.Net Programming

Unit 2 - Object Oriented Concept

By – Dr. Jay B. Teraiya

Agenda of Unit 2

- Class and Object
- Encapsulation
- Method Overloading
- Method “ref” and “out” parameter
- Static and Non-Static Members
- Constructor & Destructor
- Operator Overloading
- Modifiers or Specifiers
- Creating and Using Property
- Creating and Using Indexers
- Creating and Using Delegates
- Creating and Using Events with Event Delegate
- Creating and Using Pointers
- Inheritance
- Sealed Class and Abstract Class
- Interface
- Collections

Class and Object

- Class and Object are the basic concepts of all object oriented programming languages.
- A class is a user-defined blueprint or prototype from which objects are created.
- Basically, a class combines the fields or variables and methods(member function which defines actions) into a single unit.
- Example

```
public class Demo
{


}
```

Class and Object

- Class declaration contains only keyword **class**, followed by an **identifier(name of class)**.
- But there are some optional attributes which can be used with class declaration according to the application requirement. In general, class declarations can include these components, in order:
- **Modifiers**: A class can be **public**, **private** or **internal** etc. By **default** modifier of class is **internal**.
- **Keyword class**: A class keyword is used to declare the type class.
- **Class Identifier**: The variable of type class is provided. The identifier(or name of class) should begin with a initial letter which should be capitalized by convention.
- **Base class or Super class**: The name of the class's parent (superclass), if any, preceded by the : (colon). This is optional.
- **Interfaces**:
  - A comma-separated list of interfaces implemented by the class, if any, preceded by the : (colon).
  - A class can implement more than one interface. This is optional.
- **Body**: The class body is surrounded by { } (curly braces).

Class and Object

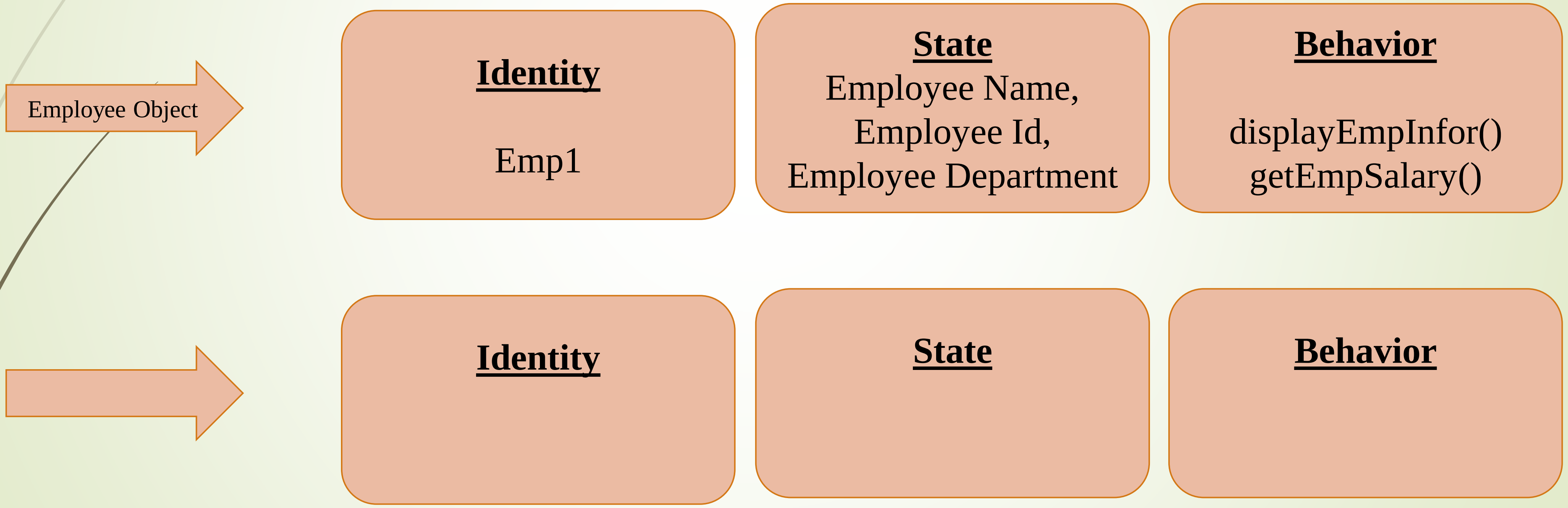
```
// declaring public class
  public class Demo
  {
      // fields or variables
      public int a, b;

      // member function or method
      public void display()
      {
        Console.WriteLine(“Hello! Class & Objects");
      }
  }
```

Class and Object

- **Object** is a basic unit of oop and it represents the real-world entities.
- C# program creates many objects, which as you know**, interact by invoking methods or functions.**
- An object consists of :
- **Identity**:
  - It gives a unique name to an object and enables one object to interact with other objects.
- **State**:
  - It is represented by attributes of an object.
  - It also reflects the properties of an object.
- **Behavior**:
  - It is represented by methods of an object.
  - It also reflects the response of an object with other objects.

Class and Object

- Consider Student, Employee or Bank Account as an object and see the below diagram for its identity, state, and behavior.



Class and Object

- Objects are the real world entities. For example, a graphics program may have objects such as “circle”, “square”, etc.
- An online shopping system might have objects such as “shopping cart”, “customer”, and “product”.
- When an object of a class is created, the class is said to be instantiated.
- All the instances share the attributes and the behavior of the class.
- But the values of those attributes, i.e. the state are unique for each object.
- A single class may have any number of instances (Objects).

Class and Object

- Example

```
using System;
namespace MyNameSpace
{
    public class Semester
    {
        public void display(int sem)
        {
            Console.WriteLine(“Current Semester is {0}”, sem);
        }
    }
    public class Program
    {
        public static void Main(String[] args)
        {
            Semester s4 = new Semester(); //Object 1
            s4.display(4);
            Semester s2 = new Semester(); //Object 2
            s2.display(2);
            Console.ReadLine();
        }
    }
}
```

Encapsulation

- The wrapping up of data and functions into a single unit is known as encapsulation
- The insulation of the data from direct access by the program is called data hiding or information hiding.
- It is the process of enclosing one or more details from outside world through access right.
- The data is not accessible to the outside world, and only those functions which are wrapped in the class can access it.
- These functions provide the interface between the object's data and the program.
- This insulation of the data from direct access by the program is called data hiding or information hiding.

Method Overloading

- Method overloading (Function overloading) is a programming concept that allows programmers to define two or more functions with the same name.

- C# also allows us to define multiple functions with the same name differing in the argument type and order of arguments. This is termed as method overloading.

- There is no need to use any keyword while overloading a function or method either in same class or in derived class.

- While overloading functions or methods, you have to follow the rules that overloaded methods must differ either in number of arguments or the data type of at least one argument.

Method Overloading

- In case of function or method overloading, compiler identifies which overloaded method to execute based on number of arguments and their data types during compilation itself.
- Hence method overloading is an example for compile time polymorphism.

```
public class calculations
    {
        public int add(int x, int y)
        {
            return x + y;
        }

        public int add(int x, int y, int z)
        {
            return x + y + z;
        }
    }
```

```
class Program
    {
        public static void Main(string[] args)
        {
            add ad = new add();
            int i = ad.sum(2, 3);
            Console.WriteLine("Addtion is {0}",i);
            int b = ad.sum(2);
            Console.WriteLine("Addtion is {0}",b);
            Console.Read();
        }
    }
```

Method “ref” and “out” parameter

- **ref** and **out** keywords in C# are used to pass arguments within a method or function.
- **Both indicate that an argument/parameter is passed by reference. By default parameters are passed to a method by value.**
- By using these keywords (ref and out) we can pass a parameter by reference.

Method “ref” and “out” parameter

- **ref keyword -** The ref keyword passes arguments by reference. It means any changes made to this argument in the method will be reflected in that variable when control returns to the calling method.

```
public static string GetNextName(ref int id)
{
    string returnText = "Next-" + id.ToString();
    id += 1;
    return returnText;
}
static void Main(string[] args)
{
    int i = 1;
    Console.WriteLine(“Previous value of integer i:{0}”, i);
    string test = GetNextName(ref i);
    Console.WriteLine(“Current value of integer i:{0}”, i);
}
```

Method “ref” and “out” parameter

- **out keyword -** The out keyword passes arguments by reference. This is very similar to the ref keyword.

```
public static string GetNextNameByOut(out int id)
{
    id = 1;
    string returnText = "Next-" + id.ToString();
    return returnText;
}
static void Main(string[] args)
{
    int i = 0;
    Console.WriteLine("Previous value of integer i:{0}”, i);
    string test = GetNextNameByOut(out i);
    Console.WriteLine("Current value of integer i:{0}”, i);
}
```

Method “ref” and “out” parameter

Ref	out
The parameter or argument must be initialized first before it is passed to ref.	It is not compulsory to initialize a parameter or argument before it is passed to an out.
It is not required to assign or initialize the value of a parameter (which is passed by ref) before returning to the calling method.	A called method is required to assign or initialize a value of a parameter (which is passed to an out) before returning to the calling method.
Passing a parameter value by Ref is useful when the called method is also needed to modify the pass parameter.	Declaring a parameter to an out method is useful when multiple values need to be returned from a function or method.
It is not compulsory to initialize a parameter value before using it in a calling method.	A parameter value must be initialized within the calling method before its use.
When we use REF, data can be passed bi-directionally.	When we use OUT data is passed only in a unidirectional way (from the called method to the caller method).
The parameter or argument must be initialized first before it is passed to ref.	It is not compulsory to initialize a parameter or argument before it is passed to an out.
It is not required to assign or initialize the value of a parameter (which is passed by ref) before returning to the calling method.	A called method is required to assign or initialize a value of a parameter (which is passed to an out) before returning to the calling method.

Static and Non-Static Members

- In C#, static means something which cannot be instantiated. You cannot create an object of a static class and cannot access static members using an object.
- C# **classes**, **variables**, **methods**, **properties**, **operators**, **events**, and **constructors** can be defined as static using the static modifier keyword.
- In C#, if we use a static keyword with class members, then there will be a single copy of the type member.
- And, all objects of the class share a single copy instead of creating individual copies.

