect the fields.

Following is the example how it used to work with getter and setter methods before properties. The following get and set id’s protect the ID field as it’s made private.

ID cannot be negative, name cannot be empty and marks should be readonly.

using System;

public class Student

{

private int \_ID;

private String \_name;

private int \_marks;

public int getmarks() // read only

{

return this.\_marks;

}

public void setID(int id)

{

if (this.\_ID <= 0)

{

throw new Exception("The ID cannot be negative");

}

this.\_ID = id;

}

public int getID()

{

return this.\_ID;

}

public void setName(String name)

{

if (String.IsNullOrEmpty(name))

{

throw new Exception("Name cannot be empty");

}

this.\_name = name;

}

public string getName()

{

// String.IsNullOrEmpty(this.\_name) ? "No Name" : this.\_name; // terinary operator.

if (String.IsNullOrEmpty(this.\_name))

{

return "No Name";

} else

{

return this.\_name;

}

}

}

public class program

{

public static void Main()

{

Student C1 = new Student();

C1.setID(-101); // throws Exception

C1.setID(101);

C1.setName(null); // throws exception

C1.setName("somename"); //sets name

// On the other hand if the name is not specified at all, then it should return no name in the get emthod

Console.WriteLine("The student ID is {0}", C1.getID());

Console.WriteLine("The student ID is {0}", C1.getName());

Console.WriteLine("The student ID is {0}", C1.getmarks()); // gets read only marks

//Console.WriteLine("the ID is {0}, && the name is {1}, && the marks is {2}"

// , C1.ID, C1.name, C1.marks);

Console.ReadKey();

}

}

**NEVER NEVER expose your class fields as public as we will not have any control to these fields and what goes out. To have a better control and encapsulate those fields then we make use of properties in C#. In programming languages that does not support properties we use these setter and getter methods.**

**Encapsulation is one of the 4 pillars of OOPS.**

**But in C# we can achieve these without having to use getter and setter methods.**

### 20. Properties in C#

Read/Write Properties

Read Only properties

Write Only properties

Auto Implemented Properties.

In C# to encapsulate and protect fields we use properties.

1. We use get and set assessors 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.

The advantage over traditional get, set method is that you can access them as if they were public fields.

using System;

public class Student

{

private int \_ID;

private String \_name;

private int \_marks;

public int marks // read only

{

get{

return this.\_marks;

}

}

public int id // set a return type and you dont pass parameters

{

set

{

if (value <= 0) // Value is inbuilt kw what the data being sent to retrieved (the one used in place of parameters.)

{

throw new Exception("The ID cannot be negative");

}

this.\_ID = value;

}

get

{

return this.\_ID;

}

}

public String Name

{

set

{

if (String.IsNullOrEmpty(value))

{

throw new Exception("Name cannot be empty");

}

this.\_name = value;

}

get

{

return String.IsNullOrEmpty(this.\_name) ? "No Name" : this.\_name; // terinary operator.

}

}

}

public class program

{

public static void Main()

{

Student C1 = new Student();

C1.id = 101;

// C1.setName(null); // throws exception

C1.Name = "mark";

// On the other hand if the name is not specified at all, then it should return no name in the get emthod

Console.WriteLine("The student ID is {0}", C1.id);

Console.WriteLine("The student ID is {0}", C1.Name);

Console.WriteLine("The student ID is {0}", C1.marks); // gets read only marks

//Console.WriteLine("the ID is {0}, && the name is {1}, && the marks is {2}"

// , C1.ID, C1.name, C1.marks);

Console.ReadKey();

}

}

### 21. Structs

Just like classes, a struct can have private fields, public properties, constructors, methods.

**Object Initialiser syntax**: Currently if we have a constructor, we either send the input as parameters to the object or create an object instance and initialise data. Instead we can use object initilaiser

Syntax: customer c1 = new customer

{ ID = 10; Name = “something” };

C1.print details

using System;

public struct customer

{

// create private fields

private int \_id;

private String \_name;

// Encapsute the private fields by using the properties

public int id

{

get { return this.\_id; }

set { this.\_id = value; }

}

public String name

{

get { return this.\_name; }

set { this.\_name = value; }

}

// constructor to initialise the data fields

public customer(int id, string name)

{

this.\_id = id;

this.\_name = name;

}

// A method :

public void printdetails()

{

Console.WriteLine("ID is {0} && nme is {1}", this.\_id, this.\_name);

}

}

public class Program

{

static void Main(string[] args)

{

// first method of creating object with paramenetrs

customer c1 =new customer(101, "name1");

c1.printdetails();

// method2 without sendign parameters:

customer c2 = new customer();

c2.id = 210;

c2.name = "name2";

c2.printdetails();

// object initialiser syntax:

customer c3 = new customer

{

id = 103 ,

name = "test3",

};

c3.printdetails();

Console.ReadKey();

}

}

**Object Initialiser syntax**: can be used to initialise a structure or a class.

Classes Vs Structure

var payload = new Dictionary<string, object> { { "Username", "cheese" } };

var payload = new LdapUserValidationPayload { Username = username, Password = password, AdditionalSettings = additionalSettings };

### 22. Interfaces:

1. Interfaces will only have method, properties etc declarations but no implementations.
2. Interface members are public by default and you cannot have a keyword public declared or an explicit access modifier.
3. They are public by default.
4. Interfaces cannot have fields.
5. Classes can inherit from inheritences
6. When a inheritance is inherited into a class, the method signature in the interface must match the method in the class which is inheriting. For example, if an interface has a method with no parameters, the class also must have the same signature of the method which is with no parameters.
7. If a class or a struct inherits from an interface, it must provide implementation for all interfaces members otherwise we get a compiler error.
8. A class or a struct can inherit more than 1 inheritance at the same time.
9. A class cannot inherit from more than one class at the same time. (cannot have multiple bases classes where as a class can have multiple interfaces at the same time.)
10. If an interface inherits from other interfaced and the class inherits from that interface, then then class must provide implementation for both the inherits.
11. \*\*\* We cannot create an instance of an interface, but an interface reference variable can point to a derived class object. Where as a I can use the inheritance reference can point to a derived object.

Syntax: interface InameofInterface.

//Multiple Interface in one class.

using System;

using System.Security.Cryptography.X509Certificates;

interface ICustomer1

{

void printI1();// Cannot have a public indentifier in an interface as its public by default

}

interface i2

{

void i2method();

}

public class customer1 : ICustomer1, i2

{

public void printI1() // Name signature of the method should be the same as in the interface.

{

Console.WriteLine("interface1");

}

public void i2method()

{

Console.WriteLine("interface2");

}

}

class Program

{

static void Main(string[] args)

{

customer1 C1 = new customer1();

C1.printI1();

C1.i2method();

ICustomer1 C2 = new customer1();

C2.printI1();

i2 c3 = new customer1();

c3.i2method();

Console.ReadKey();

}

}

ANotehr Excample:

using System;

interface I1

{

void printI1();

}

interface I2

{

void printI2();

}

public class customer : I1

{

public void printI1()

{

Console.WriteLine("Inteeface1");

}

public void printI2()

{

Console.WriteLine("Interface2");

}

}

class Program

{

static void Main(string[] args)

{

customer c1 = new customer();

c1.printI1();

c1.printI2();

I1 i1 = new customer();

i1.printI1();

}

}

### 23. Explicit interface Implementation

using System;

interface I1

{

void print();

}

interface I2

{

void print();

}

public class customer : I1, I2

{

void I1.print() // since this is implementation of 2 same interfaces, remove public and put the interface name

{

Console.WriteLine("Inteeface1");

}

void I2.print() // since this is implementation of 2 same interfaces, remove public and put the interface name

{

Console.WriteLine("Inteeface2");

}

}

class Program

{

static void Main(string[] args)

{

customer C = new customer();

((I1)C).print(); // explicit interfaces

((I2)C).print(); // explicit interfaces

Console.ReadLine();

}

}

//Other way instead of type cast

using System;

interface I1

{

void print();

}

interface I2

{

void print();

}

public class customer : I1, I2

{

void I1.print() // since this is implementation of 2 same interfaces, remove public and put the interface name

{

Console.WriteLine("Inteeface1");

}

void I2.print() // since this is implementation of 2 same interfaces, remove public and put the interface name

{

Console.WriteLine("Inteeface2");

}

}

class Program

{

static void Main(string[] args)

{

//Other way instead of type cast

I1 C1 = new customer();

I2 C2 = new customer();

C1.print();

C2.print();

Console.ReadLine();

}

}

**If I want to make one of the interface with 2 similar names unique, declare the default one as normal.**

using System;

using System.Runtime.Remoting;

interface I1

{

void print();

}

interface I2

{

void print();

}

public class customer : I1, I2

{

public void print() // to make this default of the 2 similar interfaces

{

Console.WriteLine("Inteeface1");

}

void I2.print() // since this is implementation of 2 same interfaces, remove public and put the interface name

{

Console.WriteLine("Inteeface2");

}

}

class Program

{

static void Main(string[] args)

{

//Other way instead of type cast

// I1 C1 = new customer();

I2 C2 = new customer();

customer C1 = new customer(); // default interface

C1.print();

C2.print();

Console.ReadLine();

}

}

### 24. Abstract Classes

Abstract classes cannot be instantiated.

The Abstract class has the keyword as Abstract.

They can only be used as bases classes for other classes.

Any abstract members cannot have implementation.

If a non abstract class inherits an abtract class, then the non abstract class has to provide implementation for all the abstract members within the base class.

If a class inherits an abstract class, ther are 2 options available for that class:

**Option1** : Provide implementation for all the abstract members inherited from the base abstract class

**Option2**: If the class does not wish to provie implementation for all the abstract members inherited from the abstract class, then the class has to be marked as abstract.

using System;

public abstract class Customer

{

public abstract void Print(); // if I use the abstract keyword, this method cannot have implementation.

}

public class Program : Customer // this is a non abstract class inheriting from an abstract class and this class has to provide implementation to the method.

{

public static void Main()

{

}

}

using System;

public abstract class Customer

{

public abstract void Print(); // if I use the abstract keyword, this method cannot have implementation.

}

public abstract class Program : Customer // If I dont want to provide implementation, then this class has to be a abstract

{

public static void Main()

{

}

}

using System;

public abstract class Customer

{

public abstract void Print(); // if I use the abstract keyword, this method cannot have implementation.

}

public class Program : Customer // Provide implementation

{

public override void Print() //override as it hides the base inherited class.

{

Console.WriteLine("Print Method");

}

public static void Main()

{

Program P = new Program();

P.Print();

Customer C = new Program();

C.Print();

}

}

\*\*\*\*\* An Abstract class cannot be sealed.

From inheritance concept, we learnt that If you want to prevent your class from beinf ingerited by another class, then we use a sealed keyword. So if we use the sealed keyword, then that class cannot be used as a base class.

By definition, an abstract class is a base class.

As a gist :

Abstract KW means – this class can be used as a base class

Sealed KW means - this class cannot be used as a base class.

An abstract class may contain the abstract members, methods, fields , inders and events. etc but is not mandatory.

A non abstract class from an abstract class must provide implemetauon from al inherited abstract members.

### 25. Abstract Classes vs Interfaces

1. Abstract class can have implementation for some of its members but an interface cant have implementation for any of its members.
2. An Abstract class member is a private by default.
3. An interface cannot have fields where as an abstract class can have fields where as an Abstract class can have fields.
4. An interface can inherit from another interface only and cannot inherit from an abstract class where as an abstract class inherit from another abstract class or another interface.
5. A class can inherit from multiple interfaces at the same time but a class cannot inherit from multiple classes at the same time.
6. Abstract class members can have access modifiers where as interface members cannot have access modifiers.

Problems of Multiple class Inheritecne

If we mark a base class as virtual, the derived class can override it. If we have class A, Class B inheritring from Class C. And if Cllass D has to inherit from A, B, it is s diamonf problem.

### 26. Delegates

A **delegate** is a type safe function pointer.That is, they hold 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.*

**Sample Delegate Program:**  
  
using System;  
  
  
// Delegate Declaration.   
public delegate void HelloFunctionDelegate(string Message);  
  
  
class Pragim  
{  
    public static void Main()  
    {  
        // Create the instance of the delegate and pass in the function  
        // name as the parameter to the constructor. The passed in  
        // function signature must match the signature of the delegate  
        HelloFunctionDelegate del = new HelloFunctionDelegate(Hello);  
        // Invoke the delegate, which will invoke the method  
        del("Hello from Delegte");  
    }  
  
  
    public static void Hello(string strMessge)  
    {  
        Console.WriteLine(strMessge);  
    }  
}

### 26.1 Delegates Example1

Method sent as a parameter.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace ConsoleApplication7

{

class Program

{

static void Main(string[] args)

{

List<Employee> allEmployees = new List<Employee>();

allEmployees.Add(new Employee() { ID = 101, name = "name1", Sal = 4000, Exp = 4});

allEmployees.Add(new Employee() { ID = 102, name = "name2", Sal = 6000, Exp = 10 });

allEmployees.Add(new Employee() { ID = 103, name = "name3", Sal = 2000, Exp = 3});

allEmployees.Add(new Employee() { ID = 104, name = "name4", Sal = 7000, Exp = 5 });

allEmployees.Add(new Employee() { ID = 105, name = "name5", Sal = 1000, Exp = 6 });

allEmployees.Add(new Employee() { ID = 106, name = "name6", Sal = 8000, Exp = 9 });

Employee.promoEmp(allEmployees);

}

}

class Employee

{

public int ID { get; set; }

public string name { get; set; }

public int Sal { get; set; }

public int Exp { get; set; }

public static void promoEmp(List<Employee> emplist)

{

foreach (Employee e in emplist)

{

if (e.Exp > 5)

{

Console.WriteLine(e.name + "is promoted");

}

}

Console.ReadKey();

}

}

}

o/p : name2 is promoted,name5 ispromoted, name6 is promoted

### 26.2 Delegates Example2

We want methods to be re-uable instead of hardcoded as in example1.

In example1, we have a if statement which will return a true or a false, so we have the return type of a delegate as bool.

Pass the delegate as a parameter to the function

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace ConsoleApplication7

{

using System.Security.Cryptography;

class Program

{

static void Main(string[] args)

{

List<Employee1> l1 = new List<Employee1>();

l1.Add(new Employee1() { ID = 101, name = "name1", Exp = 5, sal = 5000});

l1.Add(new Employee1() { ID = 102, name = "name2", Exp = 6, sal = 6000 });

l1.Add(new Employee1() { ID = 103, name = "name3", Exp = 7, sal = 7000 });

l1.Add(new Employee1() { ID = 104, name = "name4", Exp = 2, sal = 2000 });

l1.Add(new Employee1() { ID = 105, name = "name5", Exp = 3, sal = 3000 });

Employee1.emppromo(l1, x => x.Exp >=5);

}

public delegate bool Ispromotable(Employee1 delE1);

}

internal class Employee1

{

public int ID { get; set; }

public string name { get; set; }

public int Exp { get; set; }

public int sal { get; set; }

public static void emppromo(List<Employee1> emp1list, Program.Ispromotable xyz)

{

foreach (Employee1 e1 in emp1list)

{

if (xyz(e1))

{

Console.WriteLine(e1.name + "is promoted");

}

}

Console.ReadKey();

}

public bool Ispromotablemethod(Employee1 delMetE1)

{

if (delMetE1.Exp >= 5)

{

return true;

}

else

{

return false;

}

}

}

}

**Example without lamda expression:**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace ConsoleApplication7

{

using System.Security.Cryptography;

class Program

{

static void Main(string[] args)

{

List<Employee1> l1 = new List<Employee1>();

l1.Add(new Employee1() { ID = 101, name = "name1", Exp = 5, sal = 5000});

l1.Add(new Employee1() { ID = 102, name = "name2", Exp = 6, sal = 6000 });

l1.Add(new Employee1() { ID = 103, name = "name3", Exp = 7, sal = 7000 });

l1.Add(new Employee1() { ID = 104, name = "name4", Exp = 2, sal = 2000 });

l1.Add(new Employee1() { ID = 105, name = "name5", Exp = 3, sal = 3000 });

Ispromotable del1 = new Ispromotable(Ispromotablemethod);

Employee1.emppromo(l1, del1);

//Employee1.emppromo(l1, x => x.Exp >=5);

}

public static bool Ispromotablemethod(Employee1 delMetE1)

{

if (delMetE1.Exp >= 5)

{

return true;

}

else

{

return false;

}

}

public delegate bool Ispromotable(Employee1 delE1);

}

class Employee1

{

public int ID { get; set; }

public string name { get; set; }

public int Exp { get; set; }

public int sal { get; set; }

public static void emppromo(List<Employee1> emp1list, Program.Ispromotable xyz)

{

foreach (Employee1 e1 in emp1list)

{

if (xyz(e1))

{

Console.WriteLine(e1.name + "is promoted");

}

}

Console.ReadKey();

}

}}

### 27. 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.**  
  
  
**Approach 1:**  
using System;  
namespace Sample  
{  
    public delegate void SampleDelegate();  
      
    public class Sample  
    {  
        static void Main()  
        {  
            SampleDelegate del1 = new SampleDelegate(SampleMethodOne);  
            SampleDelegate del2 = new SampleDelegate(SampleMethodTwo);  
            SampleDelegate del3 = new SampleDelegate(SampleMethodThree);  
            // In this example del4 is a multicast delegate. You use +(plus)   
            // operator to chain delegates together and -(minus) operator to remove.  
            SampleDelegate del4 = del1 + del2 + del3 - del2;  
              
            del4();  
        }  
  
  
        public static void SampleMethodOne()  
        {  
            Console.WriteLine("SampleMethodOne Invoked");  
        }  
  
  
        public static void SampleMethodTwo()  
        {  
            Console.WriteLine("SampleMethodTwo Invoked");  
        }  
  
  
        public static void SampleMethodThree()  
        {  
            Console.WriteLine("SampleMethodThree Invoked");  
        }  
    }  
}

**Approach 2:**  
using System;  
namespace Sample  
{  
    public delegate void SampleDelegate();  
      
    public class Sample  
    {  
        static void Main()  
        {  
            // In this example del is a multicast delegate. You use += operator   
            // to chain delegates together and -= operator to remove.  
            SampleDelegate del = new SampleDelegate(SampleMethodOne);  
            del += SampleMethodTwo;  
            del += SampleMethodThree;  
            del -= SampleMethodTwo;  
              
            del();  
        }  
  
  
        public static void SampleMethodOne()  
        {  
            Console.WriteLine("SampleMethodOne Invoked");  
        }  
  
  
        public static void SampleMethodTwo()  
        {  
            Console.WriteLine("SampleMethodTwo Invoked");  
        }  
  
  
        public static void SampleMethodThree()  
        {  
            Console.WriteLine("SampleMethodThree Invoked");  
        }  
    }  
}  
  
  
***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 parameter, the value of the output parameter, will be the value assigned by the last method.**  
  
  
**Example:** Multicast delegate with an int return type  
using System;  
namespace Sample  
{  
    // Deletegate's return type is int  
    public delegate int SampleDelegate();  
      
    public class Sample  
    {  
        static void Main()  
        {  
            SampleDelegate del = new SampleDelegate(SampleMethodOne);  
            del += SampleMethodTwo;  
  
  
            // The ValueReturnedByDelegate will be 2, returned by the SampleMethodTwo(),  
            // as it is the last method in the invocation list.  
            int ValueReturnedByDelegate = del();  
  
  
            Console.WriteLine("Returned Value = {0}", ValueReturnedByDelegate);  
        }  
  
  
        // This method returns one  
        public static int SampleMethodOne()  
        {  
            return 1;              
        }  
  
  
        // This method returns two  
        public static int SampleMethodTwo()  
        {  
            return 2;  
        }  
    }  
}  
  
  
**Example:** Multicast delegate with an integer output parameter.  
using System;  
namespace Sample  
{  
    // Deletegate has an int output parameter  
    public delegate void SampleDelegate(out int Integer);  
      
    public class Sample  
    {  
        static void Main()  
        {  
            SampleDelegate del = new SampleDelegate(SampleMethodOne);  
            del += SampleMethodTwo;  
  
  
            // The ValueFromOutPutParameter will be 2, initialized by SampleMethodTwo(),  
            // as it is the last method in the invocation list.  
            int ValueFromOutPutParameter = -1;  
            del(out ValueFromOutPutParameter);  
  
  
            Console.WriteLine("Returned Value = {0}", ValueFromOutPutParameter);  
        }  
  
  
        // This method sets ouput parameter Number to 1  
        public static void SampleMethodOne(out int Number)  
        {  
            Number = 1;  
        }  
  
  
        // This method sets ouput parameter Number to 2  
        public static void SampleMethodTwo(out int Number)  
        {  
            Number = 2;  
        }  
    }  
}  
  
  
**Where do you use multicast delegates? This is a very common interview question.**  
**Answer:** Multicast delegate makes implementation of observer design pattern very simple. Observer pattern is also called as publish/subscribe pattern.