Static and Non-Static Members

- **C# Static Variables** - If a variable is declared static, we can access the variable using the class name. For example,

```
using System;
namespace TokenSystem
{
  class Token
  {
    // static variable
    public static int count = 0;
  }

  class Program
  {
    static void Main(string[] argos)
    {
          // access static variable
          Console.WriteLine(“Token Value: " + Token.count);
    }
  }
}
```

Static and Non-Static Members

- **C# Static Variables Vs Instance Variables**

- In C#, every object of a class will have its own copy of instance variables.
- However, if we declare a variable static, all objects of the class share the same static variable. And, we don't need to create objects of the class to access the static variables.

Static and Non-Static Members

- **C# Static Methods**
- Just like static variables, we can call the static methods using the class name.
- Here, we have accessed the static method directly from Program classes using the class name.
- When we declare a method static, all objects of the class share the same static method.

```
class Test {

  public static void display() {....}

}

class Program {
  static void Main(string[] args) {

    Test.display();
  }
```

Static and Non-Static Members

- **C# Static Class**
- In C#, when we declare a class as static, we cannot create objects of the class.

```
static class Test {
    static int a = 5;
    static void display() {

      Console.WriteLine("Static method");
    }

    static void Main(string[] args) {

      // creating object of Test will give compile time error.
      Test t1 = new Test();
      Console.WriteLine(a);
      display();
    }
  }
```

Static and Non-Static Members

- **C# Static Class**

- Notice the field and method of the static class are also static because we can only have **static members inside the static class**.
- We cannot inherit a static class in C#. For example,

```
static class A {
  ...
}

// Error Code
class B : A {
  ...
}
```

Constructor & Destructor

- Constructor is a special member function of a class that is executed whenever we create new objects of that class.

- Definition : “**Special method of the class that will be automatically invoked when an instance of the class is created is called as constructor**”.

- Constructor will have exact same name as the class and it does not have any return type not even void.

- Constructors are specially used to initialize data members.

- By default, C# creates default constructor internally. Default constructor does not have any parameter.

- Class can have any number of constructors.

Constructor & Destructor

- Constructors can be classified as follows.
- **Default Constructor**
  - When you do not declare any type of constructor, the class will call its default constructor which has a default public access modifier.
  - The default constructor is a parameter less constructor which will be called by a class object.
- **Parameterized Constructor**
  - When an object is declared in a parameterized constructor, the initial values have to be passed as arguments to the constructor.
  - You can also call it as constructor overloading.
- **Copy Constructor**
  - Copy constructor is the parameterized constructor which takes a parameter of the same type.
  - It allows you to initialize a new object with the existing object values.

Constructor & Destructor

```
using System;

namespace Constructor
{
    class Square
    {
        private int length;

        //Default Constructor
        public Square()
        {
                length = 1;
        }
        //Paramerized Constructor
        public Square(int l)
        {
  //A Constructor is used to initilize private fields of a class
                length = l;
        }
        public int Area()
        {
            return length*length;
        }
    }
```

```
class MainClass
{
 public static void Main()
 {
    //Calling Default Constructor
    Square squre1 = new Square();
    int Area = squre1.Area();
    Console.WriteLine(Area);

  //Calling Parametrized Constructor
    Square MySquare = new Square(10);
    int myarea = MySquare.Area();
    Console.WriteLine(myarea);
  }
}


}
```

Output :

```
1
100
```

Constructor & Destructor

```
using System;
namespace Test
{
    public class Person
    {
        private int m_PID;
        private string m_FName, m_LName, m_City;
        public Person()
        {
            m_PID = 19929;
            m_FName = “Jay";
            m_LName = “Teraiya";
            m_City = “Gandhinagar";
        }
         public Person(string firstName, string lastName)
        {
            m_FName = firstName;
            m_LName = lastName;
        }
```

```
public Person(Person person)
        {
            m_PID = person.m_PID;
            m_FName = person.m_FName;
            m_LName = person.m_LName;
            m_City = person.m_City;
        }
    }

class Program
    {
        static void Main(string[] args)
        {
          Person p1 = new Person(); //Default
          Person p2 = new Person(“Jay”,
“Teraiya”); //Parameterized
          Person p3 = new Person(p2); //Copy
Constructor
        }
    }
}
```

Constructor & Destructor

- A destructor is a special member function of a class that is executed whenever an object of its class goes out of scope.
- A destructor will have exact same name as the class prefixed with a tilde (~) and **it can neither return a value nor can it take any parameters**.
- Destructor can be very useful for releasing resources before coming out of the program like closing files, releasing memories etc.
- A class can **have only one destructor**.
- Destructors **cannot be inherited or overloaded**.
- A destructor **does not take modifiers**.

Constructor & Destructor

```
using System;
namespace Destructor
{
    class Example
    {
        public Example() //Default Constructor
        {
            Console.WriteLine("Constructor");
        }
        ~Example() //Destructor
        {
            Console.WriteLine("Destructor");
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Example x = new Example();
           Console.ReadKey();
        }
    }
```

Constructor & Destructor

	Constructor	Destructor
Purpose	Constructor is used to **initialize the instance of a class**.	Destructor **destroys the objects** when they are no longer needed.
When Called	Constructor is Called when new instance (object) of a class is created.	Destructor is called when instance of a class is deleted or released.
Memory Management	Constructor **allocates the memory**.	Destructor **releases the memory**.
Arguments	Constructors can have arguments.	Destructor can not have any **arguments**.
Overloading	**Overloading** of constructor is possible.	Overloading of Destructor is not possible.
Name	Constructor has the same name as class name.	Destructor also has the same name as class name but with **(~) tiled operator**.

Syntex	ClassName(Arguments)
	{
	    //Body of Constructor
	}
		~ ClassName()
		{
		    //Body of Destructor
		}

Operator Overloading

- Every operator has it’s predefined meaning, most of them are given additional meaning through the concept of **Operator Overloading**.
- Suppose ‘+’ sign is used for addition
- But, we can not use ‘+’ as concatenation?
- Suppose, we have String 1 = “NFSU” & String 2 = “Gandhinagar”
- Can we concat these two strings using + operator like “NFSU Gandhinagar”?
- **When any operator is overloaded, keep in mind that its original meaning is not lost.**

Operator Overloading

- Consider an example of user defined data type **int** with the operators **+, -, * and /** provides support for mathematical operations.
- To make operations on a user – defined data type is difficult as the operations are built – in.
- An operator can be overloaded by defining a function to it.
- The function is declared using the **operator** keyword.
- The operator function must be static.
- The operator function must have the keyword **operator** followed by the **operator to be overridden.**
- The **arguments** of the function are the **operands**.
- The return value of the function is the result of the operation.

Operator Overloading

- For example, to overload the + operator, the following syntax is defined.

```
<access specifier> static classname operator + (parameters OR arguments)
  {
   //Code to be executed
  }
```

```
public static Addition operator  +(Addition a1, Addition a2)
{
}
```

Operator Overloading

Operators	Description
+, -, !, ~, ++, --	These unary operators take one operand can be overloaded
+, -, *, /, %	These binary operators take two operands and can be overloaded
==, !=, <, >, <=, >=	The comparison operators can be overloaded
&&, ||	The conditional logical operators **cannot be overloaded** directly and evaluated by using the & and | which can be overloaded
+=, -=, *=, /==, %==	The assignment operators **cannot be overloaded**
=, ? :, - >, new, sizeof, typeof	These operators **cannot be overloaded**

Operator Overloading

```
using System;

namespace Test
{
    class Distance
    {
        public int Values;

        public static Distance operator +(Distance d1, Distance d2)
        {
            Distance d = new Distance();
            d.Values = d1.Values + d2.Values;
            return d;
        }
```

```
class Program
        {
            static void Main(string[] args)
            {
                Distance d1 = new Distance();
                Distance d2 = new Distance();
                d1.Values = 10;
                d2.Values = 20;
                Distance d3 = d1 + d2;
                Console.WriteLine("Sum is {0}",
                                      d3.Values);
                Console.Read();
            }
        }
    }
}
```

Operator Overloading

```
using System;
namespace Demo
{
    class calculation
    {
        int a, b, c;
        public calculation()
        {
            a = b = c = 0;
        }
        public calculation(int x, int y, int z)
        {
            a = x;
            b = y;
            c = z;
        }
        public static calculation operator ++ (calculation op1)
        {
            op1.a++;
            op1.b++;
            op1.c++;
            return op1;
```

```
public void ShowResult()
        {
            Console.WriteLine(a + "," + b + "," + c);
            Console.ReadLine();
        }
    }
```

```
class Program
    {
        static void Main(string[] args)
        {
           calculation i = new calculation(10, 20, 30);
            i++;
            i.ShowResult();
            Console.ReadLine();
        }
    }
}
```

By - Dr. Jay B. Teraiya

Modifiers or Specifiers

- Access modifiers defines the scope of a class members. A class member can be variables or functions.

- Access modifiers are keywords used to specify the declared accessibility of a member or a type.

- **Why to use access modifiers?**
  - Access modifiers are an integral part of object-oriented programming.
  - They support the concept of **encapsulation**, which promotes the idea of **hiding functionality**.
  - Access modifiers allow you to define who does or doesn't have access for certain features.

Modifiers or Specifiers

- In C# Modifiers can be divided in five categories.
- **Public** Access Specifier
- **Private** Access Specifier
- **Protected** Access Specifier
- **Internal** Access Specifier
- **Protected Internal** Access Specifier

Modifiers or Specifiers

- **Public** is the most common access specifier in C#.
- With public we can access from anywhere, that means there is no restriction on accessibility.
- The scope of the accessibility is inside class as well as outside.
- The keyword **public** is used for it.
- **Accessibility**:
- Can be accessed by objects of the class
- Can be accessed by derived classes

Modifiers or Specifiers

```
using System;
namespace Demo
{
    class Access
    {
        public int num1;
    }
    class Program
    {
        static void Main(string[] args)
        {
            Access ob1 = new Access();
            //Direct access to public members
            ob1.num1 = 100;
            Console.WriteLine("Number one value in main {0}", ob1.num1);
            Console.ReadLine();
        }
    }
}
```

Modifiers or Specifiers

- **Private** members are accessible only within the body or scope of the class or the structure in which they are declared.

- The private members cannot be accessed outside of the class and it is the least permissive access level.

- The keyword **private** is used for it.

- **Accessibility**:
- Cannot be accessed by object
- Cannot be accessed by derived classes

Modifiers or Specifiers

```
using System;
namespace Demo
{
    class Access
    {
        public int num1;
        private int num2;
    }
   class Program
    {
        static void Main(string[] args)
        {
            Access ob1 = new Access();
            //Direct access to public members
            ob1.num1 = 100;
            //Access to private member is not permitted
            ob1.num2 = 10;
            Console.WriteLine("Number one value in main {0}", ob1.num1);
            Console.ReadLine();
        }
    }
```

Modifiers or Specifiers

- **Protected** - The accessibility of protected is limited within the class or structure and the class derived (Inherited) from base (this) class.

- A protected member of a base class is accessible in a derived class, only if the access takes place through the derived class type.

- The keyword **protected** is used for it.

- **Accessibility**:
- Cannot be accessed by object
- Access by derived classes

Modifiers or Specifiers

```
using System;
namespace Test
{
    class access
    {
        // Integer Variable declared as protected
        protected int age;
        public void print()
        {
            Console.WriteLine("\nMy Age is " + age);
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            access ac = new access();
            Console.Write("Enter your Age:\t");
            // Raise error because of its protection level
            ac.age = Convert.ToInt32(Console.ReadLine());
            ac.print();
            Console.ReadLine();
        }
```

Modifiers or Specifiers

```
using System;
namespace Test
{
    class Access
    {
        // Integer Variable declared as protected
        protected int age=0;
        public void Print()
        {
            Console.WriteLine("\nMy Age is " + age);
        }
    }
    class Access1 : Access
    {
        public void Print()
        {
            Console.WriteLine("Enter Your Age:");
            age = int.Parse(Console.ReadLine());
            Console.WriteLine("\nMy Age is " + age);
        }
    }
```

```
class Program
    {
        static void Main(string[] args)
        {
            Access1 ac = new Access1();
            ac.Print();
            Console.ReadLine();
        }
    }
}
```

```
Output:
```

Modifiers or Specifiers

- **The internal** access specifier hides its member variables and methods from other classes and objects, that is resides in **other namespace**.
- The variable or classes that are declared with internal can be access by any member within application.
- We can declare a class and it's members as internal.
- Internal members are accessible only within the **same assembly**.
- In other words, access is limited exclusively to classes defined within the current project assembly.
- The keyword **internal** is used for it.

- **Accessibility**:
- The variable or classes that are declared with internal can be access by any member within application.
- It is the **default access specifiers** for a class in C# programming.

Modifiers or Specifiers

```
using System;

namespace First_Prg
{
    class access
    {
        // Integer Variable declared as internal
        internal int age;
        public void print()
        {
            Console.WriteLine("\nMy Age is " + age);
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
          access ac = new access();
            Console.Write("Enter your Age:\t");
           // Accepting value in internal variable
          ac.age =  Convert.ToInt32(Console.ReadLine());
            ac.print();

        }
    }
}
```

Modifiers or Specifiers

- **The protected internal** accessibility means **protected** OR **internal**, not protected AND internal.
- In other words, a protected internal member is accessible from **any class in the same assembly, including derived classes**.
- The protected internal access specifier allows its members to be accessed in derived class, containing class or classes within same application.
- However, this access specifier rarely used in C# programming but it becomes important while implementing inheritance.
- **Accessibility**:
- Within the class in which they are declared
- Within the derived classes of that class available within the same assembly
- Outside the class within the same assembly
- Within the derived classes of that class available outside the assembly

Modifiers or Specifiers

```
using System;

namespace First_Prg
{
    class access
    {
       // String Variable declared as protected internal
       protected internal string name;
        public void print()
        {
            Console.WriteLine("\nMy name is " + name);
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            access ac = new access();
            Console.Write("Enter your name:\t");
            // Accepting value in internal variable
            ac.name = Console.ReadLine();
            ac.print();
            Console.ReadLine();
        }
    }
}
```

Modifiers or Specifiers

- A **default** access is used if no access modifier is specified in a member declaration.
- The following list defines the default access modifier for certain C# types:

C# Types	Description
enumeration	The default and only access modifier supported is **public**.
Class	The default access for a class is **internal**. It may be explicitly defined using any of the access modifiers.
Interface	The default and only access modifier supported is **public**.
structure	The default access is **internal.** It may be explicitly defined using any of the access modifiers.
Interface and enumeration members are always public and no other access modifiers are allowed.	

Creating and Using Property

- In C#, properties are nothing but natural extension of data fields.
- They are usually known as 'smart fields' in C# community.
- We know that data encapsulation and hiding are the two fundamental characteristics of any object oriented programming language.
- In C#, data encapsulation is possible through either classes or structures.
- Usually inside a class, we declare a data field or variable as private and put a set of public SET and GET methods to access the data fields.

Creating and Using Property

- Properties are special kind of class member, In properties we use predefined Set and Get method. They use assessors through which we can read, written or change the values of the private fields.
- We cannot access these fields from outside the class , but we can accessing these private fields through properties. A property is a combination of variable and a method.
- The get method is used to returns value from the property. The set method is used to assign a new value to the property.
- Syntax

```
Public <return type> <PropertyName>
    {
        get
        {
            return <var>;
        }
        set
        {
            <var> = value;
        }
    }
```

Creating and Using Property

```
class Example
    {
        private int number;
        public int Number
        {
            get
            {
                return number;
            }
            set
            {
                number = value;
            }
        }
    }
```

```
class Program
 {
 static void Main(string[] args)
  {
    Example example = new Example();
    // set { }
    example.Number = 5;
    // get { }
   Console.WriteLine(example.Number);
   Console.Read();
   }
 }
```

Output :

```
5
```

Creating and Using Indexers

- Indexer is a new concept introduced by C#.
- Indexers are also known as the **Smart Arrays or Parameterized Property in C#.**
- An **Indexer is a special type of property** that allows a class or structure to be accessed the same way as array for its internal collection.
- In short, Indexer is the concept that object act as an array.
- Indexer an object to be indexed in the same way as an array.
- Indexer modifier can **be private, public, protected or internal**.
- The return type can be any valid **C# types**.
- **It is same as property except that it defined with this keyword with square bracket and parameters.**
- It can be used for overloading a [] operator.
- (Indexers can be overloaded).

Creating and Using Indexers

Syntax :

```
public <return type> this [argument list]
   {
       get
       {
       //code for get
       }
       set
       {
       //code for get
       }
   }
```

- The **access modifiers** used for an indexer is **public**.
- **Return type** can be any valid C# data type, such as string or integer.
- The **this** keyword shows that the object is of the current class.
- The **argument list** specifies the parameter of the indexer. C# indexer **must have at least one parameter** otherwise compiler gives an error.
- The **get** and **set** portions of the syntax are known as **accessors**.

Creating and Using Indexers

```
class sample
    {
       private string[] name = new string[3];

       public string this[int index]
        {
            get
            {
                if (index < 0 || index >= name.Length)
                {
                    return null;
                }
                else
                {
                    return name[index];
                }
            }

            set
            {
                name[index] = value;
            }
        }
    }
```

```
class Program
    {
        static void Main(string[] args)
        {
            sample s = new sample();
            s[0] = “NFSU";
            s[1] = “Gandhinagar";
            s[2] = “Campus";
            for (int i = 0; i <= 2; i++)
            {
                Console.WriteLine(s[i]);
            }
            Console.ReadKey();
        }
    }
```

Output :

```
NFSU
Gandhinagar
Campus
```

Properties and Indexers

Properties	Indexers
Properties don't require **this** keyword	Indexers are created with this keyword
Properties are identified by their **names**	Indexers are identified by **signature**
Properties cannot take any arguments	Indexers are known as parameterized properties
Properties are also known as the **smart fields**	Indexers are also known as **smart arrays**
A get accessor of a property has no parameters & A set accessor of a property contains the implicit value parameter.	Indexers in C# must have **atleast one parameter** & it also supports more than one different types of parameters

Syntax :
<access_modifier> <return_type> <property_name>
{
get { }
set { }
}
	Syntax :
	<access_modifier> <return type> this [argument list]
	{
	get { }
	set { }
	}

Creating and Using Delegates

- A function can have one or more parameters of different data types, but what if you want to pass a function itself as a parameter?
- How does C# handle the callback functions or event handler?
- The answer is - **delegate**.
- **A delegate is like a pointer to a function.**
- It is a reference type data type and it holds the reference of a method.
- All the delegates are implicitly derived from System.Delegate class.

Creating and Using Delegates

- A delegate can be declared using delegate keyword followed by a function signature as shown below.

- **Syntax**
  - <access modifier> delegate <return type> <delegate_name>(<parameters>)

- The following **example** declares a Print delegate.
  - public delegate void Print(int value);

- The Print delegate shown above, can be used to point to any method that has same return type & parameters declared with Print.