### 28. Exception Handling

**What is an exception?**  
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**.  
  
  
**Program without exception handling**  
using System;  
using System.IO;  
class ExceptionHandling  
{  
    public static void Main()  
    {  
        //This line will throw FileNotFoundException  
        StreamReader streamReader = new StreamReader("C:\\NonExistingFile.txt");  
        Console.WriteLine(streamReader.ReadToEnd());  
        //Closes the underlying stream and releases the system resources.  
        //If there is an exception before this line, the below line will never  
 //be executed and the resources are not relased  
        streamReader.Close();  
    }  
}

**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  
  
  
using System;  
using System.IO;  
class ExceptionHandling  
{  
    public static void Main()  
    {  
        StreamReader streamReader = null;  
        try  
        {  
            // This line will throw FileNotFoundException  
            streamReader = new StreamReader("C:\\NonExistingFile.txt");  
            Console.WriteLine(streamReader.ReadToEnd());  
        }  
        // This catch block handles only FileNotFoundException  
        catch (FileNotFoundException fileNotFoundException)  
        {  
            // Log or email the exception  
            // Code to log or email exception details  
  
  
            // Display meaningful error message to the end user  
            Console.WriteLine("Please check if the file \"{0}\" is present", fileNotFoundException.FileName);  
        }  
        // This catch block handles all the other exceptions  
        catch (Exception exception)  
        {  
            Console.WriteLine(exception.Message);  
        }  
        finally  
        {  
            if (streamReader != null) // If stream reader is null, then the stream reader cannot close.  
            {  
                streamReader.Close();  
            }  
        }  
    }  
}

**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.  
  
  
**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.  
  
  
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.  
  
  
***Note:*** It is a good practice to always release resources in the finally block, because finally block is guarenteed to execute, irrespective of whether there is an exception or not.  
  
  
**Bad way of cleaning resources.**  
using System;  
using System.IO;  
class ExceptionHandling  
{  
    public static void Main()  
    {  
        StreamReader streamReader = null;  
        try  
        {  
            streamReader = new StreamReader("C:\\NonExistingFile.txt");  
            Console.WriteLine(streamReader.ReadToEnd());  
        }  
        catch (Exception)  
        {  
            throw new Exception("Intentional Exception");  
        }  
        //This code will never be executed, hence it is always a good  
        //practice to release resources in the finally block  
        streamReader.Close();  
    }  
}

### 29. Inner Exceptions

using System;  
using System.IO;  
class ExceptionHandling  
{  
    public static void Main()  
    {  
        try  
        {  
            try  
            {  
                Console.WriteLine("Enter First Number");  
                int FN = Convert.ToInt32(Console.ReadLine());  
  
  
                Console.WriteLine("Enter Second Number");  
                int SN = Convert.ToInt32(Console.ReadLine());  
  
  
                int Result = FN / SN;  
                Console.WriteLine("Result = {0}", Result);  
            }  
            catch (Exception ex)  
            {  
                string filePath = @"C:\Sample Files\Log.txt";  
                if (File.Exists(filePath))  
                {  
                    StreamWriter sw = new StreamWriter(filePath);  
                    sw.Write(ex.GetType().Name + ex.Message + ex.StackTrace);  
                    sw.Close();  
                    Console.WriteLine("There is a problem! Plese try later");  
                }  
                else  
                {  
                    //To retain the original exception pass it as a parameter  
                    //to the constructor, of the current exception  
                    throw new FileNotFoundException(filePath + " Does not Exist", ex);  
                }  
            }  
        }  
        catch (Exception ex)  
        {  
            //ex.Message will give the current exception message  
            Console.WriteLine("Current or Outer Exception = " + ex.Message);  
  
  
            //Check if inner exception is not null before accessing Message property  
            //else, you may get Null Reference Excception  
            if(ex.InnerException != null)  
            {  
                Console.WriteLine("Inner Exception = ", ex.InnerException.Message);  
            }  
        }  
    }  
}

### 30. 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.   
  
  
**Consider that**  
**1.** I have an asp.net web application.  
**2.** The application should allow the user to have only one logged in session.  
**3.** If the user is already logged in, and if he opens another browser window and tries to login again, the application should throw an error stating he is already logged in another browser window.  
  
  
With in the .NET framework we donot have any exception, that adequately describes this problem. So this scenario is one of the examples where you want to create a custom exception.  
  
  
We know that an exception is a class. So to create a 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. If you don't do so, you won't get a compiler error, but you will be deviating from the guidelines for creating custom exceptions.**  
public class UserAlreadyLoggedInException : Exception  
{  
}  
  
  
**2. Provide a public constructor, that takes in a string parameter. This constructor simply passes the string parameter, to the base exception class constructor.**  
public UserAlreadyLoggedInException(string message)  
        : base(message)  
{  
}  
  
  
**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 as shown below. If you are new to Constructor Overloading, please watch this video.**  
public UserAlreadyLoggedInException(string message, Exception innerException)  
    : base(message, innerException)  
{  
}  
  
  
**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.**  
[Serializable]  
public class UserAlreadyLoggedInException : Exception  
{  
    public UserAlreadyLoggedInException(SerializationInfo info, StreamingContext context)  
        : base(info, context)  
    {  
    }  
}  
  
  
***Note:*** *It is also possible to provide your own custom serialization, which will discuss in a later session.*  
  
  
**Complete Example of creating a custom exception:**  
using System;  
using System.Runtime.Serialization;  
  
  
public class CustomExceptions  
{  
    public static void Main()  
    {  
        try  
        {  
            throw new UserAlreadyLoggedInException("User Already logged in");  
        }  
        catch (UserAlreadyLoggedInException ex)  
        {  
            Console.WriteLine(ex.Message);  
        }  
    }  
}  
  
  
[Serializable]  
public class UserAlreadyLoggedInException : Exception  
{  
    public UserAlreadyLoggedInException(string message)  
        : base(message)  
    {  
    }  
  
  
    public UserAlreadyLoggedInException(string message, Exception innerException)  
        : base(message, innerException)  
    {  
    }  
      
    public UserAlreadyLoggedInException(SerializationInfo info, StreamingContext context)  
        : base(info, context)  
    {  
    }  
}

### 31. 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.   
  
  
But many a times I have seen programmers using exception handling to implement **programming logic which is bad, and this is called as exception handling abuse**.

**Program using exception handling to implement logical flow:**  
using System;  
public class ExceptionHandlingAbuse  
{  
    public static void Main()  
    {  
        try  
        {  
            Console.WriteLine("Please enter Numerator");  
            int Numerator = Convert.ToInt32(Console.ReadLine());  
  
  
            Console.WriteLine("Please enter Denominator");  
            //Convert.ToInt32() can throw FormatException, if the entered value  
            //cannot be converted to integer. So use int.TryParse() instead  
            int Denominator = Convert.ToInt32(Console.ReadLine());  
  
  
            int Result = Numerator / Denominator;  
  
  
            Console.WriteLine("Result = {0}", Result);  
        }  
        catch (FormatException)  
        {  
            Console.WriteLine("Only numbers are allowed!");  
        }  
        catch (OverflowException)  
        {  
            Console.WriteLine("Only numbers between {0} & {1} are allowed",  
                Int32.MinValue, Int32.MaxValue);  
  
  
        }  
        catch (DivideByZeroException)  
        {  
            Console.WriteLine("Denominator cannot be zero");  
        }  
        catch (Exception ex)  
        {  
            Console.WriteLine(ex.Message);  
        }  
    }  
}

### 32. Preventing Exception Handling Abuse

**Rewritten example that doesn't use exception handling to control program's logical flow.**  
using System;  
public class ExceptionHandlingAbuse  
{  
    public static void Main()  
    {  
        try  
        {  
            Console.WriteLine("Please enter Numerator");  
            int Numerator;  
            //int.TryParse() will not throw an exception, instead returns false  
            //if the entered value cannot be converted to integer  
            bool isValidNumerator = int.TryParse(Console.ReadLine(), out Numerator);  
  
  
            if (isValidNumerator)  
            {  
                Console.WriteLine("Please enter Denominator");  
                int Denominator;  
                bool isValidDenominator = int.TryParse(Console.ReadLine(), out Denominator);  
  
  
                if (isValidDenominator && Denominator != 0)  
                {  
                    int Result = Numerator / Denominator;  
                    Console.WriteLine("Result = {0}", Result);  
                }  
                else  
                {  
                    //Check if the denominator is zero and print a friendly error  
                    //message instead of allowing DivideByZeroException exception   
                    //to be thrown and then printing error message to the user.  
                    if (isValidDenominator && Denominator == 0)  
                    {  
                        Console.WriteLine("Denominator cannot be zero");  
                    }  
                    else  
                    {  
                        Console.WriteLine("Only numbers between {0} && {1} are allowed",  
                            Int32.MinValue, Int32.MaxValue);  
                    }  
                }  
            }  
            else  
            {  
                Console.WriteLine("Only numbers between {0} && {1} are allowed",  
                            Int32.MinValue, Int32.MaxValue);  
            }  
        }  
        catch (Exception ex)  
        {  
            Console.WriteLine(ex.Message);  
        }  
    }  
}

### 33. Why Enums

**Enums are strongly typed constants**. Let's understand enums with an example. I have a customer class with Name and Gender properties. Gender is an integer.   
**0 is an Unknown gender**  
**1 is Male**  
**2 is Female**  
  
This program is less readable and maintainable, as it operates on integrals instead of using **enums**.  
  
In the next session we will replace, these integral numbers with enums, which makes the program better readable and maintainable.

using System;  
public class Enums  
{  
    public static void Main()  
    {  
        Customer[] customers = new Customer[3];  
        customers[0] = new Customer()  
        {  
            Name = "Mark",  
            Gender = 1  
        };  
        customers[1] = new Customer()  
        {  
            Name = "Mary",  
            Gender = 2  
        };  
        customers[2] = new Customer()  
        {  
            Name = "Sam",  
            Gender = 0  
        };  
        foreach (Customer customer in customers)  
        {  
            Console.WriteLine("Name = {0} && Gender = {1}", customer.Name, GetGender(customer.Gender));  
        }  
    }  
  
  
    public static string GetGender(int gender)  
    {  
        // The swicth here is less readable because of these integral numbers  
        switch (gender)  
        {  
            case 0:  
                return "Unknown";  
            case 1:  
                return "Male";  
            case 2:  
                return "Female";  
            default:  
                return "Invalid Data for Gender";  
        }  
    }  
}  
  
  
// 0 - Unknown  
// 1 - Male  
// 2 - Female  
  
  
public class Customer  
{  
    public string Name { get; set; }  
    public int Gender { get; set; }  
}

### 34. Enums Example

using System;  
public class Enums  
{  
    public static void Main()  
    {  
        Customer[] customers = new Customer[3];  
        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.Unknown  
        };  
        foreach (Customer customer in customers)  
        {  
            Console.WriteLine("Name = {0} && Gender = {1}", customer.Name, GetGender(customer.Gender));  
        }  
    }  
  
  
    public static string GetGender(Gender gender)  
    {  
        // The swicth here is now more readable and maintainable because   
        // of replacing the integral numbers with Gender enum  
        switch (gender)  
        {  
            case Gender.Unknown:  
                return "Unknown";  
            case Gender.Male:  
                return "Male";  
            case Gender.Female:  
                return "Female";  
            default:  
                return "Invalid Data for Gender";  
        }  
    }  
}  
  
  
public enum Gender  
{  
    Unknown = 0,  
    Male = 1,  
    Female = 2  
}  
  
  
public class Customer  
{  
    public string Name { get; set; }  
    public Gender Gender { get; set; }  
}

### 35. Enums Concepts

If a program uses set of integral numbers, consider replacing them with enums, which makes the program more  
    **Readable**  
    **Maintainable** .   
  
  
**1.** Enums are enumerations.  
**2.** Enums are strongly 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.  
**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 keyowrd (all small letteres) is used to create enumerations, where as Enum class, contains static GetValues() and GetNames() methods which can be used to list Enum underlying type values and Names.

// Default underlying type is int and the value starts at ZERO  
public enum Gender  
{  
    Unknown,  
    Male,  
    Female  
}  
  
  
// Gender enum underlying type is now short and the value starts at ONE  
public enum Gender : short  
{  
    Unknown = 1,  
    Male = 2,  
    Female = 3  
}  
  
  
// Enum values need not be in sequential order. Any valid underlying type value is allowed   
public enum Gender : short  
{  
    Unknown = 10,  
    Male = 22,  
    Female = 35  
}  
  
  
// This enum will not compile, bcos the maximum value allowed for short data type is 32767.   
public enum Gender : short  
{  
    Unknown = 10,  
    Male = 32768,  
    Female = 35  
}   
  
  
**Note:** Use short.MaxValue to find out the maximum value that a short data type can hold  
  
  
**An explicit cast is needed to convert from enum type to an integral type and vice versa.**  
int i = Gender.Male;  
The above line will not compile. A compiler error will be generated stating:  
Cannot implicitly convert type 'Gender' to 'int'. An explicit conversion exists (are you missing a cast?)  
  
  
Gender female = 2;   
The above line will also not compile. A slightly different compiler error will be generated stating  
Cannot implicitly convert type 'int' to 'Gender'. An explicit conversion exists (are you missing a cast?)  
  
**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. An explicit cast is required as shown below.**  
using System;  
public class Enums  
{  
    public static void Main()  
    {  
        // This line will not compile. Cannot implicitly convert type 'Season' to 'Gender'.   
        // An explicit conversion is required.  
        // Gender gender = Season.Winter;  
  
  
        // This line comiples as we have an explicit cast  
        Gender gender = (Gender)Season.Winter;  
    }  
}  
public enum Gender : int  
{  
    Unknown = 1,  
    Male = 2,  
    Female = 3  
}  
public enum Season : int  
{  
    Winter = 1,  
    Spring = 2,  
    Summer = 3  
}  
  
  
enum keyowrd (all small letteres) is used to create enumerations, where as Enum class, contains  static **GetValues**() and **GetNames**() methods which can be used to list Enum underlying type values and Names.  
  
  
**Sample Program listing all enum member values and Names**  
using System;  
public class Enums  
{  
    public static void Main()  
    {  
        int[] Values = (int[])Enum.GetValues(typeof(Gender));  
        Console.WriteLine("Gender Enum Values");  
        foreach (int value in Values)  
        {  
            Console.WriteLine(value);  
        }  
          
        Console.WriteLine();  
        string[] Names = Enum.GetNames(typeof(Gender));  
        Console.WriteLine("Gender Enum Names");  
        foreach (string Name in Names)  
        {  
            Console.WriteLine(Name);  
        }  
    }  
}  
public enum Gender : int  
{  
    Unknown = 1,  
    Male = 2,  
    Female = 3  
}

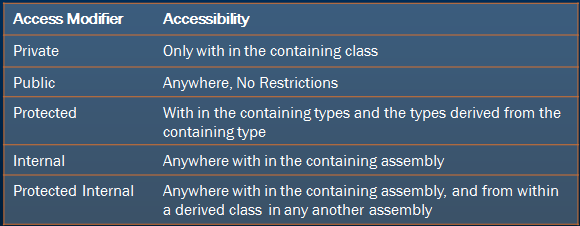
### 36. Types V/S Type members

**1.** Understand the difference between Types and Type Members  
**2.** Understand Organising code with regions  
  
  
In the example below **Customer** is the Type and **private fields**(\_id, \_firstName, \_lastName), **Properties**(Id, FirstName, LastName) and GetFullName() **method** are type members.  
  
public class Customer  
{  
    #region Private Fields  
    private int \_id;  
    private string \_firstName;  
    private string \_lastName;  
    #endregion  
  
    #region Properties  
    public int Id  
    {  
        get { return \_id; }  
        set { \_id = value; }  
    }  
    public string FirstName  
    {  
        get { return \_firstName; }  
        set { \_firstName = value; }  
    }  
    public string LastName  
    {  
        get { return \_lastName; }  
        set { \_lastName = value; }  
    }  
    #endregion  
  
    #region Methods  
    public string GetFullName()  
    {  
        return this.\_firstName + " " + this.\_lastName;  
    }  
    #endregion

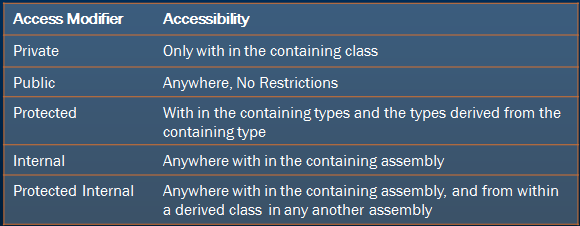
So, in general **classes, structs, enums, interfaces, delegates** are called as **types** and **fields, properties, constructors, methods** etc., that normally reside in a type are called as **type members.**  
  
  
**In C# there are 5 different access modifiers.**  
1. Private  
2. Protected  
3. Internal  
4. Protected Internal  
5. Public  
  
  
**Type members** can have all the access modifiers, where as **types** can have only 2 (internal, public) of the 5 access modifiers  
  
  
*In the next session we will discuss about all the access modifiers in detail.*  
  
  
**Customer** class makes use of regions. Using **regions** you can expand and collapse sections of your code either manually, or using visual studio **Edit** -> **Outlining** -> **Toggle All Outlining**

### 37. Access Modifiers – private, Public and Protected

**There are 5 different access modifiers in c#.**  
**1.** Private  
**2.** Protected  
**3.** Internal  
**4.** Protected Internal  
**5.** Public  
  
  
**Private members** are available **only with in the containing type**, where as public members are available **any where**. There is **no restriction**.

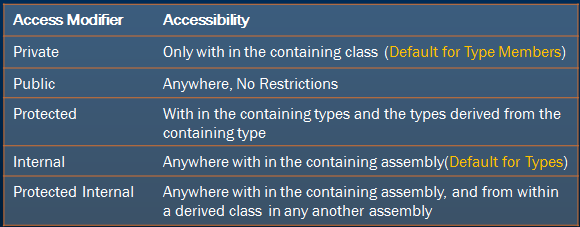
In the example below, **\_id** is private, so this member is only available with in the Customer class (Containing Type). It is a compile time error to access \_id outside of the Customer Class.   
  
The following line in the MainClass will generate a compiler error stating, 'Customer.\_id' is inaccessible due to its protection level.  
CustomerInstance.\_id = 101;  
  
  
On the other hand, since Id is a public member, you can access this member any where, even outside of the Customer class. Infact, we invoke the Id property of the Customer class, in the Main() method as shown below.  
CustomerInstance.Id = 101;  
  
  
**Example for Private and Public Members:**  
public class Customer  
{  
    private int \_id;  
  
  
    public int Id  
    {  
        get  
        {  
            return \_id;  
        }  
        set  
        {  
            \_id = value;  
        }  
    }  
}  
  
  
public class MainClass  
{  
    private static void Main()  
    {  
        Customer CustomerInstance = new Customer();  
        CustomerInstance.Id = 101;  
          
        // Compiler Error : 'Customer.\_id' is inaccessible due to its protection level  
        // CustomerInstance.\_id = 101;  
    }  
}  
  
  
**Protected Members** are available, with in the **containing type** and to the types that **derive from the containing type**. Let me explain with an example.   
  