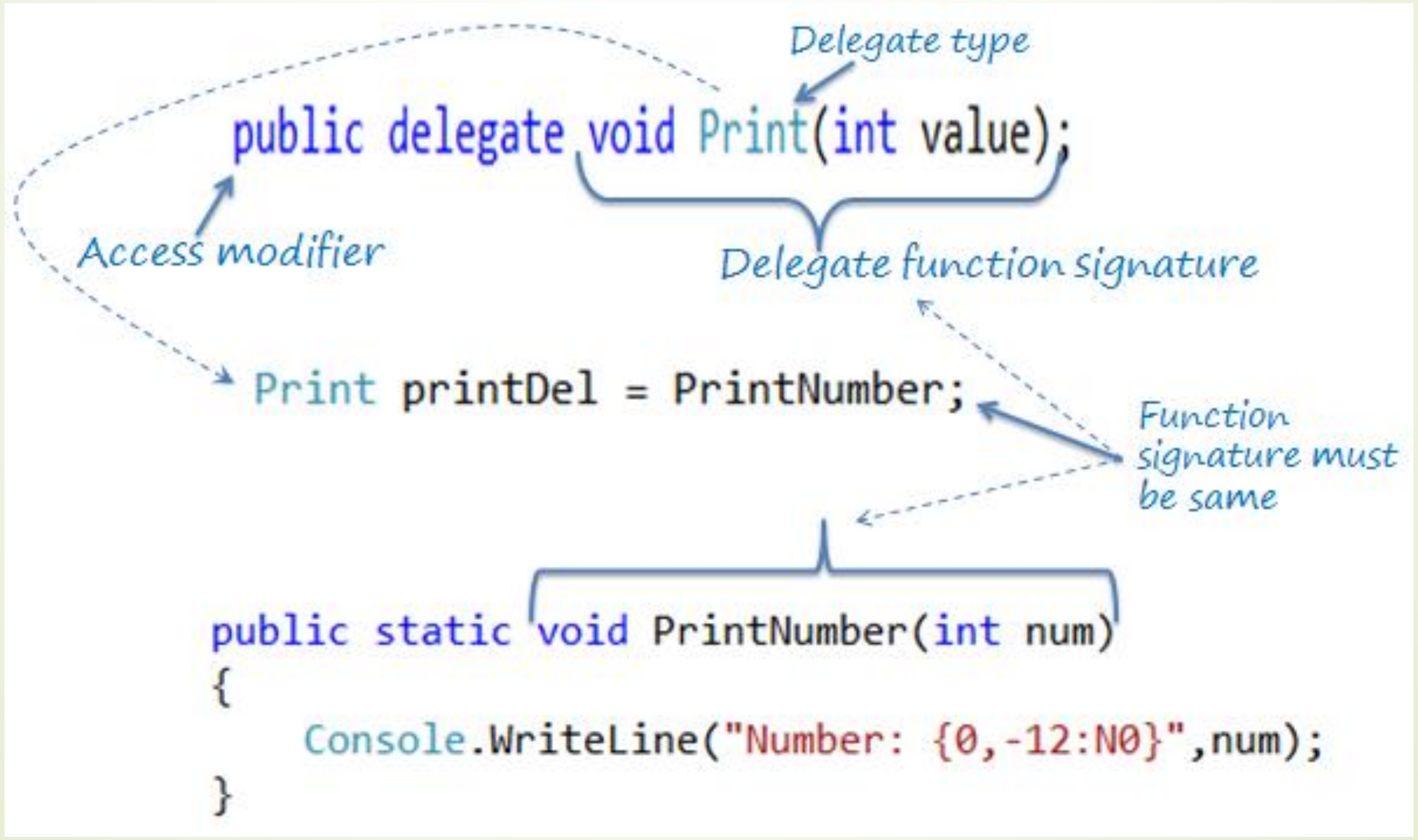
Creating and Using Delegates

```
using System;

namespace Demo
{
    class Program
    {
        // declare delegate
        public delegate void Print(int value);

        static void Main(string[] args)
        {
      // Print delegate points to PrintNumber
            Print printDel = PrintNumber;

            printDel(100000);
            printDel(200);

     // Print delegate points to PrintMoney
            printDel = PrintMoney;

            printDel(10000);
            printDel(200);
            Console.ReadLine();
        }
```

```
public static void PrintNumber(int num)
{
 Console.WriteLine("Number: {0,-12:N0}", num);
 Console.ReadLine();
}
 public static void PrintMoney(int money)
{
 Console.WriteLine("Money: {0:C}", money);
 Console.ReadLine();
}
}
}
```

Output :

Creating and Using Delegates



```
public delegate void Print(int value);

Print printDel = PrintNumber;

public static void PrintNumber(int num)
{
    Console.WriteLine("Number: {0,-12:N0}",num);
}
```

Creating and Using Delegates

- In the above example, we have declared Print delegate that accepts int type parameter and returns void.
- In the Main() method, a variable of Print type is declared and assigned a PrintNumber method name.
- Now, invoking Print delegate will in-turn invoke PrintNumber method.
- In the same way, if the Print delegate variable is assigned to the PrintMoney method, then it will invoke the PrintMoney method.

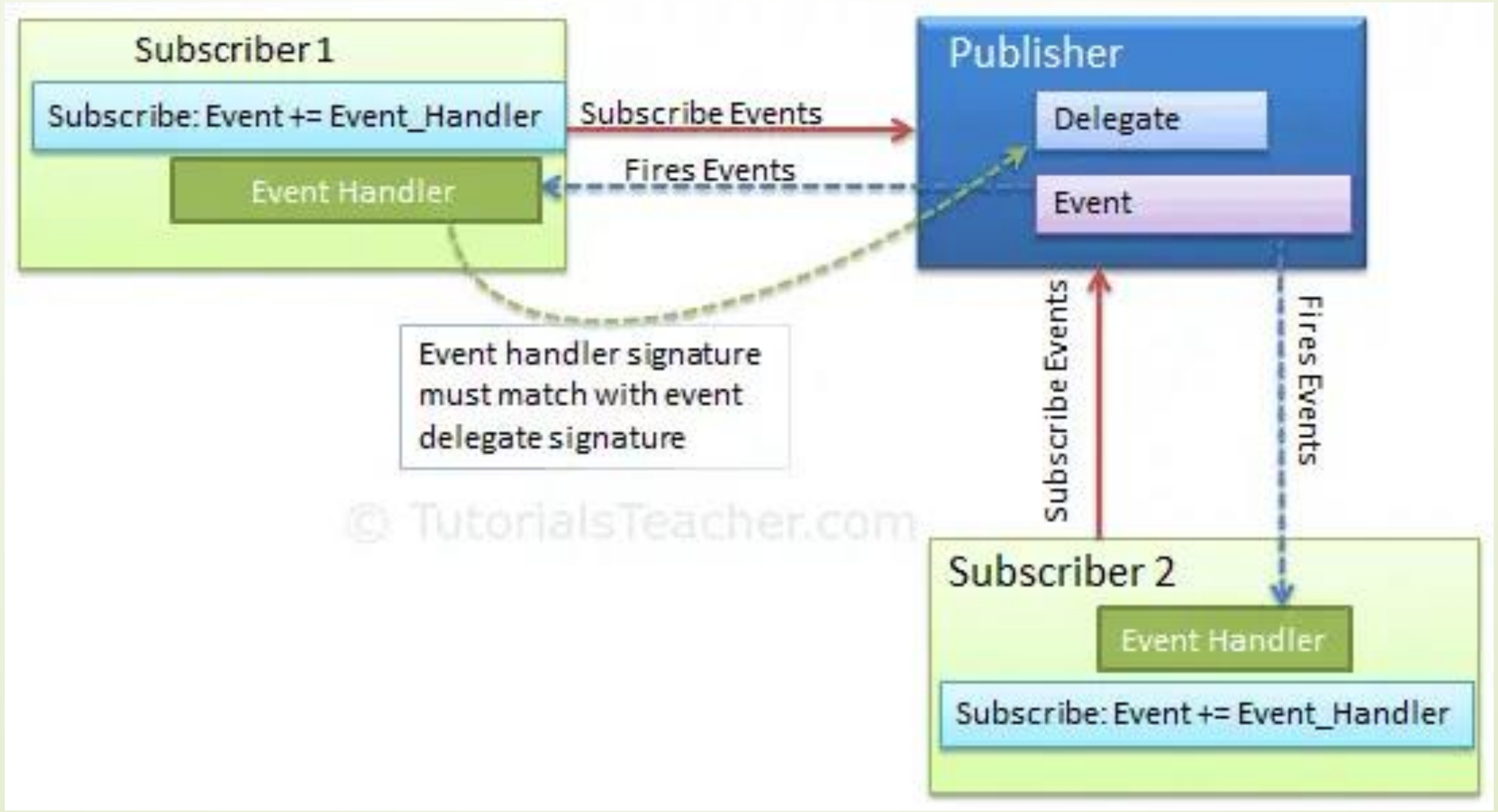
Creating and Using Delegates

- The delegate can be invoked like a method because it is a reference to a method.
- Invoking a delegate will in-turn invoke a method which id referred to.
- The delegate can be invoked by two ways: using () operator or using the Invoke() method of delegate as shown below.

```
Print printDel = PrintNumber;

printDel.Invoke(10000);

        //or
printDel(10000);
```

Creating and Using Events with Event Delegate

- An event is a notification sent by an object to signal the occurrence of an action.
- The class who raises events is called **Publisher**, and the class who receives the notification is called **Subscriber**.



public delegate void Notify(); // delegate public class ProcessBusinessLogic { public event Notify ProcessCompleted; // event }

Creating and Using Events with Event Delegate

- Declare an Event
- An event can be declared in two steps:
  - Declare a delegate.
  - Declare a variable of the delegate with event keyword.
- The following example shows how to declare an event in publisher class.

```
public delegate void Notify();  // delegate

public class ProcessBusinessLogic
{
    public event Notify ProcessCompleted; // event

}
```

public delegate void Notify(); // delegate public class ProcessBusinessLogic { public event Notify ProcessCompleted; // event }

Creating and Using Events with Event Delegate

```
public delegate void Notify();  // delegate

public class ProcessBusinessLogic
{
    public event Notify ProcessCompleted; // event

    public void StartProcess()
    {
        Console.WriteLine("Process Started!");
        // some code here..
        OnProcessCompleted();
    }

    protected virtual void OnProcessCompleted() //protected virtual method
    {
        //if ProcessCompleted is not null then call delegate
        ProcessCompleted?.Invoke();
    }
}
```

public delegate void Notify(); // delegate public class ProcessBusinessLogic { public event Notify ProcessCompleted; // event }

Creating and Using Events with Event Delegate

```
class Program
{
    public static void Main()
    {
        ProcessBusinessLogic bl = new ProcessBusinessLogic();
        bl.ProcessCompleted += bl_ProcessCompleted; // register with an event
        bl.StartProcess();
    }

    // event handler
    public static void bl_ProcessCompleted()
    {
        Console.WriteLine("Process Completed!");
    }
}
```

Creating and Using Pointers

- **A pointer is nothing more than a variable that holds the address in memory of another variable.**
- In C#, pointers can only be used on **value types and arrays**.
- **unsafe keyword** - Typically when we write code in C# by default it is what's known **as safe code**. Unsafe code allows you to **manipulate memory directly**, in the normal world of C# and the .NET Framework, this is seen as potentially dangerous, and as such you have to mark your code as unsafe.
- Using unsafe code, you are loosing garbage collection and whilst directly accessing memory, you have the possibility of accessing memory that you didn't want to and causing untold problems with your application.