  
**Protected Access Modifier Example:**  
using System;  
public class Customer  
{  
    protected int ID = 101;  
  
  
    public void PrintID()  
    {  
        //Protected member ID is accessible with in Customer class  
        Console.WriteLine(this.ID);  
    }  
}  
public class CorporateCustomer : Customer  
{  
    public void PrintCustomerID()  
    {  
        CorporateCustomer corporateCustomerInstance = new CorporateCustomer();  
        // Can access the base class protected instance member using the derived class object  
        Console.WriteLine(corporateCustomerInstance.ID);  
        // Can access the base class protected instance member using this or base keyword  
        Console.WriteLine(this.ID);  
        Console.WriteLine(base.ID);  
    }  
}  
public class RetailCustomer  
{  
    public void PrintCustomerID()  
    {  
        RetailCustomer retailCustomerInstance = new RetailCustomer();  
        //RetailCustomer class is not deriving from Customer class, hence it is an error  
        //to access Customer class protected ID member, using the retailCustomerInstance  
        //Console.WriteLine(retailCustomerInstance.ID); //Error  
          
        //Both these below lines also produce the same Error  
        //Console.WriteLine(this.ID); // Error  
        //Console.WriteLine(base.ID); // Error  
    }  
}  
  
  
Customer class defines a protecetd member ID. CorporateCustomer class derives from the Customer class, so protected ID member is accessible in the Customer class (Containg Type) and also from the CorporateCustomer class (Derived Type).   
  
  
With in the PrintID() method in the Customer class, Protected member ID is accessible.  
Console.WriteLine(this.ID);  
  
  
**There are 3 ways to access, the base class protected member in the derived class as shown below.**  
**1.** Using the derived class object.  
Console.WriteLine(corporateCustomerInstance.ID);  
**2.** Using the this keyword  
Console.WriteLine(this.ID);  
**3.** Using the base keyword  
Console.WriteLine(base.ID);  
  
  
On the other hand, RetailCustomer class is not deriving from Customer class, hence it's a compile time error to access Customer class protected ID member.  
  


### 38 Internal and Protected Internal

**Internal:**A member with internal access modifier is available any where with in the containing assembly. It's a compile time error to access, an internal member from outside the containing assembly.  
  
**To understand inernal access modifier, we need 2 assemblies. To generate, the 2 assemblies we need follow these steps.**  
1. Open Solution Explorer (From the Viiew menu, select Solution Explorer)  
2. From the Solution Explorer, right click on the project and select Add -> New Project  
3. In the Add New Project Dialog Box, Select Visual C# from the Installed Templates section, and select Class Library from the center pane, and then enter AssemblyOne for the Name of the project and click OK.  
4. Follow steps 2 & 3, to create AssemblyTwo project.  
5. If you have followed these steps correctly, you should now see three projects in the soultion explorer.  
**Now, if we build the solution we should have 3 assemblies generated. Two dll's and one exe. To locate the physical assembly follow these steps.**  
1. Right click on AssemblyOne project, in solution explorer and select Open Folder in Windows Explorer.  
2. Open bin folder  
3. Now open Debug folder  
4. In the Debug folder you should see AssemblyOne.dll, which is the physical assembly.  
 **Copy and paste the following code in class1.cs file of AssemblyOne project** using System;  
namespace AssemblyOne  
{  
    public class AssemblyOneClassI  
    {  
        internal int ID = 999;  
    }  
    public class AssemblyOneClassII  
    {  
        public void Test()  
        {  
            AssemblyOneClassI instance = new AssemblyOneClassI();  
            // Can access inetrnal member ID, AssemblyOneClassII and AssemblyOneClassI  
            // are present in the same assembly              
            Console.WriteLine(instance.ID);  
        }  
    }  
}  
  
In this example, **AssemblyOneClassI** has an **internal member ID**. We can access ID member from **AssemblyOneClassII**, because this class is also present in the same assembly as **AssemblyOneClassI**.  
  
**Now, Copy and Paste the following code, in Class1.cs of AssemblyTwo project.**using System;  
using AssemblyOne;  
namespace AssemblyTwo  
{  
    public class AssemblyTwoClassI  
    {  
        public void Test()  
        {  
            AssemblyOneClassI instance = new AssemblyOneClassI();  
            //Console.WriteLine(instance.ID);          
        }  
    }  
}  
  
**Note: You will get 3 compiler errors at this point. To solve this we need to add an assembly reference. Follow these steps.**  
1. Expand References folder under AssemblyTwo project, from Solution Explorer.  
2. Right Click on References folder and select Add Reference  
3. From the Add Reference dialog box, select Projects tab  
4. From the list, select AssemblyOne project and click OK.  
At, this point all the compiler errors should have gone.  
  
**Uncomment the following line from Class1.cs file from AssemblyTwo project and rebuild the solution.**  
Console.WriteLine(instance.ID);  
  
Now, you will get a compiler error stating 'AssemblyOne.AssemblyOneClassI' does not contain a definition for 'ID' and no extension method 'ID' accepting a first argument of type 'AssemblyOne.AssemblyOneClassI' could be found (are you missing a using directive or an assembly reference?).   
  
This is because, **AssemblyTwoClassI** is not present in **AssemblyOne** assembly and hence cannot access the **internal ID member** defined in **AssemblyOne** assembly. **This proves that internal members are only accessible with in the same assembly. Code outside of the containing assembly cannot access internal members.**  
  
**Protected Internal:**Protected Internal members 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.  
  
Now, change the **access modifier** from **internal** to **protected internal** for **ID member** in **AssemblyOneClassI** of **class1.cs** file in **AssemblyOne** project.  
internal int ID = 999; to protected internal int ID = 999;  
  
**Finally modify the code in Class1.cs file in AssemblyTwo project as shown below.**using System;  
using AssemblyOne;  
namespace AssemblyTwo  
{  
    // Make AssemblyTwoClassI inherit from AssemblyOneClassI      
    public class AssemblyTwoClassI : AssemblyOneClassI  
    {  
        public void Test()  
        {  
            AssemblyOneClassI instance = new AssemblyOneClassI();  
            // Access the base class member using the base keyword              
            Console.WriteLine(base.ID);  
        }  
    }  
}  
  
So, this shows protected internal **ID** member, defined in **AssemblyOne** is accessible in **AssemblyTwo**.  
  


### 39 Access Modifiers for Types

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

You can use **all the 5** access modifiers with **type members**, but types allows only **internal** and **public** access modifiers. It is a compile time error to use private, protected and protected internal access modifiers with types.   
  
  
**The following code will generate a compiler error stating Elements defined in a namespace cannot be explicitly declared as private, protected, or protected internal**  
using System;  
namespace Pragim  
{  
    //Error: Cannot mark types with private, protected and protected internal access modifiers  
    private class MainClass  
    {  
        public static void Main()  
        {  
            Console.WriteLine("This code will not compile");              
        }  
    }  
}  
  
  
[Add 2 class library projects to the solution with names AssemblyOne and AssemblyTwo. If you want to learn to do this, please check Part 50 - Access Modifiers - Internal and Protected Internal.](http://csharp-video-tutorials.blogspot.com/2012/07/part-50-c-tutorial-internal-and.html)  
  
  
**Copy and paste the following code in Class1.cs file of AssemblyOne project.**  
using System;  
namespace AssemblyOne  
{  
    //Class is marked internal. This class is available only with in AssemblyOne  
    internal class AssemblyOneClass  
    {  
        public void Print()  
        {  
            Console.WriteLine("Hello");  
        }  
    }  
}  
  
  
**Copy and paste the following code in Class1.cs file of AssemblyTwo project.**  
using System;  
using AssemblyOne;  
namespace AssemblyTwo  
{  
    //Class is marked public. This class is available in any assembly  
    public class AssemblyTwoClass  
    {  
        public void Print()  
        {  
            AssemblyOneClass instance = new AssemblyOneClass();  
            instance.Print();  
        }  
    }  
}  
  
  
Add a reference to AssemblyOne project, from AssemblyTwo project. Please check the previous session, to learn about adding project references.  
  
  
**Now build the solution. You will notice the following 4 compiler errors.**  
1. 'AssemblyOne.AssemblyOneClass' is inaccessible due to its protection level  
2. The type 'AssemblyOne.AssemblyOneClass' has no constructors defined  
3. 'AssemblyOne.AssemblyOneClass' is inaccessible due to its protection level   
4. 'AssemblyOne.AssemblyOneClass' does not contain a definition for 'Print' and no extension method 'Print' accepting a first argument of type 'AssemblyOne.AssemblyOneClass' could be found (are you missing a using directive or an assembly reference?)  
  
  
All these errors are in AssemblyTwo project, and are related to AssemblyOne.AssemblyOneClass being inaccessible due to its protection level.   
  
  
Now convert the access modifier of AssemblyOneClass from internal to public and rebuild the solution. Now we get no errors. This shows that internal types are accessible only with in the containing assembly.  
  
  
Now just remove the public access modifier from AssemblyOneClass and rebuild the solution. You now again get the same 4 errors that we got before. This is because, if you don't specify an access modifier for a type, then by default the access modifier will be internal.  
  
So if you don't specify an access modifier, then for Types the default is internal and for type members it is private.  
  


### 40 Attributes

**In this part we will learn**  
1. The purpose of attributes  
2. Using an attribute  
3. Customizing attribute using parameters  
  
  
**Purpose:**Attributes allow you to add declarative information to your programs. This information can then be queried at runtime using reflection.

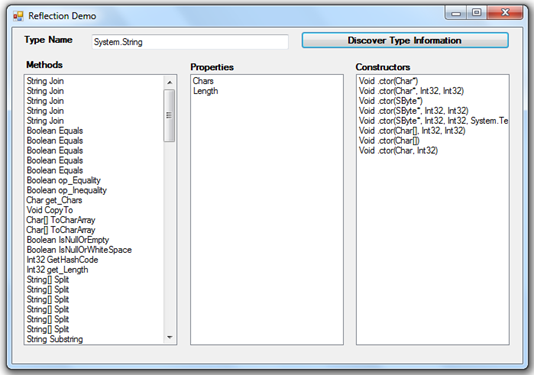
There are several Pre-defined Attributes provided by .NET. It is also possible to create your own Custom Attributes. Creating custom attributes is beyond the scope of this article.  
  
  
**A few pre-defined attributes with in the .NET framework.**  
Obsolete - Marks types and type members outdated  
WebMethod - To expose a method as an XML Web service method  
Serializable - Indicates that a class can be serialized  
  
  
**Example program using pre defined Obsolete attribute:**  
Obsolete attribute can be used with types or type members that are **obsolete (Outdated).** If a developer uses a type or a type member that is decorated with obsolete attribute, the compiler issues a warning or an error depending on how the attribute is configured.  
  
  
In this sample program, **Add(int FirstNumber, int SecondNumber)** method is decorated with [Obsolete] attribute. If you compile this program, in the output window you will see a warning message (Compile complete -- 0 errors, 1 warnings). Also, visual studio, shows a green squiggly line under the  **Add(int FirstNumber, int SecondNumber)** method. If you hover the mouse over the squiggly line, you should see the warning message.  
  
  
**Note**: If you don't see the warning message (Compile complete -- 0 errors, 1 warnings), rebuild the soultion.  
  
  
using System;  
using System.Collections.Generic;  
public class MainClass  
{  
    private static void Main()  
    {  
        Calculator.Add(10, 15);  
    }  
}  
  
  
public class Calculator  
{  
    [Obsolete]  
    public static int Add(int FirstNumber, int SecondNumber)  
    {  
        return FirstNumber + SecondNumber;  
    }  
    public static int Add(List<int> Numbers)  
    {  
        int Sum = 0;  
        foreach (int Number in Numbers)  
        {  
            Sum = Sum + Number;  
        }  
        return Sum;  
    }  
}  
  
  
The warning message says **'Calculator.Add(int, int)' is obsolete**. However, this message is not completely useful, because it says **'Calculator.Add(int, int)' is obsolete**, but not tell us which other method **should we be using instead**. So this is when we can customize, the warning message **using attribute parameters**.  
  
  
The intention of the developer of Calculator class is that, he wanted us to use **Add(List<int> Numbers)**, instead of int **Add(int FirstNumber, int SecondNumber)**. To communicate this message we can customize the warning message using attribute parameters as shown below. With this customization we are not only communicating that **Add(int FirstNumber, int SecondNumber)** method is obsolete, we are also telling to use the alternative method that is available.  
[Obsolete("Use Add(List<int> Numbers) instead")]  
public static int Add(int FirstNumber, int SecondNumber)  
  
  
If you want to generate a compiler error instead of warning, pass true for the bool error parameter of the Obsolete attribute as shown below. Now, we can't even compile the program.  
[Obsolete("Use Add(List<int> Numbers) instead", true)]  
public static int Add(int FirstNumber, int SecondNumber)  
  
  
Finally, If you right click on Obsolete attribute and select Go To Definition, you will see that, an attribute is nothing but a class that inherits from System.Attribute base class.

### 41 Reflection

**Reflection** is the ability of inspecting an assemblie's metadata at runtime.  It is used to find all types in an assembly and/or dynamically invoke methods in an assembly. This includes information about the type, properties, methods, and events of an object. With Reflection, we can dynamically create an instance of a type, bind the type to an existing object, or get the type from an existing object and invoke its methods or access its fields and properties.There are several 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 extensivley used by IDE or a UI designers.  
  
  
**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.  
  
  
**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.  
  
  
**So, in short reflection can be used for type discovery (i.e finding methods, properties, events, fields, constructors etc) and late binding.**  
  
  
In this session we will learn how to list a specifc class methods, properties, fields etc using reflection. All the classes and methods related to reflection are present in System.Reflection namespace.  
  
  
The Type class is the most importanct class.  
  
  
**Consider the Customer class example. This class has got**   
1. Two constructors  
2. Two auto implemeneted properties  
3. Two methods  
  
  
using System;  
using System.Reflection;  
namespace Pragim  
{  
    public class MainClass  
    {  
        private static void Main()  
        {  
            // Get the Type Using GetType() static method  
            Type T = Type.GetType("Pragim.Customer");  
            // Print the Type details  
            Console.WriteLine("Full Name = {0}",T.FullName);  
            Console.WriteLine("Just the Class Name = {0}",T.Name);  
            Console.WriteLine("Just the Namespace = {0}", T.Namespace);  
            Console.WriteLine();  
            // Print the list of Methods  
            Console.WriteLine("Methods in Customer Class");  
            MethodInfo[] methods = T.GetMethods();  
            foreach (MethodInfo method in methods)  
            {  
                // Print the Return type and the name of the method  
                Console.WriteLine(method.ReturnType.Name + " " + method.Name);  
            }  
            Console.WriteLine();  
            //  Print the Properties  
            Console.WriteLine("Properties in Customer Class");  
            PropertyInfo[] properties = T.GetProperties();  
            foreach (PropertyInfo property in properties)  
            {  
                // Print the property type and the name of the property  
                Console.WriteLine(property.PropertyType.Name + " " + property.Name);  
            }  
            Console.WriteLine();  
            //  Print the Constructors  
            Console.WriteLine("Constructors in Customer Class");  
            ConstructorInfo[] constructors = T.GetConstructors();  
            foreach (ConstructorInfo constructor in constructors)  
            {  
                Console.WriteLine(constructor.ToString());  
            }  
        }  
    }  
    public class Customer  
    {  
        public int Id { get; set; }  
        public string Name { get; set; }  
  
  
        public Customer(int ID, string Name)  
        {  
            this.Id = ID;  
            this.Name = Name;  
        }  
  
  
        public Customer()  
        {  
            this.Id = -1;  
            this.Name = string.Empty;  
        }  
  
  
        public void PrintID()  
        {  
            Console.WriteLine("ID = {0}", this.Id);  
        }  
        public void PrintName()  
        {  
            Console.WriteLine("Name = {0}", this.Name);  
        }  
    }  
}  
  
  
In this example to get the type of customer class we have used **GetType()** static method defined on the **Type** class. We pass in the fully qualified name of the type including the namespace as a parameter to the GetType() method.  
Type T = Type.GetType("Pragim.Customer");  
  
  
**To get the type information we have the following 2 ways as well.**  
**Use typeof keyowrd**  
Type T = typeof(Customer);  
  
  
**Use GetType() on the instance of the customer class.**  
Customer C1 = new Customer();  
Type T = C1.GetType();  
  
  
To get the methods information, we use Type.GetMethods(), which returns MethodInfo[] array and along the same lines we use Type.GetProperties() to get properties information, but Type.GetProperties() returns PropertyInfo[] array.

### 42. Reflection Example

  
  
**In the properties window**  
Set the Name of the text box to txtTypeName  
Set the Name of the button to btnDiscoverTypeInformation  
Set the Name of the list boxes, to lstMethods, lstProperties, and lstConstructors  
  
  
Now double click the button control to generate the event handler.  
  
  
**Copy and paste the following code in the button click event handler** (btnDiscoverTypeInformation\_Click).  
  
  
string TypeName = txtTypeName.Text;  
Type T = Type.GetType(TypeName);  
lstMethods.Items.Clear();  
lstProperties.Items.Clear();  
lstConstructors.Items.Clear();  
MethodInfo[] methods = T.GetMethods();  
foreach (MethodInfo method in methods)  
{  
    lstMethods.Items.Add(method.ReturnType.Name + " " + method.Name);  
}  
PropertyInfo[] properties = T.GetProperties();  
foreach (PropertyInfo property in properties)  
{  
    lstProperties.Items.Add(property.PropertyType.Name + " " + property.Name);  
}  
ConstructorInfo[] constructors = T.GetConstructors();  
foreach (ConstructorInfo constructor in constructors)  
{  
    lstConstructors.Items.Add(constructor.ToString());  
}  
  
  
Run the application and enter the type name for which you want to find type information. For example, if you enter **System.Console**, you should see the list of methods, properties and constructors.

### 43. Late Binding using Reflection

1. Early binding and late binding  
2. The difference between the two approaches  
  
  
**Early Binding Example:**  
using System;  
namespace Pragim  
{  
    public class MainClass  
    {  
        private static void Main()  
        {  
            Customer C1 = new Customer();  
            string fullName = C1.GetFullName("Pragim", "Tech");  
            Console.WriteLine("Full Name = {0}", fullName);  
        }  
    }  
    public class Customer  
    {  
        public string GetFullName(string FirstName, string LastName)  
        {  
            return FirstName + " " + LastName;  
        }  
    }  
}

In this example, we have the knowledge of Customer class at compile time. So, we are able to create the instance of the Customer class using the new operator. We are also able to invoke the **GetFullName**() method using **C1**. Intellisense detects the presence of this method and the number and type of parameters that need to be passed in. If you make any mistake in the name of the method, or the number and type of parameters, those mistakes will be immediately raised as compiler errors.  
  
  
**Late Binding Example:**  
using System;  
using System.Reflection;  
namespace Pragim  
{  
    public class MainClass  
    {  
        private static void Main()  
        {  
            // Load the current executing assembly as the Customer class is present in it.  
            Assembly executingAssembly = Assembly.GetExecutingAssembly();  
            // Load the Customer class for which we want to create an instance dynamically  
            Type customerType = executingAssembly.GetType("Pragim.Customer");  
            // Create the instance of the customer type using Activator class   
            object customerInstance = Activator.CreateInstance(customerType);  
            // Get the method information using the customerType and GetMethod()  
            MethodInfo getFullName = customerType.GetMethod("GetFullNames");  
            // Create the parameter array and populate first and last names  
            string[] methodParameters = new string[2];  
            methodParameters[0] = "Pragim"; //FirstName  
            methodParameters[1] = "Tech";     //LastName  
            // Invoke the method passing in customerInstance and parameters array  
            string fullName = (string)getFullName.Invoke(customerInstance, methodParameters);  
            Console.WriteLine("Full Name = {0}", fullName);  
        }  
    }  
    public class Customer  
    {  
        public string GetFullName(string FirstName, string LastName)  
        {  
            return FirstName + " " + LastName;  
        }  
    }  
}  
  
  
**Let's assume we don't have the knowledge of Customer class at compile time, and it will be provided only at run time. In this case we need to bind to the Customer class at runtime.**  
1. Load the assembly which contains the Customer class. In our case, the Customer class is present in the same assembly as the MainClass. So, we use **Assembly.GetExecutingAssembly()** to load the current executing assembly. On the Assembly class, there are several static methods which can be used to load an assembly at runtime dynamically.  
2. Next, we load the Customer class for which we want to create an instance dynamically using **executingAssembly.GetType("Pragim.Customer")**. Make sure you pass in the fully qualified name to the GetType() method, including the namespace. Otherwise you risk getting a NullReferenceException at runtime.  
3. Create the instance of the Customer class using **Activator.CreateInstance(customerType)**.  
4. Once we have the Customer instance, now get the method information which we want to invoke dynamically. we use **customerType.GetMethod("GetFullName").**  
5. The GetFullName() method expects 2 string parameters. So, we need to create a string array,  and populate it with the first and last name parameters.  
6. Finally, invoke the method passing in customerInstance and parameters array.  
  
  
If you mis-spell the method name or if you pass in the wrong number or type of parameters, you wouldn't get a compiler error, but the application crashes at runtime.  
  
  
**Difference between early and late binding:**  
1. Early binding can flag errors at compile time. With late binding there is a risk of run time exceptions.  
2. Early binding is much better for performance and should always be preferred over late binding. Use late binding only when working with onjects that are not available at compile time.

### 44. Generics

**Generics** are introduced in C# 2.0. Generics allow us to **design classes and methods decoupled from the data types**. Generic classes are extensively used by collection classes available in System.Collections.Generic namespace. [Click here to watch the video on generic collection classes.](http://csharp-video-tutorials.blogspot.com/2012/06/what-are-generics.html)  
  
In this example, AreEqual(int value1, int value2) only works with int data type. If, we pass any other data type, we get a compiler error. So, **AreEqual()** method in **Calculator** class is tightly coupled with the int data type, and prevents it from being used with any other data type.

using System;  
namespace Pragim  
{  
    public class MainClass  
    {  
        private 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 value1, int value2)  
        {  
            return value1 == value2;  
        }  
    }  
}  
  
  
**It's a compile time error to invoke AreEqual() method with string parameters.**  
bool Equal = Calculator.AreEqual("A", "B");  
  
  
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, AreEuqal() 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.   
using System;  
namespace Pragim  
{  
    public class MainClass  
    {  
        private static void Main()  
        {  
            bool Equal = Calculator.AreEqual("A", "B");  
            if (Equal)  
            {  
                Console.WriteLine("Equal");  
            }  
            else  
            {  
                Console.WriteLine("Not Equal");  
            }  
        }  
    }  
    public class Calculator  
    {  
        public static bool AreEqual(object value1, object value2)  
        {  
            return value1 == value2;  
        }  
    }  
}  
  
  
**So, the probem with using System.Object type is that**  
1. AreEqual() method is not type safe  
2. Performance degradation due to boxing and unboxing.  
  
  
Both of these issues can be solved with generics and still make AreEqual() method work with different data types. The re written example using generics is shown below.   
using System;  
namespace Pragim  
{  
    public class MainClass  
    {  
        private static void Main()  
        {  
            bool Equal = Calculator.AreEqual<int>(2, 1);  
            if (Equal)  
            {  
                Console.WriteLine("Equal");  
            }  
            else  
            {  
                Console.WriteLine("Not Equal");  
            }  
        }  
    }  
    public class Calculator  
    {  
        public static bool AreEqual<T>(T value1, T value2)  
        {  
            return value1.Equals(value2);  
        }  
    }  
}  
  
  
To make AreEqual() method generic, we specify a type parameter using angular brackets as shown below.  
public static bool AreEqual<T>(T value1, T value2)  
  
  
At the point, When the client code wants to invoke this method, they need to specify the type, they want the method to operate on. If the user wants the AreEqual() method to work with integers, they can invoke the method specifying int as the datatype using angular brackets as shown below.  
bool Equal = Calculator.AreEqual<int>(2, 1);  
  
  
To operate with string data type  
bool Equal = Calculator.AreEqual<string>("A", "B");  
  
  
In this example, we made the method generic. Along the same lines, it is also possible to make classes, interfaces and delegates generic.

### 45. Generic Collections

**Arrays:**

**Adv**:Arrays are strongly types.

**Disadv**: Zero indexBased. (Cannot grow in size)

**.Net1.0 Collections(Ex Arraylist,, Hashtable, stack, queue etc):**

**Adv**: Can grow in size, can Add/remove.

**Disadv**: Not strongly typed,run time errors converting types, Unecessary boxing , low performance.

class Program

{

static void Main(string[] args)

{

ArrayList Numbers = new ArrayList(2);

Numbers.Add(101);

Numbers.Add(102);

Numbers.Add(101);

Numbers.Add(“text”);

foreach (int i in Numbers)

{

Console.WriteLine(i);

}

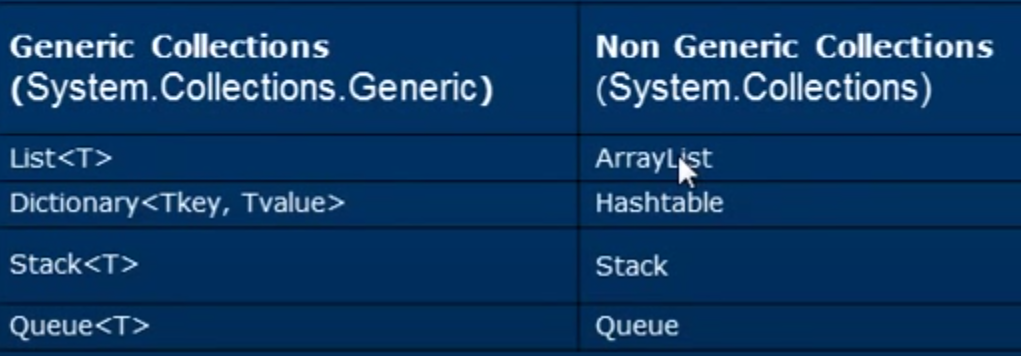
}

}

Hence generics has best of both.

1. Type safe like arrays
2. Can grow automatically in size like arraylists
3. Are convenient to work with using methods like Add, Remove etc.

Generic Vs Non Generic Collections:



class Program

{

static void Main(string[] args)

{

//ArrayList Numbers = new ArrayList(2);

List<int> Numbers = new List<int>(3);

Numbers.Add(101);

Numbers.Add(102);

Numbers.Add(101);

Numbers.Add("text"); // Get a compiler error as the generics are stronlgy typed.. Generics can grow in size just like collections.

foreach (int i in Numbers)

{

Console.WriteLine(i);

}

}

}

### 46. Reasons to Override ToString() Method

Within dot net framework, we know that everytime inherits from system.object

ToString representation is used to give the complex type string representation a meaning rather than using a base class representation which gives the type name.

Complex type (class)

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace ConsoleApplication4

{

using System.Collections;

class Program

{

static void Main(string[] args)

{

int Number = 10;

Console.WriteLine(Number.ToString()); // Converting this number to string representation of an integer

Customer C1 = new Customer();

C1.FirstName = "Test";

C1.LastName = "something";

Console.WriteLine((C1.ToString())); // this acttually prints the name space.classname which is nothing but the type name.

//if some one calls tostring on an object, I want customers last name space his last name, we need to override the toString method.

//can also use a convert class.to string

Console.WriteLine(Convert.ToString(C1));

}

}

public class Customer

{

public string FirstName { get; set; }

public string LastName { get; set; }

//override has 3 methods (equals, gethash and tostring

public override string ToString()

{

// return base.ToString(); // This is the default syntax . The base implementation is nothing but it prints the above implementation which is the namespace

return this.LastName + ", " + this.FirstName; // We can customise 2 string depending on the project to give a meaningful representation

}

}

}

### 47. Reasons to Override Equals() Method

In this session, let's understand the difference between **"=="** operator and **Equals()** method. In C#, every type directly or indirectly inherits from System.Object. So, the **Equals()** virtual method, that has a default implementation is available in every type via inheritance. In this example, variables i and j are integers. So, == and Equals() method returns true, since i and j, both variables have a value of 10.  
  
  
using System;  
namespace Pragim  
{  
    public class MainClass  
    {  
        private static void Main()  
        {  
            int i = 10;  
            int j = 10;  
            Console.WriteLine(i == j);  
            Console.WriteLine(i.Equals(j));  
        }  
    }  
}

Along the same lines, the sample program below, compares 2 enums and both, the == operator and Equals() method returns true, since bothe direction1 and direction2 enums has the same underlying integer value of 1.  
using System;  
namespace Pragim  
{  
    public class MainClass  
    {  
        private static void Main()  
        {  
            Direction direction1 = Direction.East;  
            Direction direction2 = Direction.East;  
  
  
            Console.WriteLine(direction1 == direction2);  
            Console.WriteLine(direction1.Equals(direction2));  
        }  
    }  
    public enum Direction  
    {  
        East = 1,  
        West = 2,  
        North = 3,  
        South = 4  
    }  
}  
  
  
However, if the type is a reference type, then by default **"=="** operator checks for **reference equality** and **.Equals()** method checks for **value equality**. Let's understand what we mean by reference and value equality.  
  
  
In the example below, C1 and C2 are 2 different object reference variables, but they point to the same object. Keep in mind, object reference variables are different from objects. Object reference variables, stay on the stack and are pointers to actual objects on the heap. Since, C1 and C2 both refer to the same object, the reference equality and value equality is true. Value equality means that two objects contain the same values. In this example, the actual object is only one, so obviously the values are also equal. If two objects have reference equality, then they also have value equality, but value equality does not guarantee reference equality.  
using System;  
namespace Pragim  
{  
    public class MainClass  
    {  
        private static void Main()  
        {  
            Customer C1 = new Customer();  
            C1.FirstName = "Simon";  
            C1.LastName = "Tan";  
  
  
            Customer C2 = C1;  
  
  
            Console.WriteLine(C1 == C2);  
            Console.WriteLine(C1.Equals(C2));  
        }  
    }  
    public class Customer  
    {  
        public string FirstName { get; set; }  
        public string LastName { get; set; }  
    }  
}  
  
  
For the example below, == operator returns False. This makes sense because C1 and C2 are referring to different objects. However, .Equals() method returns flase, inspite of the values across C1 and C2 being the same. Hence, it makes sense to override, the Equals() method to return true when the values across the objects are same.  
using System;  
namespace Pragim  
{  
    public class MainClass  
    {  
        private static void Main()  
        {  
            Customer C1 = new Customer();  
            C1.FirstName = "Simon";  
            C1.LastName = "Tan";  
  
  
            Customer C2 = new Customer();  
            C2.FirstName = "Simon";  
            C2.LastName = "Tan";  
  
  
            Console.WriteLine(C1 == C2);  
            Console.WriteLine(C1.Equals(C2));  
        }  
    }  
    public class Customer  
    {  
        public string FirstName { get; set; }  
        public string LastName { get; set; }  
    }  
}  
  
  
The example below overrides, Equals() method. When overriding Equals() method, make sure the passed in object is not null and can be casted to the type we are comparing. When overriding Equals(), you also need to override GetHashCode(), otherwise you get a compiler warning.  
using System;  
namespace Pragim  
{  
    public class MainClass  
    {  
        private static void Main()  
        {  
            Customer C1 = new Customer();  
            C1.FirstName = "Simon";  
            C1.LastName = "Tan";  
  
  
            Customer C2 = new Customer();  
            C2.FirstName = "Simon";  
            C2.LastName = "Tan";  
  
  
            Console.WriteLine(C1 == C2);  
            Console.WriteLine(C1.Equals(C2));  
        }  
    }  
    public class Customer  
    {  
        public string FirstName { get; set; }  
        public string LastName { get; set; }  
  
  
        public override bool Equals(object obj)  
        {  
            // If the passed in object is null  
            if (obj == null)  
            {  
                return false;  
            }  
            if (!(obj is Customer))  
            {  
                return false;  
            }  
            return (this.FirstName == ((Customer)obj).FirstName)  
                && (this.LastName == ((Customer)obj).LastName);  
        }  
  
  
        public override int GetHashCode()  
        {  
            return FirstName.GetHashCode() ^ LastName.GetHashCode();  
        }  
    }  
}

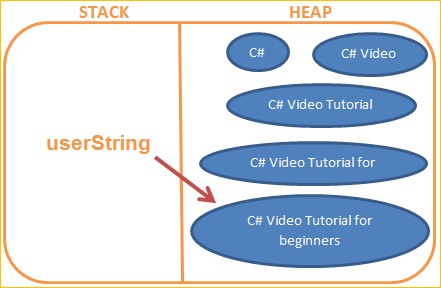
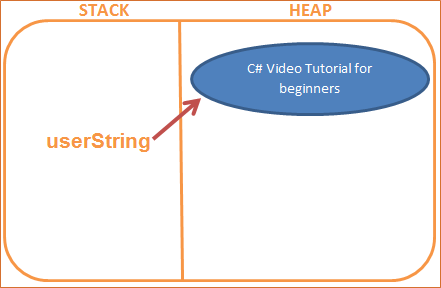
### 48. Difference between Convert.ToString() and ToString()

To understand the difference consider the example below. The **ToString**() method, expects the instance on which you are invoking to be **NOT NULL**. If the object is NULL, you get a NULL Reference exception.  
using System;  
public class MainClass  
{  
    public static void Main()  
    {  
        Customer C1 = null;  
        Console.WriteLine(C1.ToString());  
    }  
}  
public class Customer  
{  
    public string Name { get; set; }  
}

On the other hand, Convert.ToString() returns an empty string if the object is NULL.  
using System;  
public class MainClass  
{  
    public static void Main()  
    {  
        Customer C1 = null;  
        Console.WriteLine(Convert.ToString(C1));  
    }  
}  
public class Customer  
{  
    public string Name { get; set; }  
}  
  
  
**So in summary, Convert.ToString() handles null, while ToString() doesn't, and throws a NULL Reference exception. Depending on the type of the application, architecture and what you are trying to achieve, you choose one over the other.**

### 49. difference between System.String and System.Text.StringBuilder

Strings of type **StringBuilder** are mutable where as strings of type **System.String** are immutable.  As StringBuilder objects are mutable, they offer better performance than string objects of type System.String, when heavy string manipulation is involved.

**Let's understand the meaning of mutable and immutable strings with an example.**  
using System;  
public class MainClass  
{  
    public static void Main()  
    {  
        string userString = "C#";  
        userString += " Video";  
        userString += " Tutorial";  
        userString += " for";  
        userString += " beginners";  
        Console.WriteLine(userString);  
    }  
}  
  
  
**In this example, userString variable is changed 5 times.**  
1. C#  
2. C# => C# Video  
3. C# Video => C# Video Tutorial  
4. C# Video Tutorial => C# Video Tutorial for  
5. C# Video Tutorial for => C# Video Tutorial for beginners  
  
  
Since, userString variable is of type System.String, and when we change this string 5 times, we end up with 5 string objects on the heap as shown in the diagram below. Immutable means, once a string object is created it cannot be changed, without creating another new string object. So in our example, When we initialize userString variable to **"C#"** we get one immutable string object on the heap. When we concatenate **" Video"** word to userString variable, the first created **"C#"** string object is orphaned(userString variable no longer points to this object). Now another new string object with words **"C# Video"** will be created to which the userString variable points to. So this process continues until, userString reference variable, points to the last string object (**C# Video Tutorial for beginners**), leaving the other 4 string onbjects on the heap(orphaned), until they are garbage collected, increasing the pressure on memory.  
  
  
  
  
But on the other hand, StringBuilder string objects are mutable, meaning they can be changed inplace, without the need of creating another new StringBuilder object. The above example is rewritten using StringBuilder object.  
using System;  
using System.Text;  
namespace Pragim  
{  
    public class MainClass  
    {  
        public static void Main()  
        {  
            StringBuilder userStringBuilder =   
                new StringBuilder("C#");  
            userStringBuilder.Append(" Video");  
            userStringBuilder.Append(" Tutorial");  
            userStringBuilder.Append(" for");  
            userStringBuilder.Append(" beginners");  
            Console.WriteLine(userStringBuilder.ToString());  
        }  
    }  
}  
  
  
With StringBuilder, no matter how many times you manipulate a string, you will ever have only one instance.   
  
  
  
  
**So in brief, here are the differences between String and StringBuilderobjects.**  
1. Objects of type StringBuilder are mutable where as objects of type System.String are immutable.   
2. As StringBuilder objects are mutable, they offer better performance than string objects of type System.String.  
3. StringBuilder class is present in System.Text namespace where String class is present in System namespace.  
  
  
Just imagine, the number of orphaned string objects that get created on the heap when you have a program as shown below.  
using System;  
namespace Pragim  
{  
    public class MainClass  
    {  
        public static void Main()  
        {  
            string strNumbers = string.Empty;  
            for (int i = 0; i < 1000; i++)  
            {  
                strNumbers += i.ToString() + " ";  
            }  
            Console.WriteLine(strNumbers);  
        }  
    }  
}

### 50. Partial Class

**.** What are partial classes?  
**2.** What are the advantages of using partial classes?  
**3.** Where are partial classes used?  
  
**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.

**Let's understand partial classes with an example.**Create an asp.net web application project. Add a class file, with name **Customer.cs** to the project. Copy and paste the following code in the **customer.cs** file. This is a very simple customer class, with 2 private fields, 2 public properties and a public method.  
public class Customer  
{  
    private string \_firstName;  
    private string \_lastName;  
  
    public string FirstName  
    {  
        get { return \_firstName; }  
        set { \_firstName = value; }  
    }  
  
    public string LastName  
    {  
        get { return \_lastName; }  
        set { \_lastName = value; }  
    }  
  
    public string GetFullName()  
    {  
        return \_firstName + ", " + \_lastName;  
    }  
}

**Now, let us split this class into 2 files.** One file is going to contain, the private fields and public properties, and the other file is going to contain the public method. Right click on the web application project, and add a class file, with name **PartialCustomerOne.cs**. Notice, that the **PartialCustomer** class is marked with the **partial** keyword and it contains, only, the 2 private fields and the public properties.   
public partial class PartialCustomer  
{  
    private string \_firstName;  
    private string \_lastName;  
  
    public string FirstName  
    {  
        get { return \_firstName; }  
        set { \_firstName = value; }  
    }  
  
    public string LastName  
    {  
        get { return \_lastName; }  
        set { \_lastName = value; }  
    }  
}  
  
**Now, add another class file with name, PartialCustomerTwo.cs**. Notice that, the **PartialCustomer** class, in this file is also marked as a **partial** class, and contains only the public method - **GetFullName()**. We are able to access the private fields, **\_firstName** and **\_lastName**, that are defined in **PartialCustomerOne.cs** file.  
public partial class PartialCustomer  
{  
    public string GetFullName()  
    {  
        return \_firstName + ", " + \_lastName;  
    }  
}  
  
**Copy and paste the following code in the Page\_Load() event of the webform1.** Though, the **PartialCustomer** class is split across 2 files(PartialCustomerOne.cs and PartialCustomerTwo.cs), we are able to use it the same way as the Customer class.  
Customer c1 = new Customer();  
c1.FirstName = "Pragim";  
c1.LastName = "Technologies";  
  
string FullName1 = c1.GetFullName();  
Response.Write("Full Name = " + FullName1 + "<br/>");  
  
PartialCustomer c2 = new PartialCustomer();  
c2.FirstName = "Pragim";  
c2.LastName = "Tech";  
  
string FullName2 = c2.GetFullName();  
Response.Write("Full Name = " + FullName2 + "<br/>");  
  
**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, I haven't really seen anywhere, people using partial classes, just to work on them simultaneously.

### 51. Creating Partial Class

**1.** All the parts spread across different files, must use the **partial keyword**. Otherwise a compiler error is raised.   
**Missing partial modifier. Another partial declaration of this type exists**  
  
**2.** All the parts spread across different files, must have the **same access modifiers**. Otherwise a compiler error is raised.   
**Partial declarations have conflicting accessibility 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. The following code will raise a compiler error stating - **Partial declarations must not specify different base classes.**  
public partial class SamplePartialClass : Employee  
{  
}  
public partial class SamplePartialClass : Customer  
{  
}  
public class Employee  
{  
}  
public class Customer  
{  
}

**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.** In the example below, **SamplePartialClass** needs to provide implementation for both **IEmployee**, and **ICustomer** interface methods.  
public partial class SamplePartialClass : IEmployee  
{  
    public void EmployeeMethod()  
    {  
        //Method Implementation  
    }  
}  
public partial class SamplePartialClass : ICustomer  
{  
    public void CustomerMethod()  
    {  
        //Method Implementation  
    }  
}  
public interface IEmployee  
{  
    void EmployeeMethod();  
}  
public interface ICustomer  
{  
    void CustomerMethod();  
}  
  
**8.** Any **members that are declared in a partial definition** are available to all of the other parts of the partial class.