Creating and Using Pointers

```
static void Main(string[] args)
{
    unsafe
    {
        int age = 32;
        int* age_ptr;
        age_ptr = &age
        Console.WriteLine("age = {0}", age);
        Console.WriteLine("age_ptr = {0}", *age_ptr);
    }
}
```

OR

```
unsafe static void Main(string[] args)
{
        ------------
        ------------
}
```

Inheritance

- Inheritance is where one class (child class) inherits the properties of another class (parent class).
- Inheritance is a mechanism of acquiring the features and behaviors of a class by another class.
- The class whose members are inherited is called the base class, and the class that inherits those members is called the derived class.
- Inheritance implements the IS-A relationship.
- **Example**
- “He had many inherited property of his father”

Inheritance

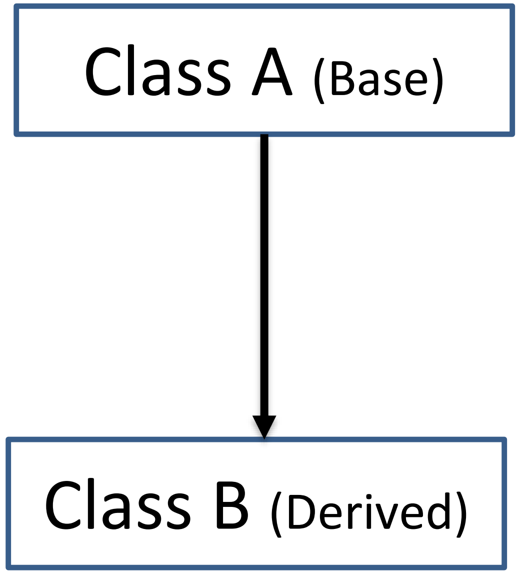
- **Advantages of Inheritance**
- Reduce code redundancy
- Provides code reusability
- Reduces source code size and improves code readability
- Code is easy to manage and divided into parent and child classes
- **Disadvantages of Inheritance**
- In Inheritance base class and child classes are tightly coupled. Hence If you change the code of parent class, it will get affects to the all the child classes.
- In class hierarchy many data members remain unused and the memory allocated to them is not utilized.
- Hence affect performance of your program if you have not implemented inheritance correctly.

Inheritance

- Single Inheritance
- Multi-level Inheritance
- Multiple Inheritance
- Multipath Inheritance
- Hierarchical Inheritance
- Hybrid Inheritance

Inheritance

- **Single Inheritance**



Single Inheritance

In single inheritance, a derived class is created from a single base class.

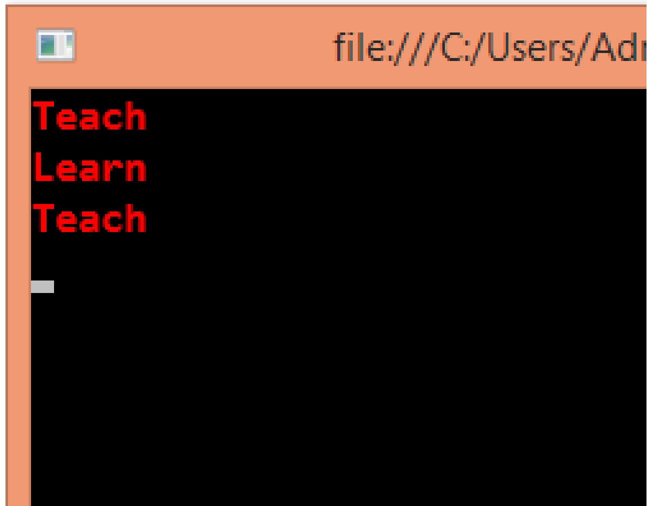
Inheritance

- **Single Inheritance**

```
//Base Class
class Teacher
    {
     public void Teach()
        {
          Console.WriteLine("Teach");
        }
    }
```

```
//Derived Class
class Student : Teacher
    {
      public void Learn()
        {
          Console.WriteLine("Learn");
        }
    }
```

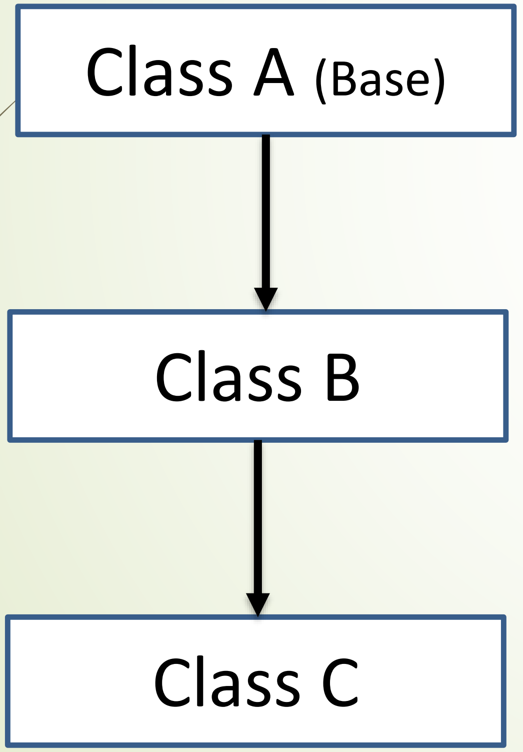
```
class Program
    {
        static void Main(string[] args)
        {
            Teacher d = new Teacher();
            d.Teach();
            Student s = new Student();
            s.Learn();
            s.Teach();
            Console.ReadKey();
        }
```


file:///C:/Users/Adr

```
Teach
Learn
Teach
```

Inheritance

- **Multilevel Inheritance**



Multi-level Inheritance

In Multi-level inheritance, a derived class is created from another derived class.

Inheritance

- **Multilevel Inheritance**

```
//Base Class
class Teacher
    {
     public void Teach()
        {
          Console.WriteLine("Teach");
        }
    }
```

```
//Base,Derived Class
    class Student : Teacher
    {
        public void Learn()
        {

         Console.WriteLine("Learn");
        }
    }
```

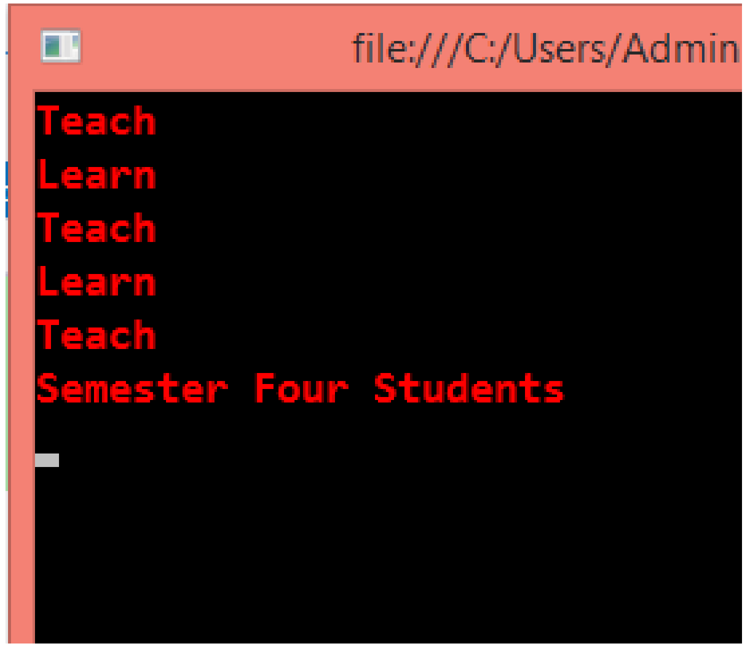
```
//Derived Class
    class SemFour : Student
    {
       public void FourthSem()
        {
         Console.WriteLine("Semester
         Four Students");
        }
    }
```

```
class Program
    {
        static void Main(string[] args)
        {
            Teacher d = new Teacher();
            d.Teach();

            Student s = new Student();
            s.Learn();
            s.Teach();

            SemFour Sem = new SemFour();
            Sem.Learn();
            Sem.Teach();
            Sem.FourthSem();

            Console.ReadKey();
        }
```

Inheritance

- **Multilevel Inheritance**

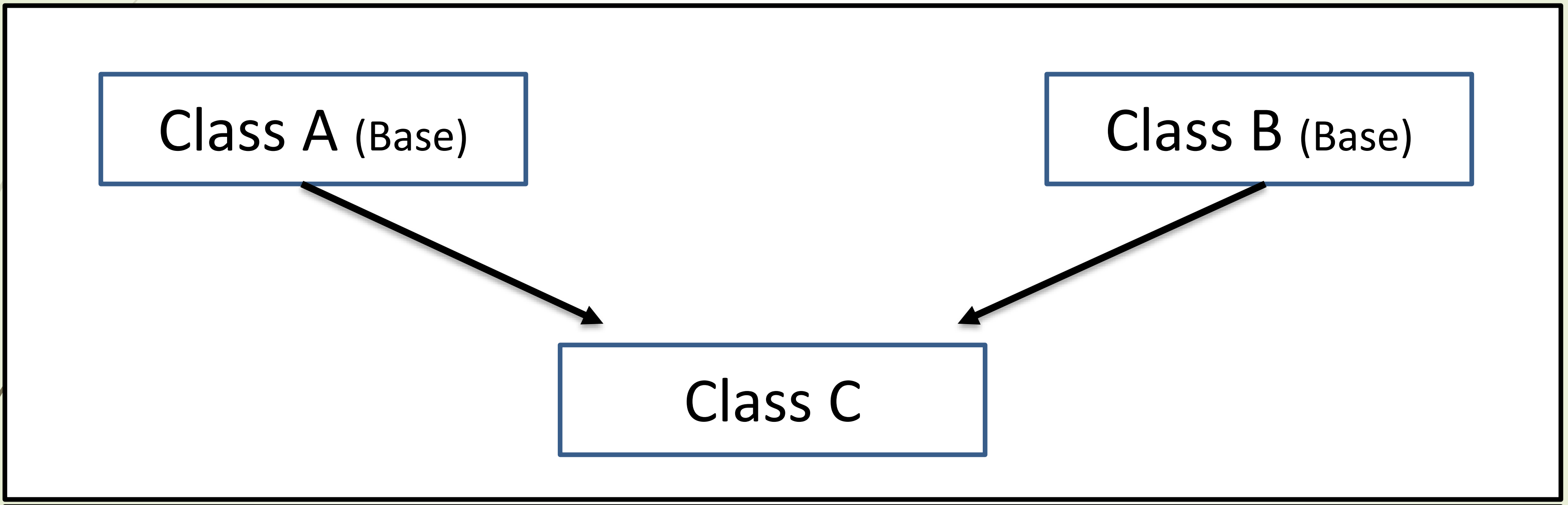
Output :



```
file:///C:/Users/Admin
Teach
Learn
Teach
Learn
Teach
Semester Four Students
```