### 52. Partial Methods

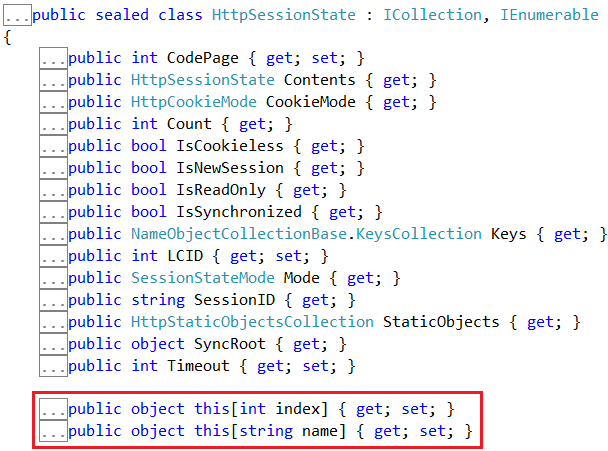
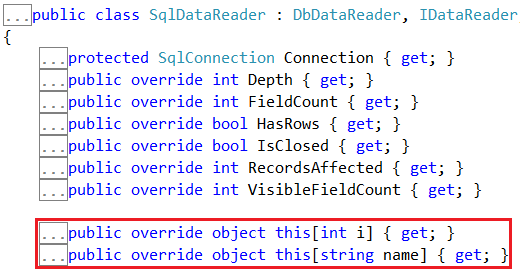
**A partial class or a struct can contain partial methods. A partial method is created using the partial keyword.** Let us understand partial methods with an example. Create a console application. Add a class file, with name **PartialClassFileOne.cs**, to the project. copy and paste the following code.   
  
Notice, that, the **SampleMethod**() definition has the **partial keyword**, and does not have a **body(implementation) only the signature**. 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.   
  
**The implementation can be provided in the same physical file, or in another physical file**, that contains the partial class. In this example, the partial SampleMethod() is invoked in the PublicMethod().  
partial class SampleClass  
{  
    // Declaration of the partial method.  
    partial void SampleMethod();  
  
    // A public method calling the partial method  
    public void PublicMethod()  
    {  
        Console.WriteLine("Public Method Invoked");  
        SampleMethod();  
    }  
}

**Copy and paste the following code in the Main() method of the console application.** When we run the application now, notice that, we don't get a compiler error, in spite of not having an implementation for the partial **SampleMethod**(). Since, the implementation for the partial method is missing, the compiler will remove the signature and all calls to the method.  
SampleClass SC = new SampleClass();  
SC.PublicMethod();  
  
**Now, add a class file**, with name **PartialClassFileTwo.cs**. Copy and paste the following code. The implementation for the partial method is provided here.  
partial class SampleClass  
{  
    // Partial method implemented  
    partial void SampleMethod()  
    {  
        Console.WriteLine("Partial SampleMethod Invoked");  
    }  
}

**Now, run the console application and notice the output.** The partial method and the public method messages are printed on the console.   
  
**A partial method declaration consists of two parts.**  
**1.** The definition (only the method signature ending with a semi-colon, without method body)  
**2.** The implementation.   
**These may be in separate parts of a partial class, or in the same part.**  
  
**Partial methods are private by default**, and it is a compile time error to include any access modifiers, including private. The following code will raise an error stating - A partial method cannot have access modifiers or the virtual, abstract, override, new, sealed, or extern modifiers.  
partial class SampleClass  
{  
    private partial void SampleMethod();  
}  
  
**It is a compile time error, to include declaration and implementation at the same** time for a partial method. Code below produces a compile time error - No defining declaration found for implementing declaration of partial method 'PartialMethodsDemo.SampleClass.SampleMethod()'  
partial class SampleClass  
{  
    partial void SampleMethod()  
    {  
        Console.WriteLine("SampleMethod Implemented");  
    }  
}  
  
**A partial method return type must be void.** Including any other return type is a compile time error - Partial methods must have a void return type  
partial class SampleClass  
{  
    partial int SampleMethod();  
}  
  
**A partial method must be declared within a partial class or partial struct.** A non partial class or struct cannot include partial methods.  
  
**Signature of the partial method declaration**, must match with the signature of the implementation.  
  
**A partial method can be implemented only once**. Trying to implement a partial method more than once, raises a compile time error - A partial method may not have multiple implementing declarations.

### 53. How and where are indexers used in .net

**1.** Where are indexers used in .NET  
**2.** What are indexers in c#

**Where are indexers used in .NET**  
To store or retrieve data from session state or application state variables, we use **indexers**.  
// 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 class, you can see that there is an **integral** and **string indexer** defined. We use "this" keyword to create indexers in c#. We will discuss about creating indexers in our next video session.   
  
  
**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 and select **"Go To Definition"**, to view it's 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.

### 54. Indexers

In this video we will discuss about **creating indexers**.  Let us understand indexers with an example. Create an asp.net web application. Add a class file, with name = **Company.cs**. Copy and paste the following code.

using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Web;  
  
namespace Demo  
{  
    public class Employee  
    {  
        public int EmployeeId { get; set; }  
        public string Name { 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, Name = "Mike", Gender = "Male" });  
            listEmployees.Add(new Employee   
            { EmployeeId = 2, Name = "Pam", Gender = "Female" });  
            listEmployees.Add(new Employee   
            { EmployeeId = 3, Name = "John", Gender = "Male" });  
            listEmployees.Add(new Employee   
            { EmployeeId = 4, Name = "Maxine", Gender = "Female" });  
            listEmployees.Add(new Employee   
            { EmployeeId = 5, Name = "Emiliy", Gender = "Female" });  
            listEmployees.Add(new Employee   
            { EmployeeId = 6, Name = "Scott", Gender = "Male" });  
            listEmployees.Add(new Employee   
            { EmployeeId = 7, Name = "Todd", Gender = "Male" });  
            listEmployees.Add(new Employee   
            { EmployeeId = 8, Name = "Ben", Gender = "Male" });  
        }  
  
        // Use "this" keyword to create an indexer  
        // This indexer takes employeeId as parameter  
        // and returns employee name  
        public string this[int employeeId]  
        {  
            // Just like properties indexers have get and set accessors  
            get  
            {  
                return listEmployees.  
                    FirstOrDefault(x => x.EmployeeId == employeeId).Name;  
            }  
            set  
            {  
                listEmployees.  
                    FirstOrDefault(x => x.EmployeeId == employeeId).Name = value;  
            }  
        }  
    }  
}  
  
**Points to remember:**  
**1.** In the Company class constructor, we are initializing variable **"listEmployees"** and adding employees to the list.  
**2.** We then created an indexer using **"this"** keyword. This indexer takes **employeeId** as parameter and returns **employee name**.  
public string this[int employeeId]  
**3.** Just like properties indexers have **get** and **set** accessors.  
**4.** Indexers can also be overloaded. We will discuss about indexer overloading in our next video.  
  
Now let's discuss about,  **using the indexer**, that we just created. Copy and paste the following code in WebForm1.aspx.cs  
Company company = new Company();  
Response.Write("Name of Employee with Id = 2: " + company[2]);  
Response.Write("<br/>");  
Response.Write("Name of Employee with Id = 5: " + company[5]);  
Response.Write("<br/>");  
Response.Write("Name of Employee with Id = 8: " + company[8]);  
  
Response.Write("<br/>");  
Response.Write("<br/>");  
  
Response.Write("Changing names of employees with Id = 2,5,8");  
Response.Write("<br/>");  
company[2] = "Employee 2 Name Changed";  
company[5] = "Employee 5 Name Changed";  
company[8] = "Employee 8 Name Changed";  
  
Response.Write("Name of Employee with Id = 2: " + company[2]);  
Response.Write("<br/>");  
Response.Write("Name of Employee with Id = 5: " + company[5]);  
Response.Write("<br/>");  
Response.Write("Name of Employee with Id = 8: " + company[8]);  
  
**Points to remember:**  
**1.** EmployeeId's **2,5** and **8** are passed into the company object, to retrieve the respective **employee names**. To retrieve the names of the employees, the **"get"** accessor of the indexer is used.  
**2.** To change the names of employees, we are again using the integral indexer defined on Company class.  
company[2] = "Employee 2 Name Changed";  
  
Notice that, because of the **"employeeId"** indexer, I am able to use company object like an array.

### 55. Overloading Indexers

In [Part 65](http://csharp-video-tutorials.blogspot.com/2013/04/part-65-c-tutorial-indexers-in-c.html), we discussed about creating an indexer based on integer parameter.   
public string this[int employeeId]  
{  
    get  
    {  
        return listEmployees.  
            FirstOrDefault(x => x.EmployeeId == employeeId).Name;  
    }  
    set  
    {  
        listEmployees.  
            FirstOrDefault(x => x.EmployeeId == employeeId).Name = value;  
    }  
}  
  
**Now let us create another indexer based on a string parameter.**  
public string this[string gender]  
{  
    get  
    {  
        // Returns the total count of employees whose gender matches  
        // with the gender that is passed in.  
        return listEmployees.Count(x => x.Gender == gender).ToString();  
    }  
    set  
    {  
        // Changes the gender of all employees whose gender matches  
        // with the gender that is passed in.  
        foreach (Employee employee in listEmployees)  
        {  
            if (employee.Gender == gender)  
            {  
                employee.Gender = value;  
            }  
        }  
    }  
}  
  
Please note that, indexers can be overloaded based on the number and type of parameters.  
  
**Here is the complete code of Company class.**  
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Web;  
using System.Data;  
using System.Data.SqlClient;  
using System.Configuration;  
  
namespace Demo  
{  
    public class Employee  
    {  
        public int EmployeeId { get; set; }  
        public string Name { 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, Name = "Mike", Gender = "Male" });  
            listEmployees.Add(new Employee   
            { EmployeeId = 2, Name = "Pam", Gender = "Female" });  
            listEmployees.Add(new Employee   
            { EmployeeId = 3, Name = "John", Gender = "Male" });  
            listEmployees.Add(new Employee   
            { EmployeeId = 4, Name = "Maxine", Gender = "Female" });  
            listEmployees.Add(new Employee   
            { EmployeeId = 5, Name = "Emiliy", Gender = "Female" });  
            listEmployees.Add(new Employee   
            { EmployeeId = 6, Name = "Scott", Gender = "Male" });  
            listEmployees.Add(new Employee   
            { EmployeeId = 7, Name = "Todd", Gender = "Male" });  
            listEmployees.Add(new Employee   
            { EmployeeId = 8, Name = "Ben", Gender = "Male" });  
        }  
  
        public string this[int employeeId]  
        {  
            get  
            {  
                return listEmployees.  
                    FirstOrDefault(x => x.EmployeeId == employeeId).Name;  
            }  
            set  
            {  
                listEmployees.  
                    FirstOrDefault(x => x.EmployeeId == employeeId).Name = value;  
            }  
        }  
  
        public string this[string gender]  
        {  
            get  
            {  
                return listEmployees.Count(x => x.Gender == gender).ToString();  
            }  
            set  
            {  
                foreach (Employee employee in listEmployees)  
                {  
                    if (employee.Gender == gender)  
                    {  
                        employee.Gender = value;  
                    }  
                }  
            }  
        }  
    }  
}  
  
Notice that the Company class has **2 indexers**. The first indexer has an **integer** (employeeId) parameter and the second indexer has got a **string** (gender) parameter.  
  
To test the string indexer, that we have just created, copy and paste the following code in **Page\_Load**() event of **WebForm1.aspx.cs**  
  
Company company = new Company();  
  
Response.Write("Before changing the Gender of all male employees to Female");  
Response.Write("<br/>");  
  
// Get accessor of string indexer is invoked to return the total  
// count of male employees  
Response.Write("Total Employees with Gender = Male:" + company["Male"]);  
Response.Write("<br/>");  
Response.Write("Total Employees with Gender = Female:" + company["Female"]);  
Response.Write("<br/>");  
Response.Write("<br/>");  
  
// Set accessor of string indexer is invoked to change the gender  
// all "Male" employees to "Female"  
company["Male"] = "Female";  
  
Response.Write("After changing the Gender of all male employees to Female");  
Response.Write("<br/>");  
Response.Write("Total Employees with Gender = Male:" + company["Male"]);  
Response.Write("<br/>");  
Response.Write("Total Employees with Gender = Female:" + company["Female"]);

### 56. Optional parameters

**There are 4 ways that can be used to make method parameters optional.**  
**1.** Use parameter arrays  
**2.** Method overloading  
**3.** Specify parameter defaults  
**4.** Use OptionalAttribute that is present in System.Runtime.InteropServices namespace

**Using parameter arrays, to make method parameters optional:**  
AddNumbers function, allows the user to add 2 or more numbers. firstNumber and secondNumber parameters are mandatory, where as restOfTheNumbers parameter is optional.   
public static void AddNumbers(int firstNumber, int secondNumber,   
    params object[] restOfTheNumbers)  
{  
    int result = firstNumber + secondNumber;  
    foreach (int i in restOfTheNumbers)  
    {  
        result += i;  
    }  
  
    Console.WriteLine("Total = " + result.ToString());  
}  
  
**Please note that, a parameter array must be the last parameter** in a formal parameter list. The following function will not compile.  
public static void AddNumbers(int firstNumber, params object[] restOfTheNumbers,   
    int secondNumber)  
{  
    // Function implementation  
}   
  
If the user wants to add just 2 numbers, then he would invoke the method as shown below.  
AddNumbers(10, 20);  
  
On the other hand, if the user wants to add 5 numbers, then he would invoke the method as shown below.  
AddNumbers(10, 20, 30, 40, 50);  
**or**  
AddNumbers(10, 20, new object[] { 30, 40, 50 });  
  
**In our next video**, we will discuss method overloading, specifying parameter defaults & using OptionalAttribute.

### 57. Making method parameters optional using method overloading

**This method allows us to add any number of integers**  
public static void AddNumbers(int firstNumber, int secondNumber,   
    int[] restOfNumbers)  
{  
    int result = firstNumber + secondNumber;  
    if (restOfNumbers != null)  
    {  
        foreach(int i in restOfNumbers)  
        {  
            result += i;  
        }  
    }  
  
    Console.WriteLine("Sum = " + result);  
}

**If we want to add 5 integers** - 10, 20, 30, 40 and 50. We call the method as shown below.  
AddNumbers(10, 20, new int[]{30, 40, 50});  
  
**At the moment all the 3 parameters are mandatory.** If I want to add just 2 numbers, then I can invoke the method as shown below. Notice that, I am passing null as the argument for the 3rd parameter.  
AddNumbers(10, 20, null);  
  
We can make the 3rd parameter optional by overloading AddNumbers() function as shown below.  
public static void AddNumbers(int firstNumber, int secondNumber)  
{  
    AddNumbers(firstNumber, secondNumber, null);  
}  
  
Now, we have **2 overloaded versions** of AddNumbers() function. If we want to add just 2 numbers, then I can use the overloaded version of AddNumbers() function, that takes 2 parameters as shown below.  
AddNumbers(10, 20);  
  
If I want to add 3 or more numbers, then I can use the overloaded version of AddNumbers() function, that takes 3 parameters as shown below.  
AddNumbers(10, 20, new int[] { 30, 40 });

### 58. Making method parameters optional by specifying parameter defaults

**This method allows us to add any number of integers**  
public static void AddNumbers(int firstNumber, int secondNumber,   
    int[] restOfTheNumbers)  
{  
    int result = firstNumber + secondNumber;  
    foreach (int i in restOfTheNumbers)  
    {  
        result += i;  
    }  
  
    Console.WriteLine("Total = " + result.ToString());  
}

**If we want to add 5 integers** - 10, 20, 30, 40 and 50. We call the method as shown below.  
AddNumbers(10, 20, new int[]{30, 40, 50});  
  
**At the moment all the 3 parameters are mandatory**. If I want to add just 2 numbers, then I can invoke the method as shown below. Notice that, I am passing an empty integer array as the argument for the 3rd parameter.  
AddNumbers(10, 20, new int[]{});  
  
**We can make the 3rd parameter optional by specifying a default value of null** for the 3rd parameter.   
public static void AddNumbers(int firstNumber, int secondNumber,  
    int[] restOfTheNumbers = null)  
{  
    int result = firstNumber + secondNumber;  
  
    // loop thru restOfTheNumbers only if it is not null  
    // otherwise you will get a null reference exception  
    if (restOfTheNumbers != null)  
    {  
        foreach (int i in restOfTheNumbers)  
        {  
            result += i;  
        }  
    }  
    Console.WriteLine("Total = " + result.ToString());  
}  
  
**Since we have specified a default value for the 3rd parameter**, it is optional. So, if we want to add just 2 numbers, we can use the function as shown below.  
AddNumbers(10, 20);  
  
**Optional parameters must appear after all required parameters**  
The following method will not comiple. This is because, we are making parameter "a" optional, but it appears before the required parameters "b" and "c".  
public static void Test(int a = 10, int b, int c)  
{  
    // Do something  
}  
  
**The following method will compile**, as optional parameter "a" is specified after all the required parameters ("b" & "c").  
public static void Test(int b, int c, int a = 10)  
{  
    // Do something  
}  
  
**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 paased as the argument for parameter "a" and "2" is passed as the argument for parameter "b" by default.  
Test(1, 2);  
  
My intention is to pass "2" as the argument for parameter "c". To achieve this we can make use of named parameters, as shown below. Notice that, I have specified the name of the parameter for which value "2" is being passed.  
Test(1, c: 2);

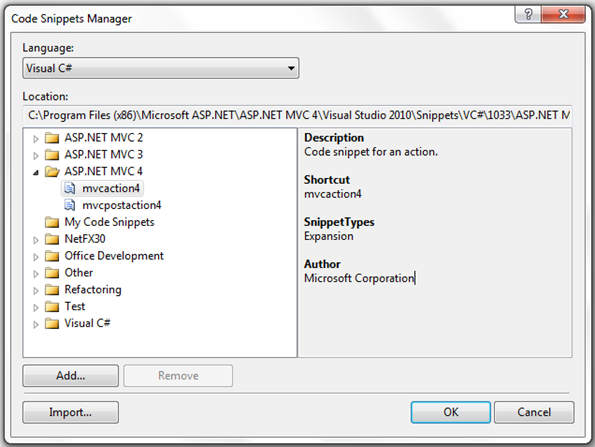
### 59. Making method parameters optional by using OptionalAttribute

**This method allows us to add any number of integers.**  
public static void AddNumbers(int firstNumber, int secondNumber,   
    int[] restOfTheNumbers)  
{  
    int result = firstNumber + secondNumber;  
    foreach (int i in restOfTheNumbers)  
    {  
        result += i;  
    }  
  
    Console.WriteLine("Total = " + result.ToString());  
}

If we want to add 5 integers - 10, 20, 30, 40 and 50. We call the method as shown below.  
**AddNumbers**(10, 20, new int[]{30, 40, 50});  
  
**At the moment all the 3 parameters are mandatory**. If I want to add just 2 numbers, then I can invoke the method as shown below. Notice that, I am passing an empty integer array as the argument for the 3rd parameter.  
AddNumbers(10, 20, new int[]{});  
  
**We can make the 3rd parameter optional** by using OptionalAttribute that is present in System.Runtime.InteropServices namespace. Make sure you have "using" declaration for System.Runtime.InteropServices namespace.  
public static void AddNumbers(int firstNumber, int secondNumber,  
    [Optional] int[] restOfTheNumbers)  
{  
    int result = firstNumber + secondNumber;  
  
    // loop thru restOfTheNumbers only if it is not null  
    // otherwise you will get a null reference exception  
    if (restOfTheNumbers != null)  
    {  
        foreach (int i in restOfTheNumbers)  
        {  
            result += i;  
        }  
    }  
  
    Console.WriteLine("Total = " + result.ToString());  
}  
  
So, if we want to add just 2 numbers, we can now use the function as shown below.  
AddNumbers(10, 20);

### 60. Code snippets in visual studio

**Code snippets are ready-made snippets of code you can quickly insert into your code.** To insert code snippets there are several ways  
**1.** Keyboard shortcut: CTRL K + X  
**2.** Right click and select "Insert Snippet...", from the context menu  
**3.** Click on Edit - Intellisense - 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.  
**SurroundsWith:** These snippets allows the code snippet to be placed around a selected piece of code.  
**Refactoring:** These snippets are used during code refactoring.   
  