Inheritance

- **Multiple Inheritance**

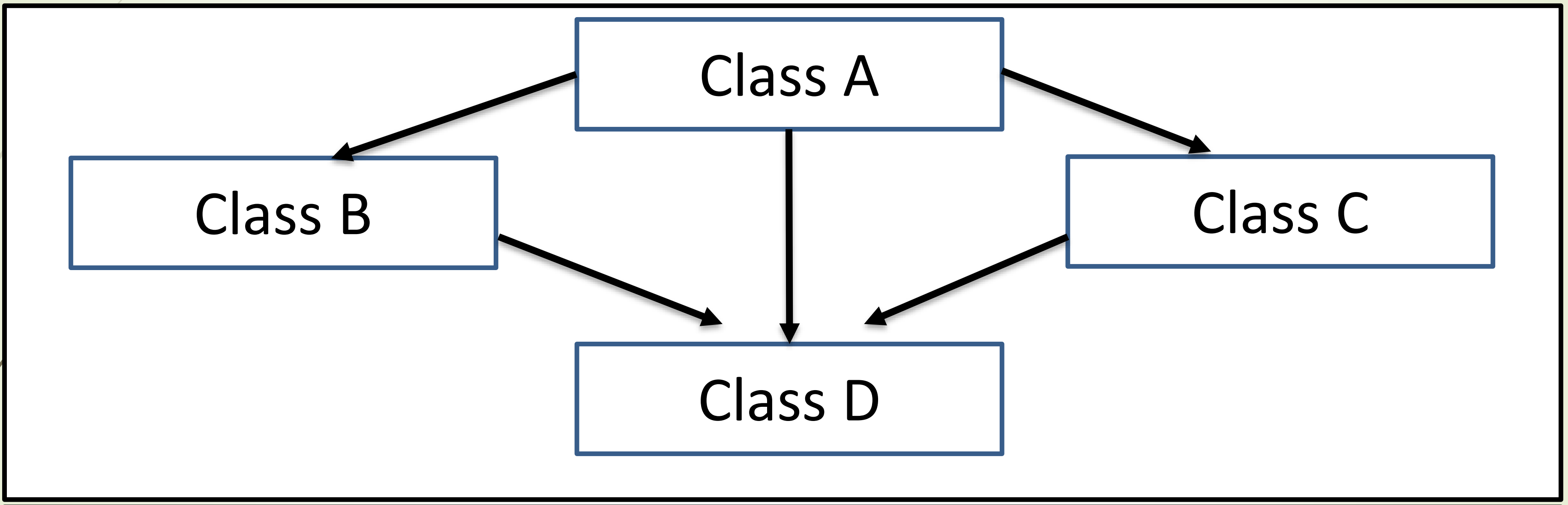


Multiple Inheritance

- In Multiple inheritance, a derived class is created from more than one base class.
- **This inheritance is not supported by C#.**

Inheritance

- **Multipath Inheritance**

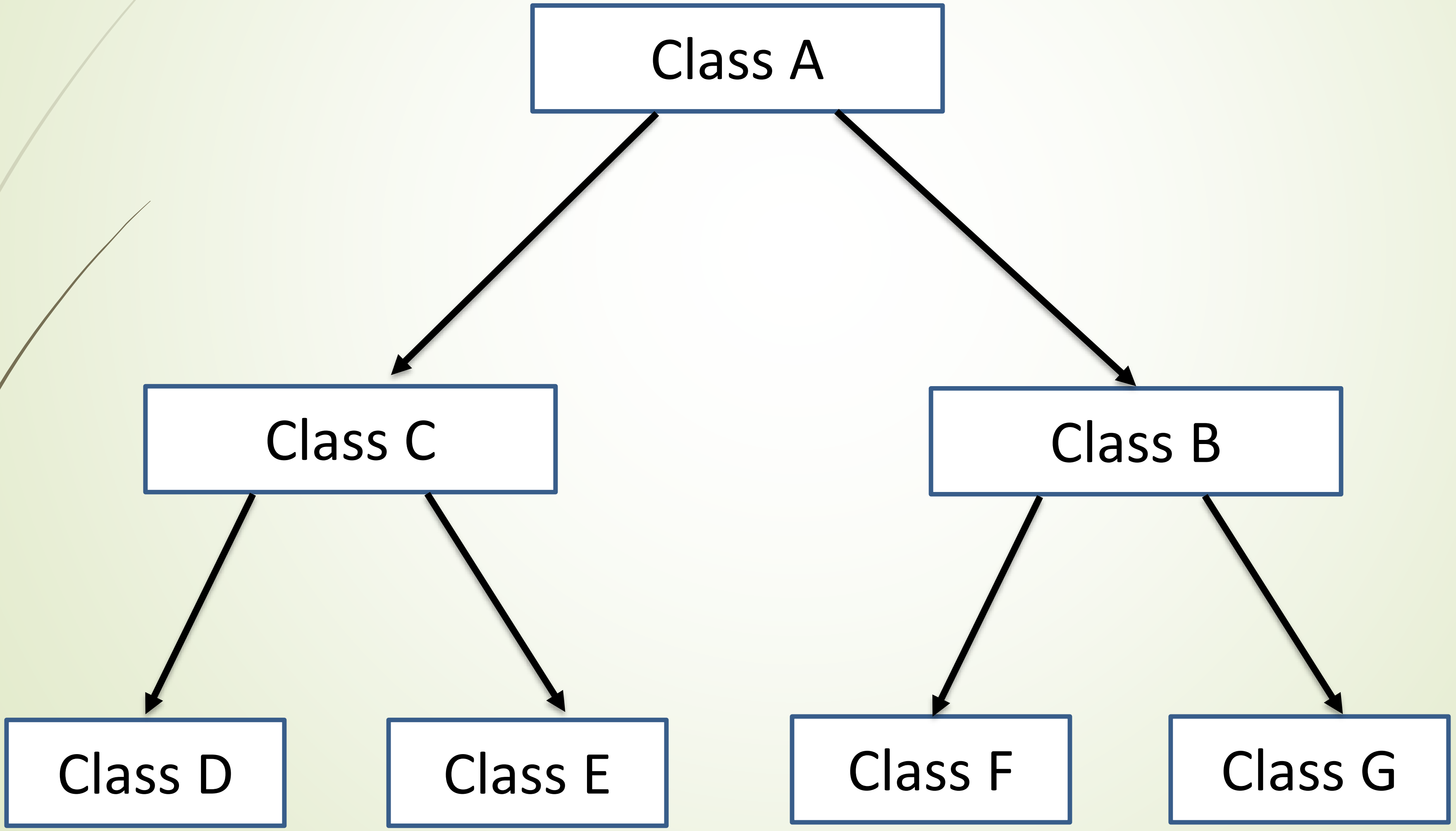


Multipath Inheritance

- In Multipath inheritance, a derived class is created from another derived classes and the same base class of another derived classes.
- **This inheritance is not supported C#.**

Inheritance

- **Hierarchical Inheritance**



Inheritance

- **Hierarchical Inheritance**

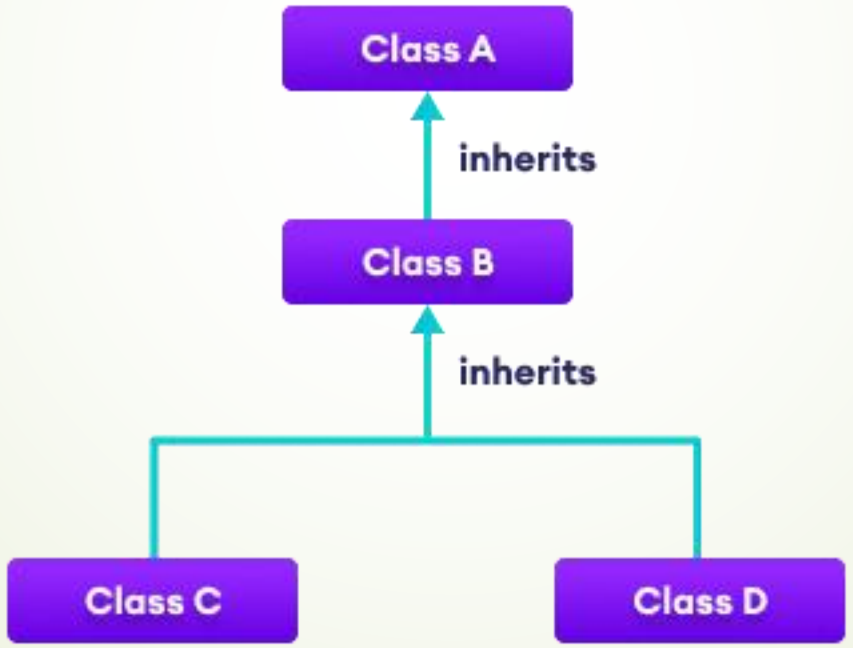
```
//Base Class
class A
{
    public void fooA()
    {
        //TO DO:
    }
}

//Derived Class
class B : A
{
    public void fooB()
    {
        //TO DO:
    }
}

//Derived Class
class C : A
{
    public void fooC()
    {
        //TO DO:
    }
```

```
//Derived Class
class D : C
{
    public void fooD()
    {
        //TO DO:
    }
}
//Derived Class
class E : C
{
    public void fooE()
    {
        //TO DO:
    }
}
```

```
//Derived Class
class F : B
{
    public void fooF()
    {
        //TO DO:
    }
}
//Derived Class
class G : B
{
    public void fooG()
    {
        //TO DO:
    }
}
```

Inheritance

- **Hybrid Inheritance -** Hybrid inheritance is a combination of two or more types of inheritance. The combination of **multilevel** and **hierarchical** inheritance is an example of Hybrid inheritance.



Inheritance

- **Method Overriding -** If the same method is present in both the base class and the derived class, the method in the derived class overrides the method in the base class. This is called method overriding in C#.

```
using System;
namespace Inheritance {

class Animal {
    public virtual void eat() {
      Console.WriteLine("I eat food");
    }
  }

  // derived class of Animal
  class Dog : Animal {

    // overriding method from Animal
    public override void eat() {
      Console.WriteLine("I eat Dog food");
    }
  }
}
```

```
class Program {

    static void Main(string[] args) {
      // object of derived class
      Dog d = new Dog();

      // accesses overridden method
      d.eat();
    }
  }
}
```

Inheritance

- **base Keyword in C# Inheritance -** what if we want to call the method of the base class as well? In that case, we use the **base** keyword to call the method of the base class from the derived class.

```
using System;
namespace Inheritance {

class Animal {
    public virtual void eat() {
      Console.WriteLine("I eat food");
    }
  }
    // derived class of Animal
  class Dog : Animal {

    // overriding method from Animal
    public override void eat() {
      // call method from Animal class
      base.eat();
      Console.WriteLine("I eat Dog food");
    }
  }
}
```

```
class Program {

    static void Main(string[] args) {
      // object of derived class
      Dog d = new Dog();

      // accesses overridden method
      d.eat();
    }
  }
}
```

Sealed Class and Abstract Class

- **Sealed Class and Method -** In C#, when we **don't want a class to be inherited by another class**, we can declare the class as a **sealed class**.
- A sealed class **cannot have a derived class**. We use the sealed keyword to create a sealed class. For example,

```
using System;
namespace SealedClass
{
  sealed class Math
  {
      ---------
      ---------
  }
  // trying to inherit sealed class  Error Code
  class MyMath : Math
  {

  }
```

Sealed Class and Abstract Class

- **Sealed Class and Method -** During method overriding, if we don't want an a method to be overridden by another class, we can declare it as a **sealed** method.
- We use a **sealed** keyword with an overridden method to create a sealed method. For example,

```
using System;
namespace SealedClass
{
  class Abc
  {
      sealed public void f1()
      {
        ----------
        ----------
      }

  }
```

```
class Xyz : Abc
{
    // trying to override the f1() method. Will give compile
    // time error.
   public override void f1()
   {
     ------------
     ------------
   }
}
```

Sealed Class and Abstract Class

- **Abstract Class and Method -** In C#, we **cannot create objects of an abstract class**. We use the abstract keyword to create an abstract class.
- An abstract class can have both abstract methods (method without body) and non-abstract methods (method with the body). For example,

```
  abstract class Language {

  // abstract method
  public abstract void display1();

  // non-abstract method
  public void display2() {
    Console.WriteLine("Non abstract method");
  }
}
// try to create an object Language
// throws an error
Language obj = new Language();
```

Sealed Class and Abstract Class

- **Abstract Class and Method -** A method that does not have a body is known as an abstract method. We use the abstract keyword to create abstract methods.
- When a non-abstract class inherits an abstract class, it should provide an implementation of the abstract methods.

```
abstract class Language
{

   // abstract method
   public abstract void display1();

   // non-abstract method
   public void display2()
   {
      Console.WriteLine("Non abstract
                                  method");
   }
}
```

```
class Gujarati
{

       // abstract method override
       public override void display1()
       {
                ----------
                -----------
       }
}
```

Sealed Class and Abstract Class

- **Abstract Class and Method**
- We can use abstract class only as a base class. This is why abstract classes cannot be sealed.
- If the base class had not provided the implementation of the abstract method , base class should have been marked abstract as well.

Interface

- In C#, an interface is similar to abstract class. However, unlike abstract classes, all methods of an interface are fully abstract (method without body).
- We use the interface keyword to create an interface. For example,

```
interface IPolygon
{

    // method without body
    void calculateArea();
}
```

- Here,
  - IPolygon is the name of the interface.
  - By convention, interface starts with I so that we can identify it just by seeing its name.
  - We cannot use access modifiers inside an interface.
  - All members of an interface are public by default.
  - An interface doesn't allow fields.

Interface

- We cannot create objects of an interface. To use an interface, other classes must implement it. Same as in **C# Inheritance**, **we use : symbol** to implement an interface. For example,

```
interface Ipolygon
{
        void calculateArea(int l, int b);
}

class Rectangle : IPolygon
{
        public void calculateArea(int l, int b)
        {
              int area = l * b;
              Console.WriteLine("Area of Rectangle: "
                                                + area);
        }
  }
```

```
class Program {
    static void Main (string [] args) {

      Rectangle r1 = new Rectangle();

      r1.calculateArea(100, 200);

    }
  }
```

Note: We must provide the implementation of all the methods of interface inside the class that implements it.

Interface

- Unlike inheritance, a class can implement multiple interfaces. For example,

```
interface IPolygon
{
        void calculateArea(int l, int b);
}
interface IColor
{
    void getColor();
}
class Rectangle : IPolygon, IColor
{
        public void calculateArea(int l, int b)
        {
              int area = l * b;
              Console.WriteLine("Area of Rectangle: "
                                             + area);
        }
```

```
        public void getColor()
        {
                Console.WriteLine("Red
                        Rectangle");
        }
}
class Program
{
    static void Main (string [] args)
    {

      Rectangle r1 = new Rectangle();

      r1.calculateArea(100, 200);
      r1.getColor();

    }
}
```

Interface

- We can use the reference variable of an interface. For example,

```
interface Ipolygon
{
        void calculateArea(int l, int b);
}

class Rectangle : IPolygon
{
        public void calculateArea(int l, int b)
        {
              int area = l * b;
              Console.WriteLine("Area of Rectangle: "
                                             + area);
        }
  }
```

```
class Program {
    static void Main (string [] args) {

      IPolygon r1 = new Rectangle();

      r1.calculateArea(100, 200);

    }
  }
```

Note: Though we cannot create objects of an interface, we can still use the reference variable of the interface that points to its implemented class.

Interface

- **Advantages of C# interface**
- Similar to abstract classes, interfaces help us to achieve abstraction in C#.
- Interfaces provide specifications that a class (which implements it) must follow.
- Interfaces are used to achieve multiple inheritance in C#.
- Interfaces provide loose coupling(having no or least effect on other parts of code when we change one part of a code).

Collections

- We mostly use arrays to store and manage data.
- However, arrays can store a particular type of data, such as integer and characters.
- To resolve this issue, the .NET framework introduces the concept of collections for data storage and retrieval, where collection means a group of different types of data.
- Collections are similar to arrays, it provide a more flexible way of working with a group of objects.
- In arrays, you need to define the number of elements at declaration time.
- But in a collection, you don't need to define the size of the collection in advance. You can add elements or even remove elements from the collection at any point of time.

Collections

- The .NET Framework provides you a variety of collection classes, which are available in the System.Collections namespace.

- Most useful collection classes defined are as follows.
  - ArrayList Class
  - Stack Class
  - Queue Class
  - Hashtable Class
  - SortedList Class

Collections

- **ArrayList Class**

- Using an arrays, the main problem arises is that their size always remain fixed by the number that you specify when declaring an arrays.
- In addition, you cannot add items in the middle of array.
- Also you can not deal with different data types.
- The solution to these problems is the use of ArrayList Class.
- Similar to an array, the ArrayList class allows you to store and manage multiple elements.
- With the ArrayList class, you can add and remove items from a list of array items from a specified position, and then the array items list automatically resizes itself accordingly.

Collections

- **Properties of ArrayList Class**

Property	Description
Capacity	Gets or sets the number of elements that the ArrayList can contain.
Count	Gets the number of elements actually contained in the ArrayList.
IsFixedSize	Gets a value indicating whether the ArrayList has a fixed size.
IsReadOnly	Gets a value indicating whether the ArrayList is read-only.
Item	Gets or sets the element at the specified index.

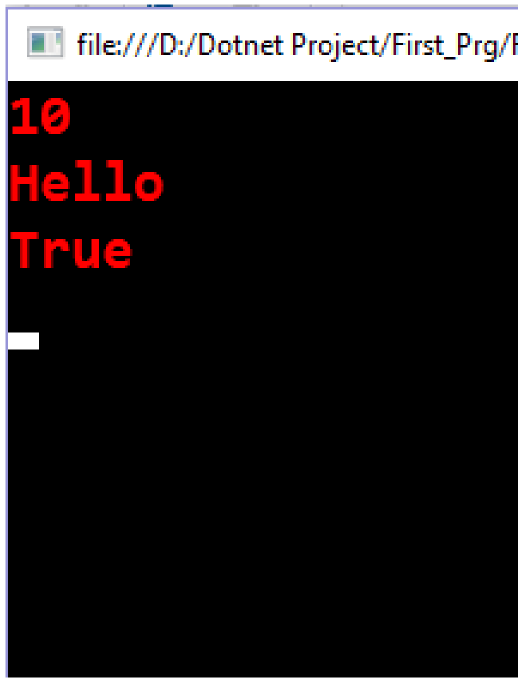
Collections

- **Methods of ArrayList Class**

Sr.	Description
(1)	**public virtual int Add(object value);** Adds an object to the end of the ArrayList.
(2)	**public virtual void Clear();** Removes all elements from the ArrayList.
(3)	**public virtual bool Contains(object item);** Determines whether an element is in the ArrayList.
(4)	**public virtual void Remove(object obj);** Removes the first occurrence of a specific object from the ArrayList.
(5)	**public virtual void RemoveAt(int index);** Removes the element at the specified index of the ArrayList.