**Surround-with code snippets**: These 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.** C#  
**2.** Visual Basic  
**3.** XML  
**4.** HTML  
**5.** JScript  
**6.** SQL  
  
**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.  


### 61 What is 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.

**Here is an example. The code is commented and is self-explanatory.**  
public class Program  
{  
    public static void Main()  
    {  
        // Create a Dictionary, CustomerID is the key. Type is int  
        // Customer object is the value. Type is Customer  
        Dictionary<int, Customer> dictionaryCustomers = new Dictionary<int, Customer>();  
  
        // Create Customer Objects  
        Customer customr1 = new Customer()  
        {  
            ID = 101,  
            Name = "Mark",  
            Salary = 5000  
        };  
  
        Customer customr2 = new Customer()  
        {  
            ID = 102,  
            Name = "Pam",  
            Salary = 7000  
        };  
  
        Customer customr3 = new Customer()  
        {  
            ID = 104,  
            Name = "Rob",  
            Salary = 5500  
        };  
  
        // Add customer objects to the dictionary  
        dictionaryCustomers.Add(customr1.ID, customr1);  
        dictionaryCustomers.Add(customr2.ID, customr2);  
        dictionaryCustomers.Add(customr3.ID, customr3);  
  
        // Retrieve the value (Customer object) from the dictionary,   
        // using key (customer ID). The fastest way to get a value   
        // from the dictionary is by using its key  
        Console.WriteLine("Customer 101 in customer dictionary");  
        Customer customer101 = dictionaryCustomers[101];  
        Console WriteLine("ID = {0}, Name = {1}, Salary = {2}",   
                            customer101.ID, customer101.Name, customer101.Salary);  
        Console.WriteLine("--------------------------------------------------");  
  
        // It is also possible to loop thru each key/value pair in a dictionary  
        Console.WriteLine("All customer keys and values in customer dictionary");  
        foreach (KeyValuePair<int, Customer> customerKeyValuePair in dictionaryCustomers)  
        {  
            Console.WriteLine("Key = " + customerKeyValuePair.Key);  
            Customer cust = customerKeyValuePair.Value;  
            Console.WriteLine("ID = {0}, Name = {1}, Salary = {2}", cust.ID, cust.Name, cust.Salary);  
        }  
        Console.WriteLine("--------------------------------------------------");  
  
        // You can also use implicitly typed variable VAR to   
        // loop thru each key/value pair in a dictionary. But try  
        // to avoid using var, as this makes your code less readable  
        Console.WriteLine("All customer keys and values in customer dictionary");  
        foreach (var customerKeyValuePair in dictionaryCustomers)  
        {  
            Console.WriteLine("Key = " + customerKeyValuePair.Key);  
            Customer cust = customerKeyValuePair.Value;  
            Console.WriteLine("ID = {0}, Name = {1}, Salary = {2}", cust.ID, cust.Name, cust.Salary);  
        }  
        Console.WriteLine("--------------------------------------------------");  
  
        // To get all the keys in the dictionary  
        Console.WriteLine("All Keys in Customer Dictionary");  
        foreach (int key in dictionaryCustomers.Keys)  
        {  
            Console.WriteLine(key);  
        }  
        Console.WriteLine("--------------------------------------------------");  
  
        // To get all the values in the dictionary  
        Console.WriteLine("All Customer objects in Customer Dictionary");  
        foreach (Customer customer in dictionaryCustomers.Values)  
        {  
            Console.WriteLine("ID = {0}, Name = {1}, Salary = {2}", customer.ID, customer.Name, customer.Salary);  
        }  
  
        // If you try to add a key that already exists in the dictionary you   
        // will get an exception - An item with same key has already been   
        // added. So, check if the key already exists  
        if (!dictionaryCustomers.ContainsKey(101))  
        {  
            dictionaryCustomers.Add(101, customr1);  
        }  
  
        // When accessing a dictionary value by key, make sure the dictionary   
        // contains the key, otherwise you will get KeyNotFound exception.  
        if (dictionaryCustomers.ContainsKey(110))  
        {  
            Customer cus = dictionaryCustomers[110];  
        }  
        else  
        {  
            Console.WriteLine("Key does not exist in the dictionary");  
        }  
    }  
}  
  
public class Customer  
{  
    public int ID { get; set; }  
    public string Name { get; set; }  
    public int Salary { get; set; }  
}

we will discuss the following methods of Dictionary class.  
**1.** TryGetValue()  
**2.** Count()  
**3.** Remove()  
**4.** Clear()  
**5.** Using LINQ extension methods with Dictionary  
**6.** Different ways to convert an array into a dictionary

**Code used in the demo:**  
public class Program  
{  
    public static void Main()  
    {  
        // Create Customer Objects  
        Customer customr1 = new Customer()  
        {  
            ID = 101,  
            Name = "Mark",  
            Salary = 5000  
        };  
  
        Customer customr2 = new Customer()  
        {  
            ID = 102,  
            Name = "Pam",  
            Salary = 7000  
        };  
  
        Customer customr3 = new Customer()  
        {  
            ID = 104,  
            Name = "Rob",  
            Salary = 5500  
        };  
  
        // Create a Dictionary, CustomerID is the key. Type is int  
        // Customer object is the value. Type is Customer  
        Dictionary<int, Customer> dictionaryCustomers = new Dictionary<int, Customer>();  
  
        // Add customer objects to the dictionary  
        dictionaryCustomers.Add(customr1.ID, customr1);  
        dictionaryCustomers.Add(customr2.ID, customr2);  
        dictionaryCustomers.Add(customr3.ID, customr3);  
  
        // If you are not sure if a key is present or not, you can use   
        // TryGetValue() method to get the value from a dictionary.  
        Customer customer999;  
        if (dictionaryCustomers.TryGetValue(999, out customer999))  
        {  
            Console.WriteLine("ID = {0}, Name = {1}, Salary = {2}", customer999.ID, customer999.Name, customer999.Salary);  
        }  
        else  
        {  
            Console.WriteLine("Customer with Key = 999 is not found in the dictionary");  
            Console.WriteLine("-------------------------------------------------------------------");  
        }  
  
        // To find the total number of items in a dictionary use Count() method  
        Console.WriteLine("Total items in Dictionary = {0}", dictionaryCustomers.Count());  
        Console.WriteLine("-------------------------------------------------------------------");  
  
        // LINQ extension methods can be used with Dictionary. For example, to find the   
        // total employees whose salary is greater than 5000.  
        Console.WriteLine("Items in dictionary where Salary is greater than 5000 = {0}",  
            dictionaryCustomers.Count(x => x.Value.Salary > 5000));  
        Console.WriteLine("-------------------------------------------------------------------");  
  
        // To remove an item from the dictionary, use Remove() method  
        dictionaryCustomers.Remove(101);  
  
        // To remove all items from the dictionary, use Clear() method  
        dictionaryCustomers.Clear();  
  
        // Create an array of customers  
        Customer[] arrayCustomers = new Customer[3];  
        arrayCustomers[0] = customr1;  
        arrayCustomers[1] = customr2;  
        arrayCustomers[2] = customr3;  
  
        // Convert customer array to a dictionary using ToDictionary() method.  
        // In this example, key is Customer ID and value is the customer object  
        Dictionary<int, Customer> dict = arrayCustomers.ToDictionary(customer => customer.ID, customer => customer);  
        // OR          
        // Dictionary<int, Customr> dict = arrayCustomers.ToDictionary(customer => customer.ID);  
        // OR use a foreach loop  
        // Dictionary<int, Customer> dict = new Dictionary<int, Customer>();  
        // foreach (Customer cust in arrayCustomers)  
        // {  
        //     dict.Add(cust.ID, cust);  
        // }  
  
        // Loop thru the dictionary and print the key/value pairs  
        foreach (KeyValuePair<int, Customer> kvp in dict)  
        {  
            Console.WriteLine("Key = {0}", kvp.Key);  
            Customer customr = kvp.Value;  
            Console WriteLine("ID = {0}, Name = {1}, Salary {2}", customr.ID, customr.Name, customr.Salary);  
        }  
        Console.WriteLine("-------------------------------------------------------------------");  
    }  
}  
  
public class Customer  
{  
    public int ID { get; set; }  
    public string Name { get; set; }  
    public int Salary { get; set; }  
}

### 62. List Collection Classes

List is one of the generic collection classes present in **System.Collections.Generic** namespcae. There are several generic collection classes in System.Collections.Generic namespace as listed below.  
**1.** Dictionary - Discussed in [Parts 72](http://csharp-video-tutorials.blogspot.com/2013/08/part-72-what-is-dictionary-in-c.html) & [73](http://csharp-video-tutorials.blogspot.com/2013/08/part-73-what-is-dictionary-in-c.html)  
**2.** List  
**3.** Stack  
**4.** Queue etc

A List class can be used to create a collection of any type. For example, we can create a list of Integers, Strings and even complex types. The objects stored in the list can be accessed by index. Unlike arrays, lists can grow in size automatically. This class also provides methods to search, sort, and manipulate lists.  
  
public class Program  
{  
    public static void Main()  
    {  
        // Create Customer Objects  
        Customer customer1 = new Customer()  
        {  
            ID = 101,  
            Name = "Mark",  
            Salary = 5000  
        };  
  
        Customer customer2 = new Customer()  
        {  
            ID = 102,  
            Name = "Pam",  
            Salary = 7000  
        };  
  
        Customer customer3 = new Customer()  
        {  
            ID = 104,  
            Name = "Rob",  
            Salary = 5500  
        };  
  
        Customer[] arrayCustomers = new Customer[2];  
        arrayCustomers[0] = customer1;  
        arrayCustomers[1] = customer2;  
        // The following line will throw an exception, Index was outside the bounds of the array.   
        // This is because, arrays does not grow in size automatically.  
        // arrayCustomers[2] = customer3;  
          
        // Create a List of Customers. Here, we have set the size to 2. But when I add a third   
        // element the list size will automatically grow and we will not get an exception.  
        List<Customer> listCustomers = new List<Customer>(2);  
        // To add an element to the list, use Add() method.  
        listCustomers.Add(customer1);  
        listCustomers.Add(customer2);  
        // Adding an element beyond the initial capacity of the list will not throw an exception.  
        listCustomers.Add(customer3);  
  
        // Items can be retrieved from the list by index. The following code will   
        // retrieve the first item from the list. List index is ZERO based.  
        Customer cust = listCustomers[0];  
        Console.WriteLine("ID = {0}, Name = {1}, Salary = {2}",   
                 cust.ID, cust.Name, cust.Salary);  
        Console.WriteLine("------------------------------------------------");  
  
        // foreach or for loop can be used to iterate thru all the items in the list  
        // Using for loop  
        for (int i = 0; i < listCustomers.Count; i++)  
        {  
            Customer customer = listCustomers[i];  
            Console.WriteLine("ID = {0}, Name = {1}, Salary = {2}",   
                     customer.ID, customer.Name, customer.Salary);  
        }  
        Console.WriteLine("------------------------------------------------");  
  
        // Using foreach loop  
        foreach (Customer c in listCustomers)  
        {  
            Console.WriteLine("ID = {0}, Name = {1}, Salary = {2}", c.ID, c.Name, c.Salary);  
        }  
        Console.WriteLine("------------------------------------------------");  
  
        // All generic collection classes including List are strongly typed. This means   
        // if you have created a List of type Customer, only objects of type Customer   
        // can be added to the list. If you try to add an object of different type you would   
        // get a compiler error. The following line will raise a compiler error.  
        // listCustomers.Add("This will not compile");  
  
        // If you want to insert an item at a specific index location of the list, use Insert() method.   
        // The following line will insert customer3 object at index location 1.  
        listCustomers.Insert(1, customer3);  
        Console.WriteLine("ID = {0}, Name = {1}, Salary = {2}",   
               listCustomers[1].ID, listCustomers[1].Name, listCustomers[1].Salary);  
        Console.WriteLine("------------------------------------------------");  
  
        // To get the index of specific item in the list use Indexof() method  
        Console.WriteLine("Index of Customer3 object in the List = " +        
                listCustomers.IndexOf(customer3));  
        Console.WriteLine("------------------------------------------------");  
    }  
}  
  
public class Customer  
{  
    public int ID { get; set; }  
    public string Name { get; set; }  
    public int Salary { get; set; }  
}

In this video, we will discuss  
**1. Contains() function -** Use this function to check if an item exists in the list. This method returns true if the items exists, else false.  
  
**2. Exists() function -** Use this function, to check if an item exists in the list based on a condition. This method returns true if the items exists, else false.  
  
**3. Find() function -** This method searches for an element that matches the conditions defined by the specified lambda expression and returns the first matching item from the list.  
  
**4. FindLast() function -** This method searches for an element that matches the conditions defined by the specified lambda expression and returns the Last matching item from the list.  
  
**5. FindAll() function -** This method returns all the items from the list that match the conditions specified by the lambda expression.

**6. FindIndex() function -** This method returns the index of the first item, that matches the condition specified by the lambda expression. There are 2 other overloads of this method which allows us to specify the range of elements to search, with in the list.  
  
**7. FindLastIndex() function -** This method returns the index of the last item, that matches the condition specified by the lambda expression. There are 2 other overloads of this method which allows us to specify the range of elements to search, with in the list.  
  
**8. Convert an array to a List -** Use ToList() method  
  
**9. Convert a list to an array -** Use ToArray() method  
  
**10. Convert a List to a Dictionary -** Use ToDictionary() method   
  
public class Program  
{  
    public static void Main()  
    {  
        // Create Customer Objects  
        Customer customer1 = new Customer()  
        {  
            ID = 101,  
            Name = "Mark",  
            Salary = 4000  
        };  
  
        Customer customer2 = new Customer()  
        {  
            ID = 102,  
            Name = "Pam",  
            Salary = 7000  
        };  
  
        Customer customer3 = new Customer()  
        {  
            ID = 104,  
            Name = "Rob",  
            Salary = 5500  
        };  
  
        Customer[] arrayCustomers = new Customer[3];  
        arrayCustomers[0] = customer1;  
        arrayCustomers[1] = customer2;  
        arrayCustomers[2] = customer3;  
  
        // To convert an array to a List, use ToList() method  
        List<Customer> listCustomers = arrayCustomers.ToList();  
        foreach (Customer c in listCustomers)  
        {  
            Console.WriteLine("ID = {0}, Name = {1}, Salary = {2}", c.ID, c.Name, c.Salary);  
        }  
        Console.WriteLine("------------------------------------------------------");  
  
        // To convert a List to an array, use ToLArray() method  
        Customer[] arrayAllCustomers = listCustomers.ToArray();  
        foreach (Customer c in arrayAllCustomers)  
        {  
            Console.WriteLine("ID = {0}, Name = {1}, Salary = {2}", c.ID, c.Name, c.Salary);  
        }  
        Console.WriteLine("------------------------------------------------------");  
  
        // To convert a List to a Dictionary use ToDictionary() method  
        Dictionary<int, Customer> dictionaryCustomers = listCustomers.ToDictionary(x => x.ID);  
        foreach (KeyValuePair<int, Customer> keyValuePairCustomers in dictionaryCustomers)  
        {  
            Console.WriteLine("Key = {0}", keyValuePairCustomers.Key);  
            Customer c = keyValuePairCustomers.Value;  
            Console.WriteLine("ID = {0}, Name = {1}, Salary = {2}", c.ID, c.Name, c.Salary);  
        }  
        Console.WriteLine("------------------------------------------------------");  
  
        // To check if an item exists in the list use Contains() function  
        // This method returns true if the items exists, else false  
        if (listCustomers.Contains(customer2))  
        {  
            Console.WriteLine("Customer2 object exists in the list");  
        }  
        else  
        {  
            Console.WriteLine("Customer2 object does not exist in the list");  
        }  
        Console.WriteLine("------------------------------------------------------");  
  
        // To check if an item exists in the list based on a condition, then use Exists() function  
        // This method returns true if the items exists, else false  
        if (listCustomers.Exists(x => x.Name.StartsWith("M")))  
        {  
            Console.WriteLine("List contains customer whose name starts with M");  
        }  
        else  
        {  
            Console.WriteLine("List does not contain a customer whose name starts with M");  
        }  
        Console.WriteLine("------------------------------------------------------");  
  
        // Find() method searches for an element that matches the conditions defined by   
        // the specified lambda expression and returns the first matching item from the list  
        Customer cust = listCustomers.Find(customer => customer.Salary > 5000);  
        Console.WriteLine("ID = {0}, Name = {1}, Salary = {2}", cust.ID, cust.Name, cust.Salary);  
        Console.WriteLine("------------------------------------------------------");  
  
        // FindLast() method searches for an element that matches the conditions defined  
        // by the specified lambda expression and returns the Last matching item from the list  
        Customer lastMatch = listCustomers.FindLast(customer => customer.Salary > 5000);  
        Console WriteLine("ID = {0}, Name = {1}, Salary = {2}", lastMatch.ID, lastMatch.Name, lastMatch.Salary);  
        Console.WriteLine("------------------------------------------------------");  
  
        // FindAll() method returns all the items from the list that  
        // match the conditions specified by the lambda expression  
        List<Customer> filteredCustomers = listCustomers.FindAll(customer => customer.Salary > 5000);  
        foreach (Customer cstmr in filteredCustomers)  
        {  
            Console.WriteLine("ID = {0}, Name = {1}, Salary = {2}", cstmr.ID, cstmr.Name, cstmr.Salary);  
        }  
        Console.WriteLine("------------------------------------------------------");  
  
        // FindIndex() method returns the index of the first item, that matches the   
        // condition specified by the lambda expression. There are 2 other overloads   
        // of this method which allows us to specify the range of elements to   
        // search, with in the list.  
        Console.WriteLine("Index of the first matching customer object whose salary is greater 5000 =" +  
            listCustomers.FindIndex(customer => customer.Salary > 5000));  
        Console.WriteLine("------------------------------------------------------");  
  
        // FindLastIndex() method returns the index of the last item,   
        // that matches the condition specified by the lambda expression.   
        // There are 2 other overloads of this method which allows us to specify   
        // the range of elements to search, with in the list.  
        Console.WriteLine("Index of the Last matching customer object whose salary is greater 5000 = " +  
            listCustomers.FindLastIndex(customer => customer.Salary > 5000));  
        Console.WriteLine("------------------------------------------------------");  
    }  
}  
  
public class Customer  
{  
    public int ID { get; set; }  
    public string Name { get; set; }  
    public int Salary { get; set; }  
}

### 63 Working with generic list class and ranges

**1. AddRange() -** Add() method allows you to add one item at a time to the end of the list, where as AddRange() allows you to add another list of items, to the end of the list.  
  
**2. GetRange() -** Using an item index, we can retrieve only one item at a time from the list, if you want to get a list of items from the list, then use GetRange() function. This function expects 2 parameters, i.e the start index in the list and the number of elements to return.  
  
**3. InsertRange() -** Insert() method allows you to insert a single item into the list at a specificed index, where as InsertRange() allows you, to insert another list of items to your list at the specified index.  
  