Collections

```
using System;
using System.Collections;

namespace First_Prg
{
    class MainClass
    {
        public static void Main(string[] args)
        {
            ArrayList A1 = new ArrayList();

            A1.Add(10);
            A1.Add("Hello");
            A1.Add(true);

            Console.WriteLine(A1[0]);
            Console.WriteLine(A1[1]);
            Console.WriteLine(A1[2]);

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

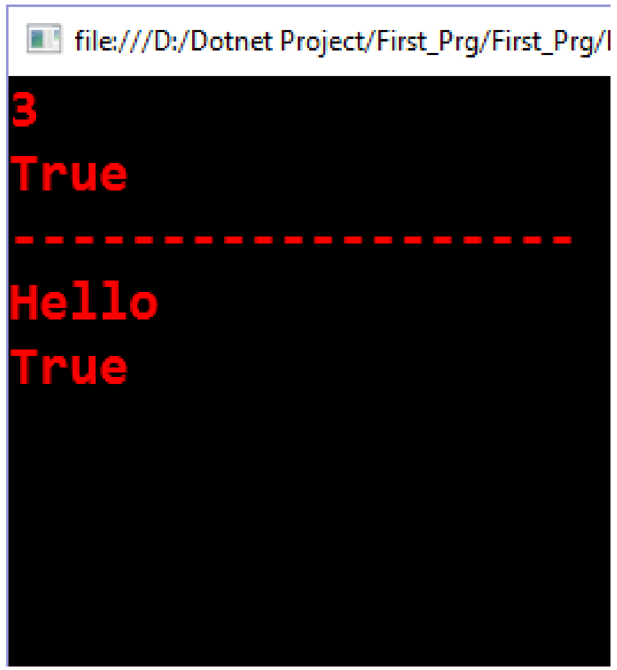


file:///D:/Dotnet Project/First_Prg/F

```
10
Hello
True
```

Collections

```
using System;
using System.Collections;
namespace First_Prg
{
    class MainClass
    {
        public static void Main(string[] args)
        {
            ArrayList A1 = new ArrayList();

            A1.Add(10);
            A1.Add("Hello");
            A1.Add(true);

            Console.WriteLine(A1.Count);
            Console.WriteLine(A1.Contains(10));
            Console.WriteLine("-------------------");
            Console.WriteLine(A1[1]);
            A1.RemoveAt(1);
            Console.WriteLine(A1[1]);
        }
    }
```


file:///D:/Dotnet Project/First_Prg/First_Prg/

```
3
True
-------------------
Hello
True
```

Collections

- **Stack Class**

- The stack class represents a last in first out (LIFO) concept.
- let's take an example. Imagine a stack of books with each book kept on top of each other.
- The concept of last in first out in the case of books means that only the top most book can be removed from the stack of books.
- It is not possible to remove a book from between, because then that would disturb the setting of the stack. Hence in C#, the stack also works in the same way.
- Elements are added to the stack, one on the top of each other.
- The process of adding an element to the stack is called a **push** operation.
- The process of removing an element from the stack is called a **pop** operation.

Collections

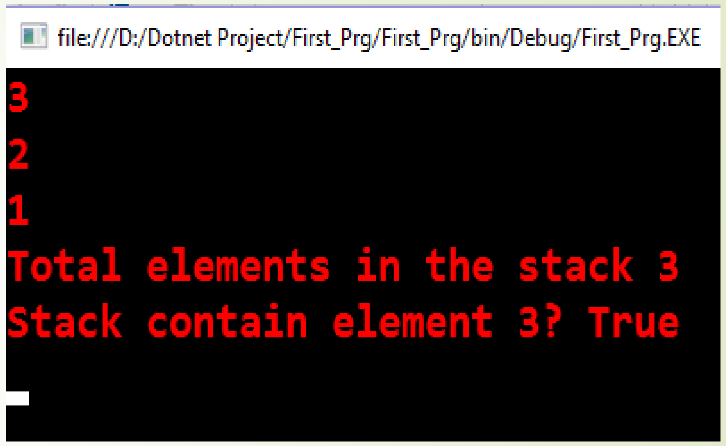
- **Properties & Methods of Stack Class**

Property	Description
Count	Gets the number of elements contained in the Stack.

Sr.	Description
(1)	**public virtual void Push(object obj);** Inserts an object at the top of the Stack.
(2)	**public virtual object Pop();** Removes and returns the object at the top of the Stack.
(3)	**public virtual bool Contains(object obj);** Determines whether an element is in the Stack.
(4)	**public virtual void Clear();** Removes all elements from the Stack.

Collections

```
using System;
using System.Collections;

namespace First_Prg
{
    class MainClass
    {
        public static void Main(string[] args)
        {
            Stack st = new Stack();
            st.Push(1);
            st.Push(2);
            st.Push(3);

            foreach (Object obj in st)
            {
                Console.WriteLine(obj);
            }
            Console.WriteLine("Total elements in the stack " + st.Count);
            Console.WriteLine("Stack contain element 3? " + st.Contains(3));
        }
    }
}
```


file:///D:/Dotnet Project/First_Prg/First_Prg/bin/Debug/First_Prg.EXE

```
3
2
1
Total elements in the stack 3
Stack contain element 3? True
```

Collections

- **Queue Class**

- The **Queue** class represents a first in first out (FIFO) concept.
- let's take an example. Imagine a queue of people waiting for the bus.
- Normally, the first person who enters in the queue will be the first person to enter into the bus.
- Similarly, the last person to enter in the queue will be the last person to enter into the bus.
- The process of adding an element to the queue is the **Enqueue** operation.
- The process of removing an element from the queue is the **Dequeue** operation.

Collections

- **Properties and Method**

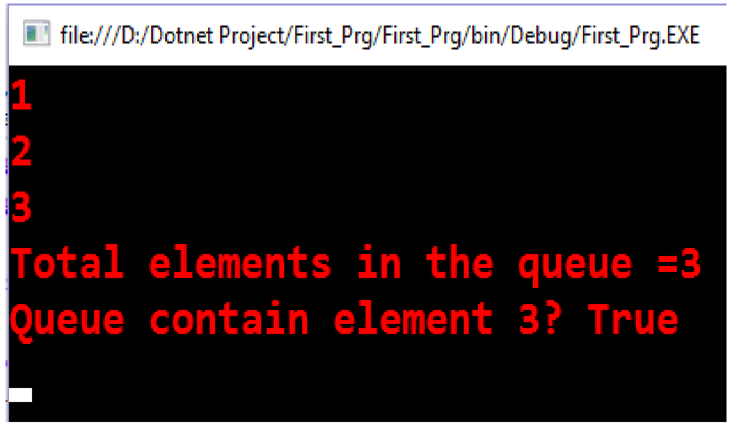
Property	Description
Count	Gets the number of elements contained in the Queue.

Sr.	Description
(1)	**public virtual void Enqueue(object obj);** Adds an object to the end of the Queue.
(2)	**public virtual object Dequeue();** Removes and returns the object at the beginning of the Queue.
(3)	**public virtual bool Contains(object obj);** Determines whether an element is in the Queue.
(4)	**public virtual void Clear();** Removes all elements from the Queue.

Collections

```
using System;
using System.Collections;

namespace First_Prg
{
    class MainClass
    {
        public static void Main(string[] args)
        {
            Queue qt = new Queue();
            qt.Enqueue(1);
            qt.Enqueue(2);
            qt.Enqueue(3);

            foreach (Object obj in qt)
            {
                Console.WriteLine(obj);
            }
            Console.WriteLine("Total elements in the queue =" + qt.Count);
            Console.WriteLine("Queue contain element 3? " + qt.Contains(3));
         }
    }
}
```

Output :


file:///D:/Dotnet Project/First_Prg/First_Prg/bin/Debug/First_Prg.EXE

```
1
2
3
Total elements in the queue =3
Queue contain element 3? True
```

Collections

- **Hashtable Class**

- The Hashtable class is similar to the ArrayList class, but it allows you to access the items by a key.
- Each item in a Hashtable object has a key/value pair.
- So instead of storing just one value like the stack, array list and queue, the Hashtable stores 2 values that is key and value.
- The key is used to access the items in a collection and each key must be unique, whereas the value is stored in an array.
- The keys are short strings or integers.

Collections

- **Hashtable Class**

- There is no direct way to display the elements of a Hashtable.
  - { "1" , “NFSU" }
  - { "2" , “IIT Gandhinagar" }
  - { "3" , “NIT Surat" }

- In order to display the Hashtable , we first need to get the list of keys (1, 2 and 3) from the hash table.
- This is done via the ICollection interface.
- This is a special data type which can be used to store the keys of a Hashtable collections.

Collections

- **Properties of Hashtable Class**

Property	Description
Count	Gets the number of key-and-value pairs contained in the Hashtable.
IsFixedSize	Gets a value indicating whether the Hashtable has a fixed size.
IsReadOnly	Gets a value indicating whether the Hashtable is read-only.
Item	Gets or sets the value associated with the specified key.
Keys	Gets an ICollection containing the keys in the Hashtable.

Collections

- **Methods of Hashtable Class**

Sr.	Description
(1)	**public virtual void Add(object key, object value);** Adds an element with the specified key and value into the Hashtable.
(2)	**public virtual void Clear();** Removes all elements from the Hashtable.
(3)	**public virtual bool ContainsKey(object key);** Determines whether the Hashtable contains a specific key.
(4)	**public virtual bool ContainsValue(object value);** Determines whether the Hashtable contains a specific value.
(5)	**public virtual void Remove(object key);** Removes the element with the specified key from the Hashtable.

Collections

```
using System;
using System.Collections;

namespace First_Prg
{
    class MainClass
    {
        public static void Main(string[] args)
        {
            Hashtable ht = new Hashtable();

            ht.Add("1",“NFSU");
            ht.Add("2", “IIT Gandhinagar");
            ht.Add("3", “NIT Surat");

            ICollection keys = ht.Keys;

            foreach (String k in keys)
            {
                Console.WriteLine(ht[k]);
            }
        }
```

Output :

Collections

- **SortedList Class**
- The SortedList class is a combination of the array and hashtable classes.
- It contains a list of items that can be accessed by using index or a key.
- When you access items using indexes, the SortedList objects acts as an ArrayList object, whereas when you access items using keys, it acts as a Hashtable object.
- The striking feature about the SortedList class is the collection of items are always sorted by the key value.

QUESTIONS?
COMMENTS?
CONCERNS?