**4. RemoveRange() - Remove**() function removes only the first matching item from the list. **RemoveAt**() function, removes the item at the specified index in the list. **RemoveAll**() function removes all the items that matches the specified condition. **RemoveRange**() method removes a range of elements from the list. This function expects 2 parameters, i.e the start index in the list and the number of elements to remove. If you want to remove all the elements from the list without specifying any condition, then use **Clear**() function.

public class Program  
{  
    public static void Main()  
    {  
        // Create Customer Objects  
        Customer customer1 = new Customer()  
        {  
            ID = 101,  
            Name = "Mark",  
            Salary = 4000,  
            Type = "RetailCustomer"  
        };  
  
        Customer customer2 = new Customer()  
        {  
            ID = 102,  
            Name = "Pam",  
            Salary = 7000,  
            Type = "RetailCustomer"  
        };  
  
        Customer customer3 = new Customer()  
        {  
            ID = 103,  
            Name = "Rob",  
            Salary = 5500,  
            Type = "RetailCustomer"  
        };  
  
        Customer customer4 = new Customer()  
        {  
            ID = 104,  
            Name = "John",  
            Salary = 6500,  
            Type = "CorporateCustomer"  
        };  
  
        Customer customer5 = new Customer()  
        {  
            ID = 105,  
            Name = "Sam",  
            Salary = 3500,  
            Type = "CorporateCustomer"  
        };  
  
          
        List<Customer> listCustomers = new List<Customer>();  
        // Add() method allows you to add one at a time to the end of the list  
        listCustomers.Add(customer1);  
        listCustomers.Add(customer2);  
        listCustomers.Add(customer3);  
  
        List<Customer> listCorporateCustomers = new List<Customer>();  
        listCorporateCustomers.Add(customer4);  
        listCorporateCustomers.Add(customer5);  
  
        // AddRange() allows you to add another list of items, to the end of the list  
        listCustomers.AddRange(listCorporateCustomers);  
  
        foreach (Customer customer in listCustomers)  
        {  
            Console.WriteLine("ID = {0}, Name = {1}, Salary = {2}, Type = {3}",  
                customer.ID, customer.Name, customer.Salary, customer.Type);  
        }  
        Console.WriteLine("------------------------------------------------------");  
  
        // GetRange() function returns a list of items from the list.  
        List<Customer> corporateCustomers = listCustomers.GetRange(3, 2);  
        foreach (Customer customer in corporateCustomers)  
        {  
            Console.WriteLine("ID = {0}, Name = {1}, Salary = {2}, Type = {3}",  
                customer.ID, customer.Name, customer.Salary, customer.Type);  
        }  
        Console.WriteLine("------------------------------------------------------");  
  
        // Remove() function removes only the first matching item from the list.  
        listCustomers.Remove(customer1);  
  
        // RemoveAt() function, removes the item at the specified index in the list.  
        listCustomers.RemoveAt(0);  
  
        // RemoveAll() function removes all the items that matches the specified condition.  
        listCustomers.RemoveAll(x => x.Type == "RetailCustomer");  
  
        foreach (Customer customer in listCustomers)  
        {  
            Console.WriteLine("ID = {0}, Name = {1}, Salary = {2}, Type = {3}",  
                customer.ID, customer.Name, customer.Salary, customer.Type);  
        }  
        Console.WriteLine("------------------------------------------------------");  
  
        // RemoveRange() method removes a range of elements from the list.   
        // This function expects 2 parameters, i.e the start index in the   
        // list and the number of elements to remove.  
        listCustomers.RemoveRange(0, 2);  
  
        // Insert() method allows you to insert a single item at a time into   
        // the list at a specificed index  
        listCustomers.Insert(0, customer1);  
        listCustomers.Insert(1, customer2);  
        listCustomers.Insert(2, customer3);  
  
        // InsertRange() allows you, to insert another list of items to your list at the specified index  
        listCustomers.InsertRange(0, listCorporateCustomers);  
  
        foreach (Customer customer in listCustomers)  
        {  
            Console.WriteLine("ID = {0}, Name = {1}, Salary = {2}, Type = {3}",  
                customer.ID, customer.Name, customer.Salary, customer.Type);  
        }  
        Console.WriteLine("------------------------------------------------------");  
  
        // If you want to remove all the elements from the list without specifying   
        // any condition, then use Clear() function.  
        listCustomers.Clear();  
  
        Console.WriteLine(" Total Items in the List = " + listCustomers.Count);  
    }  
}  
  
public class Customer  
{  
    public int ID { get; set; }  
    public string Name { get; set; }  
    public int Salary { get; set; }  
    public string Type { get; set; }  
}

### 64. Sort a list of simple types

Sorting a list of simple types like int, string, char etc, is straight forward. Just invoke **Sort**() method on the list instance and the data will be automatically sorted in ascending order.  
List<int> numbers = new List<int> { 1, 8, 7, 5, 2, 3, 4, 9, 6 };  
numbers.Sort();

If you want the data to be retrieved in descending order, use **Reverse**() method on the list instance.  
numbers.Reverse();  
  
However, when you do the same thing on a complex type like **Customer**, we get a runtime invalid operation exception - Failed to compare 2 elements in the array. This because, .NET runtime does not know, how to sort complex types. We have to tell the way we want data to be sorted in the list by implementing **IComparable** interface. We will discuss this in a later video session.  
  
So, the next obvious question - **How is the sort functionality working for simple types like int, string, char etc?**  
That is because these types (int, string, decimal, char etc) have implemented IComparable interface already.  
  
**Here is the example code used in demo:**  
public class Program  
{  
    public static void Main()  
    {  
        List<int> numbers = new List<int> { 1, 8, 7, 5, 2, 3, 4, 9, 6 };  
  
        Console.WriteLine("Numbers before sorting");  
        foreach (int i in numbers)  
        {  
            Console.WriteLine(i);  
        }  
  
        // Sort() will sort data in ascending order   
        numbers.Sort();  
  
        Console.WriteLine("Numbers after sorting");  
        foreach (int i in numbers)  
        {  
            Console.WriteLine(i);  
        }  
  
        // Use Reverse() method to retrieve data in descending order  
        numbers.Reverse();  
  
        Console.WriteLine("Numbers in descending order");  
        foreach (int i in numbers)  
        {  
            Console.WriteLine(i);  
        }  
  
        List<string> alphabets = new List<string>() { "B", "F", "D", "E", "A", "C" };  
  
        Console.WriteLine("Alphabets before sorting");  
        foreach (string alphabet in alphabets)  
        {  
            Console.WriteLine(alphabet);  
        }  
  
        alphabets.Sort();  
  
        Console.WriteLine("Alphabets after sorting");  
        foreach (string alphabet in alphabets)  
        {  
            Console.WriteLine(alphabet);  
        }  
  
        alphabets.Reverse();  
  
        Console.WriteLine("Alpabets in descending order");  
        foreach (string alphabet in alphabets)  
        {  
            Console.WriteLine(alphabet);  
        }  
  
        Customer customer1 = new Customer()  
        {  
            ID = 101,  
            Name = "Mark",  
            Salary = 4000  
        };  
  
        Customer customer2 = new Customer()  
        {  
            ID = 102,  
            Name = "Pam",  
            Salary = 7000  
        };  
  
        Customer customer3 = new Customer()  
        {  
            ID = 103,  
            Name = "Rob",  
            Salary = 5500  
        };  
  
        List<Customer> listCustomers = new List<Customer>();  
        listCustomers.Add(customer1);  
        listCustomers.Add(customer2);  
        listCustomers.Add(customer3);  
  
        Console.WriteLine("Customers before sorting");  
        foreach (Customer customer in listCustomers)  
        {  
            Console.WriteLine(customer.Name);  
        }  
  
        // Invoking Sort() on list of complex types will   
        // throw invalid operation exception, unless   
        // IComparable interface is implemented  
        listCustomers.Sort();  
              
        Console.WriteLine("Customers after sorting");  
        foreach (Customer customer in listCustomers)  
        {  
            Console.WriteLine(customer.Name);  
        }  
    }  
}  
  
public class Customer   
{  
    public int ID { get; set; }  
    public string Name { get; set; }  
    public int Salary { get; set; }  
  
}

### 65. Sort a list of complex types

To sort a list of complex types without using **LINQ**, the complex type has to implement **IComparable** interface and provide implementation for **CompareTo**() method. CompareTo() method returns an integer, and the meaning of the return value is shown below.  
**Return value greater than ZERO -** The current instance is greater than the object being compared with.  
**Return value less than ZERO -** The current instance is less than the object being compared with.  
**Return value is ZERO -** The current instance is equal to the object being compared with.

Alternatively you can also invoke **CompareTo**() method. Salary property of the Customer object is int. CompareTo() method is already implemented on integer type, so we can invoke this method and return it's value as shown below.  
return this.Salary.CompareTo(obj.Salary);  
  
public class Customer : IComparable<Customer>  
{  
    public int ID { get; set; }  
    public string Name { get; set; }  
    public int Salary { get; set; }  
  
    public int CompareTo(Customer obj)  
    {  
        //if (this.Salary > obj.Salary)  
        //    return 1;  
        //else if (this.Salary < obj.Salary)  
        //    return -1;  
        //else  
        //    return 0;  
  
        // OR, Alternatively you can also invoke CompareTo() method.   
        return this.Salary.CompareTo(obj.Salary);  
    }  
}  
  
If you prefer not to use the **Sort functionality provided by the Customer class**, then you can provide your own by implementing **IComparer** interface. For example, if I want the customers to sorted by name instead of salary.  
**Step 1:** Implement IComparer interface  
public class SortByName : IComparer<Customer>  
{  
    public int Compare(Customer x, Customer y)  
    {  
        return x.Name.CompareTo(y.Name);  
    }  
}  
  
**Step 2:** Pass an instance of the class that implements IComparer interface, as an argument to the Sort() method.  
SortByName sortByName = new SortByName();  
listCutomers.Sort(sortByName);  
  
**Here is the Main() method code:**  
public class Program  
{  
    public static void Main()  
    {  
        Customer customer1 = new Customer()  
        {  
            ID = 101,  
            Name = "Mark",  
            Salary = 4000  
        };  
  
        Customer customer2 = new Customer()  
        {  
            ID = 102,  
            Name = "John",  
            Salary = 7000  
        };  
  
        Customer customer3 = new Customer()  
        {  
            ID = 103,  
            Name = "Ken",  
            Salary = 5500  
        };  
  
        List<Customer> listCutomers = new List<Customer>();  
        listCutomers.Add(customer1);  
        listCutomers.Add(customer2);  
        listCutomers.Add(customer3);  
  
        Console.WriteLine("Customers before sorting");  
        foreach (Customer customer in listCutomers)  
        {  
            Console.WriteLine(customer.Name + " - " + customer.Salary);  
        }  
  
        // Sort() method should sort customers by salary  
        listCutomers.Sort();  
  
        Console.WriteLine("Customers after sorting");  
        foreach (Customer customer in listCutomers)  
        {  
            Console.WriteLine(customer.Name + " - " + customer.Salary);  
        }  
  
        // To sort customers by name instead of salary  
        SortByName sortByName = new SortByName();  
        listCutomers.Sort(sortByName);  
  
        Console.WriteLine("Customers after sorting by Name");  
        foreach (Customer customer in listCutomers)  
        {  
            Console.WriteLine(customer.Name + " - " + customer.Salary);  
        }  
    }  
}

### 66. Sort a list of complex types using Comparison delegate

One of the overloads of the **Sort**() method in List class expects Comparison delegate to be passed as an argument. Let us understand using this overloaded version.  
public void Sort(Comparison<T> comparison)

**Approach 1:**  
**Step 1:** Create a function whose signature matches the signature of System.Comparison delegate. This is the method where we need to write the logic to compare 2 customer objects.  
private static int CompareCustomers(Customer c1, Customer c2)  
{  
    return c1.ID.CompareTo(c2.ID);  
}  
  
**Step 2:** Create an instance of **System.Comparison** delegate, and then pass the name of the function created in Step1 as the argument. So, at this point **"Comparison"** delegate is pointing to our function that contains the logic to compare 2 customer objects.  
Comparison<Customer> customerComparer = new Comparison<Customer>(CompareCustomers);  
  
**Step 3:** Pass the delegate instance as an argument, to Sort() function.  
listCutomers.Sort(customerComparer);  
  
At this point, listCutomers should be sorted using the logic defined in CompareCustomers() function.   
  
**Approach 2:**  
In Approcah1 this is what we have done  
**1.** Created a private function that contains the logic to compare customers  
**2.** Created an instance of Comparison delegate, and then passed the name of the private function to the delegate.  
**3.** Finally passed the delegate instance to the Sort() method.  
  
**Do we really have to follow all these steps. Isn't there any other way?**   
The above code can be simplified using delegate keyword as shown below.  
listCutomers.Sort(delegate(Customer c1, Customer c2)   
                    {   
                        return (c1.ID.CompareTo(c2.ID));   
                    });  
  
**Approach 3:** The code in Approach 2, can be further simplified using lambda expression as shown below.  
listCutomers.Sort((x, y) => x.ID.CompareTo(y.ID));  
  
**Example:**  
public class Program  
{  
    public static void Main()  
    {  
        Customer customer1 = new Customer()  
        {  
            ID = 101,  
            Name = "Mark",  
            Salary = 4000  
        };  
  
        Customer customer2 = new Customer()  
        {  
            ID = 103,  
            Name = "John",  
            Salary = 7000  
        };  
  
        Customer customer3 = new Customer()  
        {  
            ID = 102,  
            Name = "Ken",  
            Salary = 5500  
        };  
  
        List<Customer> listCutomers = new List<Customer>();  
        listCutomers.Add(customer1);  
        listCutomers.Add(customer2);  
        listCutomers.Add(customer3);  
  
        Console.WriteLine("Customers before sorting");  
        foreach (Customer customer in listCutomers)  
        {  
            Console.WriteLine(customer.ID);  
        }  
  
        // Approach 1  
        // Step 2: Create an instance of Comparison delegate  
        //Comparison<Customer> customerComparer =   
        //    new Comparison<Customer>(CompareCustomers);  
  
        // Step 3: Pass the delegate instance to the Sort method  
        //listCutomers.Sort(customerComparer);  
  
        // Approach 2: Using delegate keyword  
        //listCutomers.Sort(delegate(Customer c1, Customer c2)  
        //{  
        //    return (c1.ID.CompareTo(c2.ID));  
        //});  
              
        // Aaproach 3: Using lambda expression  
        listCutomers.Sort((x, y) => x.ID.CompareTo(y.ID));  
  
        Console.WriteLine("Customers after sorting by ID");  
        foreach (Customer customer in listCutomers)  
        {  
            Console.WriteLine(customer.ID);  
        }  
  
        listCutomers.Reverse();  
        Console.WriteLine("Customers in descending order of ID");  
        foreach (Customer customer in listCutomers)  
        {  
            Console.WriteLine(customer.ID);  
        }  
    }  
  
    // Approach 1 - Step 1  
    // Method that contains the logic to compare customers  
    private static int CompareCustomers(Customer c1, Customer c2)  
    {  
        return c1.ID.CompareTo(c2.ID);  
    }  
}  
  
public class Customer  
{  
    public int ID { get; set; }  
    public string Name { get; set; }  
    public int Salary { get; set; }  
}

### 67. Some useful methods of List collection class

**In this video, we will discuss the following methods**  
**1. TrueForAll() -** Returns true or false depending on whether if every element in the list matches the conditions defined by the specified predicate.  
  
**2. AsReadOnly() -** Returns a read-only wrapper for the current collection. Use this method, if you don't want the client code to modify the collection i.e add or remove any elements from the collection. The ReadOnlyCollection will not have methods to add or remove items from the collection. You can only read items from this collection.  
  
**3. TrimExcess() -** Sets the capacity to the actual number of elements in the List, if that number is less than a threshold value.

**According to MSDN:**  
This method can be used to minimize a collection's memory overhead if no new elements will be added to the collection. The cost of reallocating and copying a large List<T> can be considerable, however, so the TrimExcess method does nothing if the list is at more than 90 percent of capacity. This avoids incurring a large reallocation cost for a relatively small gain. The current threshold is 90 percent, but this could change in the future.  
  
public class Program  
{  
    public static void Main()  
    {  
        Customer customer1 = new Customer()  
        {  
            ID = 101,  
            Name = "Mark",  
            Salary = 5200  
        };  
  
        Customer customer2 = new Customer()  
        {  
            ID = 103,  
            Name = "John",  
            Salary = 7000  
        };  
  
        Customer customer3 = new Customer()  
        {  
            ID = 102,  
            Name = "Ken",  
            Salary = 5500  
        };  
  
        List<Customer> listCutomers = new List<Customer>(100);  
        listCutomers.Add(customer1);  
        listCutomers.Add(customer2);  
        listCutomers.Add(customer3);  
  
        Console.WriteLine("Are all salaries greater than 5000: "   
            + listCutomers.TrueForAll(x => x.Salary > 5000));  
  
        // ReadOnlyCollection will not have Add() or Remove() methods  
        System.Collections.ObjectModel.ReadOnlyCollection<Customer>   
            readOnlyCustomers = listCutomers.AsReadOnly();  
  
        Console.WriteLine("Total Items in ReadOnlyCollection = " +  
            readOnlyCustomers.Count);  
  
        // listCutomers list is created with an initial capacity of 100  
        // but only 3 items are in the list. The filled percentage is   
        // less than 90 percent threshold.  
        Console.WriteLine("List capacity before invoking TrimExcess = " +  
                listCutomers.Capacity);  
        // Invoke TrimExcess() to set the capacity to the actual   
        // number of elements in the List  
        listCutomers.TrimExcess();  
        Console.WriteLine(listCutomers.Capacity);  
    }  
}  
  
public class Customer  
{  
    public int ID { get; set; }  
    public string Name { get; set; }  
    public int Salary { get; set; }  
}

### 68. When to use a dictionary over list

**Find()** method of the List class loops thru each object in the list until a match is found. So, if you want to lookup a value using a key, dictionary is better for performance over list. So, use dictionary when you know the collection will be primarily used for lookups.

**Example code used in the demo**  
public class Program  
{  
    public static void Main()  
    {  
        Country country1 = new Country()  
        {  
            Code = "AUS",  
            Name = "AUSTRALIA",  
            Capital = "Canberra"  
        };  
  
        Country country2 = new Country()  
        {  
            Code = "IND",  
            Name = "INDIA ",  
            Capital = "New Delhi"  
        };  
  
        Country country3 = new Country()  
        {  
            Code = "USA",  
            Name = "UNITED STATES",  
            Capital = "Washington D.C."  
        };  
  
        Country country4 = new Country()  
        {  
            Code = "GBR",  
            Name = "UNITED KINGDOM",  
            Capital = "London"  
        };  
  
        Country country5 = new Country()  
        {  
            Code = "CAN",  
            Name = "CANADA",  
            Capital = "Ottawa"  
        };  
  
        //List<Country> listCountries = new List<Country>();  
        //listCountries.Add(country1);  
        //listCountries.Add(country2);  
        //listCountries.Add(country3);  
        //listCountries.Add(country4);  
        //listCountries.Add(country5);  
  
        Dictionary<string, Country> dictionaryCountries = new Dictionary<string, Country>();  
        dictionaryCountries.Add(country1.Code, country1);  
        dictionaryCountries.Add(country2.Code, country2);  
        dictionaryCountries.Add(country3.Code, country3);  
        dictionaryCountries.Add(country4.Code, country4);  
        dictionaryCountries.Add(country5.Code, country5);  
  
        string strUserChoice = string.Empty;  
        do  
        {  
            Console.WriteLine("Please enter country code");  
            string strCountryCode = Console.ReadLine().ToUpper();  
  
            // Find() method of the list class loops thru each object in the list until a match   
            // is found. So, if you want to lookup a value using a key dictionary is better   
            // for performance over list.   
            // Country resultCountry = listCountries.  
            //                                    Find(country => country.Code == strCountryCode);  
  
            Country resultCountry = dictionaryCountries.ContainsKey(strCountryCode) ?                 dictionaryCountries[strCountryCode] : null;  
  
            if (resultCountry == null)  
            {  
                Console.WriteLine("The country code you enetered does not exist");  
            }  
            else  
            {  
                Console.WriteLine("Name = " + resultCountry.Name + " Captial =" + resultCountry.Capital);  
            }  
  
            do  
            {  
                Console.WriteLine("Do you want to continue - YES or NO?");  
                strUserChoice = Console.ReadLine().ToUpper();  
            }   
            while (strUserChoice != "NO" && strUserChoice != "YES");  
        }  
        while (strUserChoice == "YES");  
    }  
}  
  
public class Country  
{  
    public string Name { get; set; }  
    public string Code { get; set; }  
    public string Capital { get; set; }  
}

### 69. Generic queue collection class

**Queue is a generic FIFO (First In First Out) collection class** that is present in System.Collections.Generic namespace. The Queue collection class is analogous to a queue at the ATM machine to withdraw money. The order in which people queue up, will be the order in which they will be able to get out of the queue and withdraw money from the ATM. The Queue collection class operates in a similar fashion. The first item to be added (enqueued) to the queue, will be the first item to be removed (dequeued) from the Queue.

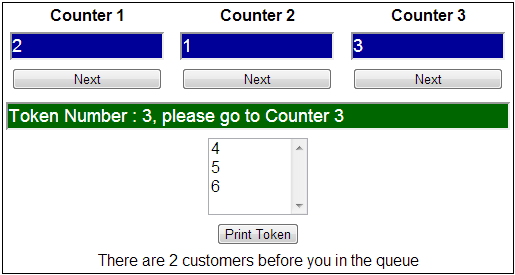
To add items to the end of the queue, use **Enqueue**() method.  
  
To remove an item that is present at the beginning of the queue, use **Dequeue**() method.  
  
A foreach loop iterates thru the items in the queue, but will not remove them from the queue.  
  
To check if an item, exists in the queue, use **Contains**() method.  
  
**What is the difference between Dequeue() and Peek() methods?**  
**Dequeue**() method removes and returns the item at the beginning of the queue, where as **Peek**() returns the item at the beginning of the queue, without removing it.  
  
**Let us undestand the Queue collection class with an example.**  
public class Program  
{  
    public static void Main()  
    {  
        Customer customer1 = new Customer()  
        {  
            ID = 101,  
            Name = "Mark",  
            Gender = "Male"  
        };  
  
        Customer customer2 = new Customer()  
        {  
            ID = 102,  
            Name = "Pam",  
            Gender = "Female"  
        };  
  
        Customer customer3 = new Customer()  
        {  
            ID = 103,  
            Name = "John",  
            Gender = "Male"  
        };  
  
        Customer customer4 = new Customer()  
        {  
            ID = 104,  
            Name = "Ken",  
            Gender = "Male"  
        };  
  
        Customer customer5 = new Customer()  
        {  
            ID = 105,  
            Name = "Valarie",  
            Gender = "Female"  
        };  
  
        // Create a Queue  
        Queue<Customer> queueCustomers = new Queue<Customer>();  
        // To add an item to the queue, use Enqueue() method.  
        // customer1 is added first, so this customer, will be the first to get out of the queue  
        queueCustomers.Enqueue(customer1);  
        // customer2 will be queued up next, so customer2 will be second to get out of the queue  
        queueCustomers.Enqueue(customer2);  
        queueCustomers.Enqueue(customer3);  
        queueCustomers.Enqueue(customer4);  
        queueCustomers.Enqueue(customer5);  
  
        // To retrieve an item from the queue, use Dequeue() method. Notice that the   
        // items are dequeued in the same order in which they were enqueued.  
        // Dequeue() method removes and returns the item at the beginning of the Queue.  
        Customer c1 = queueCustomers.Dequeue();  
        Console.WriteLine(c1.ID + " -  " + c1.Name);  
        Console.WriteLine("Items left in the Queue = " + queueCustomers.Count);  
  
        Customer c2 = queueCustomers.Dequeue();  
        Console.WriteLine(c2.ID + " -  " + c2.Name);  
        Console.WriteLine("Items left in the Queue = " + queueCustomers.Count);  
  
        Customer c3 = queueCustomers.Dequeue();  
        Console.WriteLine(c3.ID + " -  " + c3.Name);  
        Console.WriteLine("Items left in the Queue = " + queueCustomers.Count);  
  
        Customer c4 = queueCustomers.Dequeue();  
        Console.WriteLine(c4.ID + " -  " + c4.Name);  
        Console.WriteLine("Items left in the Queue = " + queueCustomers.Count);  
  
        Customer c5 = queueCustomers.Dequeue();  
        Console.WriteLine(c5.ID + " -  " + c5.Name);  
        Console.WriteLine("Items left in the Queue = " + queueCustomers.Count);  
        Console.WriteLine("-----------------------------------------------------------");  
  
        // After customer5 is dequeued, there will be no items left in the   
        // queue. So, let's enqueue the five objects once again  
        queueCustomers.Enqueue(customer1);  
        queueCustomers.Enqueue(customer2);  
        queueCustomers.Enqueue(customer3);  
        queueCustomers.Enqueue(customer4);  
        queueCustomers.Enqueue(customer5);  
  
        // If you need to loop thru items in the queue, foreach loop can be used in the   
        // same way as we use it with other collection classes. The foreach loop will   
        // only iterate thru items in the queue, but will not dequeue them.  
        foreach (Customer customer in queueCustomers)  
        {  
            Console.WriteLine(customer.ID + " -  " + customer.Name);  
            Console.WriteLine("Items left in the Queue = " + queueCustomers.Count);  
        }  
        Console.WriteLine("-----------------------------------------------------------");  
  
        // To retrieve an item that is present at the beginning of the   
        // queue, without removing it use Peek() method.  
        Customer c = queueCustomers.Peek();  
        Console.WriteLine(c.ID + " -  " + c.Name);  
        Console.WriteLine("Items left in the Queue = " + queueCustomers.Count);  
        Console.WriteLine("-----------------------------------------------------------");  
  
        // To check if an item, exists in the queue, use Contains() method.  
        if (queueCustomers.Contains(customer1))  
        {  
            Console.WriteLine("customer1 is in Queue");  
        }  
        else  
        {  
            Console.WriteLine("customer1 is not in Queue");  
        }  
    }  
}  
  
public class Customer  
{  
    public int ID { get; set; }  
    public string Name { get; set; }  
    public string Gender { get; set; }  
}

### 70. Generic stack collection class

**Stack is a generic LIFO (Last In First Out) collection class** that is present in System.Collections.Generic namespace. The Stack collection class is analogous to a stack of plates. If you want to add a new plate to the stack of plates, you place it on top of all the already existing plates.If you want to remove a plate from the stack, you will first remove the one that you have last added. The stack collection class also operates in a similar fashion. The last item to be added (pushed) to the stack, will be the first item to be removed (popped) from the stack.

To insert an item at the top of the stack, use **Push**() method.  
  
To remove and return the item that is present at the top of the stack, use **Pop**() method.  
  
A foreach loop iterates thru the items in the stack, but will not remove them from the stack. The items from the stack are retrieved in **LIFO** (Last In First Out), order. The last element added to the Stack is the first one to be removed.  
  
To check if an item exists in the stack, use **Contains**() method.  
  
**What is the difference between Pop() and Peek() methods?**  
Pop() method removes and returns the item at the top of the stack, where as Peek() returns the item at the top of the stack, without removing it.  
  
Let us undestand the Stack collection class with an example.  
public class Program  
{  
    public static void Main()  
    {  
        Customer customer1 = new Customer()  
        {  
            ID = 101,  
            Name = "Mark",  
            Gender = "Male"  
        };  
  
        Customer customer2 = new Customer()  
        {  
            ID = 102,  
            Name = "Pam",  
            Gender = "Female"  
        };  
  
        Customer customer3 = new Customer()  
        {  
            ID = 103,  
            Name = "John",  
            Gender = "Male"  
        };  
  
        Customer customer4 = new Customer()  
        {  
            ID = 104,  
            Name = "Ken",  
            Gender = "Male"  
        };  
  
        Customer customer5 = new Customer()  
        {  
            ID = 105,  
            Name = "Valarie",  
            Gender = "Female"  
        };  
  
        // Create a Stack  
        Stack<Customer> stackCustomers = new Stack<Customer>();  
        // To add an item to the stack, use Push() method.  
        // customer1 is inserted at the top of the stack  
        stackCustomers.Push(customer1);  
        // customer2 will be inserted on top of customer1 and now is on top of the stack  
        stackCustomers.Push(customer2);  
        // customer3 will be inserted on top of customer2 and now is on top of the stack  
        stackCustomers.Push(customer3);  
        stackCustomers.Push(customer4);  
        stackCustomers.Push(customer5);  
  
        // To retrieve an item from the stack, use Pop() method. This method   
        // removes and returns an object at the top of the stack. Since customer5  
        // object is the one that is pushed onto the stack last, this object will be  
        // first to be removed and returned from the stack by the Pop() method  
        Customer c1 = stackCustomers.Pop();  
        Console.WriteLine(c1.ID + " -  " + c1.Name);  
        Console.WriteLine("Items left in the Stack = " + stackCustomers.Count);  
  
        Customer c2 = stackCustomers.Pop();  
        Console.WriteLine(c2.ID + " -  " + c2.Name);  
        Console.WriteLine("Items left in the Stack = " + stackCustomers.Count);  
  
        Customer c3 = stackCustomers.Pop();  
        Console.WriteLine(c3.ID + " -  " + c3.Name);  
        Console.WriteLine("Items left in the Stack = " + stackCustomers.Count);  
  
        Customer c4 = stackCustomers.Pop();  
        Console.WriteLine(c4.ID + " -  " + c4.Name);  
        Console.WriteLine("Items left in the Stack = " + stackCustomers.Count);  
  
        Customer c5 = stackCustomers.Pop();  
        Console.WriteLine(c5.ID + " -  " + c5.Name);  
        Console.WriteLine("Items left in the Stack = " + stackCustomers.Count);  
        Console.WriteLine("-----------------------------------------------------------");  
  
        // After customer5 is removed, there will be no items left in the   
        // stack. So, let's push the five objects once again  
        stackCustomers.Push(customer1);  
        stackCustomers.Push(customer2);  
        stackCustomers.Push(customer3);  
        stackCustomers.Push(customer4);  
        stackCustomers.Push(customer5);  
  
        // If you need to loop thru items in the stack, foreach loop can be used in the same   
        // way as we use it with other collection classes. The foreach loop will only iterate   
        // thru items in the stack, but will not remove them. Notice that the items from the   
        // stack are retrieved in LIFO (Last In First Out), order. The last element added to   
        // the Stack is the first one to be removed.  
        foreach (Customer customer in stackCustomers)  
        {  
            Console.WriteLine(customer.ID + " -  " + customer.Name);  
            Console.WriteLine("Items left in the Stack = " + stackCustomers.Count);  
        }  
        Console.WriteLine("-----------------------------------------------------------");  
  
        // To retrieve an item that is present at the top of the   
        // stack, without removing it use Peek() method.  
        Customer c = stackCustomers.Peek();  
        Console.WriteLine(c.ID + " -  " + c.Name);  
        Console.WriteLine("Items left in the Stack = " + stackCustomers.Count);  
        Console.WriteLine("-----------------------------------------------------------");  
  
        Customer c99 = stackCustomers.Peek();  
        Console.WriteLine(c99.ID + " -  " + c99.Name);  
        Console.WriteLine("Items left in the Stack = " + stackCustomers.Count);  
        Console.WriteLine("-----------------------------------------------------------");  
  
        // To check if an item, exists in the stack, use Contains() method.  
        if (stackCustomers.Contains(customer1))  
        {  
            Console.WriteLine("customer1 is in stack");  
        }  
        else  
        {  
            Console.WriteLine("customer1 is not in stack");  
        }  
    }  
}  
  
public class Customer  
{  
    public int ID { get; set; }  
    public string Name { get; set; }  
    public string Gender { get; set; }  
}

### 71. Real time example of queue collection class

Here is one simple **real-time working example where a Queue class can be used**.  
  
When you walk into a bank or a passport office, you will collect a token and wait in the queue for your token number to be called. From the application perspective, when a token is issued, the token number will be added to the end of the Queue. When a representative at the counter is available to server a customer, he will push the "Next" button and the token number that is present at the beginning of the queue, will be dequeued. So, this is one example, where a Queue collection class can be effectively used.   


**WebForm1.aspx**  
<table style="border:1px solid black; font-family:Arial; text-align:center">  
    <tr>  
        <td>  
            <b>Counter 1</b>  
        </td>  
        <td>  
            <b>Counter 2</b>  
        </td>  
        <td>  
            <b>Counter 3</b>  
        </td>  
    </tr>  
    <tr>  
        <td>  
            <asp:TextBox ID="txtCounter1" Width="150px" Font-Size="Large" runat="server"  
                            BackColor="#000099" ForeColor="White">  
            </asp:TextBox>  
        </td>  
        <td>  
            <asp:TextBox ID="txtCounter2" Width="150px" Font-Size="Large" runat="server"  
                            BackColor="#000099" ForeColor="White">  
            </asp:TextBox>  
        </td>  
        <td>  
            <asp:TextBox ID="txtCounter3" Width="150px" Font-Size="Large" runat="server"  
                            BackColor="#000099" ForeColor="White">  
            </asp:TextBox>  
        </td>  
    </tr>  
    <tr>  
        <td>  
            <asp:Button ID="btnCounter1" Width="150px" runat="server" Text="Next"   
                onclick="btnCounter1\_Click" />  
        </td>  
        <td>  
            <asp:Button ID="btnCounter2" Width="150px" runat="server" Text="Next"   
                onclick="btnCounter2\_Click" />  
        </td>  
        <td>  
            <asp:Button ID="btnCounter3" Width="150px" runat="server" Text="Next"   
                onclick="btnCounter3\_Click" />  
        </td>  
    </tr>  
    <tr>  
        <td colspan="3">  
            <asp:TextBox ID="txtNextToken" Font-Size="Large" Width="500px" runat="server"  
                        BackColor="#003300" ForeColor="White">  
            </asp:TextBox>  
        </td>  
    </tr>  
    <tr>  
        <td colspan="3">  
            <asp:ListBox ID="listTokens" Width="100px" Font-Size="Medium" runat="server">  
            </asp:ListBox>  
        </td>  
    </tr>  
    <tr>  
        <td colspan="3">  
            <asp:Button ID="btnPrintToken" runat="server" Text="Print Token"   
                onclick="btnPrintToken\_Click" />  
        </td>  
    </tr>  
    <tr>  
        <td colspan="3">  
            <asp:Label ID="lblCurrentStatus" runat="server" Font-Size="Medium">  
            </asp:Label>  
        </td>  
    </tr>  
</table>  
  
**WebForm1.aspx.cs**  
public partial class WebForm1 : System.Web.UI.Page  
{  
    protected void Page\_Load(object sender, EventArgs e)  
    {  
        if (Session["TokenQueue"] == null)  
        {  
            Queue<int> tokenQueue = new Queue<int>();  
            Session["TokenQueue"] = tokenQueue;  
        }  
    }  
  
    protected void btnCounter1\_Click(object sender, EventArgs e)  
    {  
        ServerNextCustomer(txtCounter1, 1);  
    }  
  
    protected void btnCounter2\_Click(object sender, EventArgs e)  
    {  
        ServerNextCustomer(txtCounter2, 2);  
    }  
  
    protected void btnCounter3\_Click(object sender, EventArgs e)  
    {  
        ServerNextCustomer(txtCounter3, 3);  
    }  
  
    protected void btnPrintToken\_Click(object sender, EventArgs e)  
    {  
        Queue<int> tokenQueue = (Queue<int>)Session["TokenQueue"];  
        lblCurrentStatus.Text = "There are " + tokenQueue.Count.ToString()   
            + " customers before you in the queue";  
  
        if (Session["lastTokenNumberIssued"] == null)  
        {  
            Session["lastTokenNumberIssued"] = 0;  
        }  
  
        int nextTokenNumberToIssue = (int)Session["lastTokenNumberIssued"] + 1;  
        Session["lastTokenNumberIssued"] = nextTokenNumberToIssue;  
        tokenQueue.Enqueue(nextTokenNumberToIssue);  
              
        AddTokenNumbersToListBox(tokenQueue);  
    }  
  
    private void AddTokenNumbersToListBox(Queue<int> tokenQueue)  
    {  
        listTokens.Items.Clear();  
        foreach (int token in tokenQueue)  
        {  
            listTokens.Items.Add(token.ToString());  
        }  
    }  
  
    private void ServerNextCustomer(TextBox textBox, int counterNumnber)  
    {  
        Queue<int> tokenQueue = (Queue<int>)Session["TokenQueue"];  
        if (tokenQueue.Count > 0)  
        {  
            int tokenNumberToBeServed = tokenQueue.Dequeue();  
            textBox.Text = tokenNumberToBeServed.ToString();  
            txtNextToken.Text = "Token Number : " + tokenNumberToBeServed.ToString()   
                + ", please go to Counter " + counterNumnber.ToString();  
            AddTokenNumbersToListBox(tokenQueue);  
        }  
        else  
        {  
            textBox.Text = "No cutomers in Queue";  
        }  
    }  
}

### 72. Real time example of queue collection class

**Two common scenarios, where a stack can be used.**  
**1.** Implementing UNDO functionality   
**2.** Implementing browser back button  
  
In this video, let's implement **BACK** button using stack.

**Step 1:** Create an asp.net web application. Use "WebFormsDemo" as the project name.  
  
**Step 2:** Right click on the project name in solution explorer, and a class file with name = **"BasePage.cs"**. Copy and paste the following code.  
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Web;  
  
namespace WebFormsDemo  
{  
    public class BasePage : System.Web.UI.Page  
    {  
        protected override void OnLoad(EventArgs e)  
        {  
            if (Session["URLStack"] == null)  
            {  
                Stack<string> urlStack = new Stack<string>();  
                Session["URLStack"] = urlStack;  
            }  
  
            if (Request.UrlReferrer != null && !this.Page.IsPostBack   
                && Session["BackButtonClicked"] == null)  
            {  
                Stack<string> urlStack = (Stack<string>)Session["URLStack"];  
                urlStack.Push(Request.UrlReferrer.AbsoluteUri);  
            }  
  
            if (Session["BackButtonClicked"] != null)  
            {  
                Session["BackButtonClicked"] = null;  
            }  
        }  
    }  
}  
  
**Step 3:** Right click on the project name in solution explorer, and a master page with name = **Site.Master**. Copy and paste the following code.  
<table style="width: 500px; border: 1px solid black">  
    <tr>  
        <td colspan="2">  
            <asp:Label ID="lblMessage" runat="server" ForeColor="Red">  
            </asp:Label>  
        </td>  
    </tr>  
    <tr>  
        <td style="width: 100px">  
            <table style="border: 1px solid black; font-family: Arial">  
                <tr>  
                    <td>  
                        <b>Links</b>  
                    </td>  
                </tr>  
                <tr>  
                    <td>  
                        <a href="http://localhost/WebFormsDemo/WebForm1.aspx">WebForm1 </a>  
                    </td>  
                </tr>  
                <tr>  
                    <td>  
                        <a href="http://localhost/WebFormsDemo/WebForm2.aspx">WebForm2 </a>  
                    </td>  
                </tr>  
                <tr>  
                    <td>  
                        <a href="http://localhost/WebFormsDemo/WebForm3.aspx">WebForm3 </a>  
                    </td>  
                </tr>  
                <tr>  
                    <td>  
                        <a href="http://localhost/WebFormsDemo/WebForm4.aspx">WebForm4 </a>  
                    </td>  
                </tr>  
            </table>  
        </td>  
        <td style="width: 400px">  
            <asp:Button ID="btnBack" runat="server" Text="Back" OnClick="btnBack\_Click" />  
            <asp:ContentPlaceHolder ID="ContentPlaceHolder1" runat="server">  
            </asp:ContentPlaceHolder>  
        </td>  
    </tr>  
</table>  
  
**Step 4:** Copy and paste the following code in **Site.Master.cs**  
public partial class Site : System.Web.UI.MasterPage  
{  
    protected void Page\_Load(object sender, EventArgs e)  
    {  
    }  
  
    protected void btnBack\_Click(object sender, EventArgs e)  
    {  
        Session["BackButtonClicked"] = "YES";  
        if (Session["URLStack"] != null)  
        {  
            Stack<string> urlStack = (Stack<string>)Session["URLStack"];  
            if (urlStack.Count > 0)  
            {  
                string url = urlStack.Pop();  
                Response.Redirect(url);  
            }  
            else  
            {  
                lblMessage.Text = "There are no pages in the history";  
            }  
        }  
    }  
}  
  
**Step 5:** Right click on **"site.master"** page in solution explorer and select **"Add Content Page"**. This step should add WebForm1.aspx.   
  
In the code-behind file change the following line  
public partial class WebForm1 : System.Web.UI.Page  
  
**TO**  
public partial class WebForm1 : BasePage  
  
Repeat Step 5, until you add 4 WebForms.  
  
That's it, navigate to http://localhost/WebFormsDemo/WebForm1.aspx and click on the links to navigate to WebForm2.aspx, WebForm3.aspx & WebForm4.aspx   
  
  
Now click on the back button, on the page(Not the browser back  button), and notice that we are able to navigate back.