Syllogistek
Systems Private Limited

502 SreeNilayam, Lanco Hills Road,
Manikonda, Hyderabad – 500089
08413485772 (landline) / 9701377070
Email : hr@syllogistek.com

Contents

Object-oriented programming (OOP) is the core ingredient of the .NET framework. OOP is so important that, before embarking on the road to .NET, you must understand its basic principles and terminology to write even a simple program. The fundamental idea behind OOP is to combine into a single unit both data and the methods that operate on that data; such units are called an object. All OOP languages provide mechanisms that help you implement the object-oriented model. They are encapsulation, inheritance, polymorphism and reusability. Let's now take a brief look at these concepts.

This chapter covers the following:

1. OOP's overview
2. Classes and Objects
3. Constructor and Destructor
4. Function Overloading
5. Encapsulation
6. Inheritance
7. Interface
8. Polymorphism

Encapsulation

Encapsulation binds together code and the data it manipulates and keeps them both safe from outside interference and misuse. Encapsulation is a protective container that prevents code and data from being accessed by other code defined outside the container.

Inheritance

Inheritance is the process by which one object acquires the properties of another object. A type derives from a base type, taking all the base type members fields and functions. Inheritance is most useful when you need to add functionality to an existing type. For example all .NET classes inherit from the System.Object class, so a class can include new functionality as well as use the existing object's class functions and properties as well.

Polymorphism

Polymorphism is a feature that allows one interface to be used for a general class of action. This concept is often expressed as "one interface, multiple actions". The specific action is determined by the exact nature of circumstances.

Reusability

Once a class has been written, created and debugged, it can be distributed to other programmers for use in their own program. This is called reusability, or in .NET terminology this concept is called a component or a DLL. In OOP, however, inheritance provides an important extension to the idea of reusability. A programmer can use an existing class and without modifying it, add additional features to it.

Simple "Hello World" C# Program

This simple one-class console "Hello world" program demonstrates many fundamental concepts throughout this article and several future articles.

```
using System;
namespace oops
{
   //class definition
   public class SimpleHelloWorld
   {
       //Entry point of the program
      static void Main(string[] args)
      {
         //print Hello world"
         Console.WriteLine("Hello World!");
      }
   }
}
```

So SimpleHelloWorld is the name of the class that contains the Main () method. On line 1 , a using directive indicates to the compiler that this source file refers to classes and constructs declared within the System namespace. Line 6 with the public keyword indicates the program accessibility scope for other applications or components.

At line 7 there appears an opening curly brace ("{") which indicates the beginning of the SimpleHelloWorld class body. Everything belongs to the class, like fields, properties and methods appear in the class body between the opening and closing braces. The purpose of the Main () method is to provide an entry point for application execution.

The static keyword in the Main () method states that this method would be executed without instantiating the class.

Compiling the Program

You can compile a C# program into either an assembly or a module. If the program has one class that contains a Main () method then it can be compiled directly into an assembly. This file has an ".exe" extension. A program with no Main() method can be compiled into a module as in the following:

csc /target:module "program name"

You can then compile this program by F9 or by simply running the C# command line compiler (csc.exe) against the source file as the following:

cscoops.cs

Classes and Objects

Classes are special kinds of templates from which you can create objects. Each object contains data and methods to manipulate and access that data. The class defines the data and the functionality that each object of that class can contain.

A class declaration consists of a class header and body. The class header includes attributes, modifiers, and the class keyword. The class body encapsulates the members of the class that are the data members and member functions. The syntax of a class declaration is as follows:

Attributes accessibility modifiers class identifier: *baselist{ body }*

Attributes provide additional context to a class, like adjectives; for example the Serializable attribute. Accessibility is the visibility of the class. The default accessibility of a class is internal. Private is the default accessibility of class members. The following table lists the accessibility keywords:

Keyword	Description
Public	Public class is visible in the current and referencing assembly.
Private	Visible inside current class.
Protected	Visible inside current and derived class.
Internal	Visible inside containing assembly.
Internal protected	Visible inside containing assembly and descendent of thecurrent class.

Modifiers refine the declaration of a class. The list of all modifiers defined in the table is as follows:

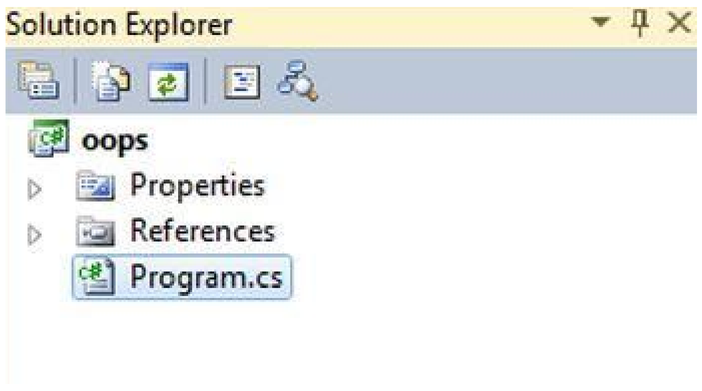
Modifier	Description
sealed	Class can't be inherited by a derived class.
static	Class contains only static members.
unsafe	The class that has some unsafe construct likes pointers.

Abstract	The instance of the class is not created if the Class is abstract.

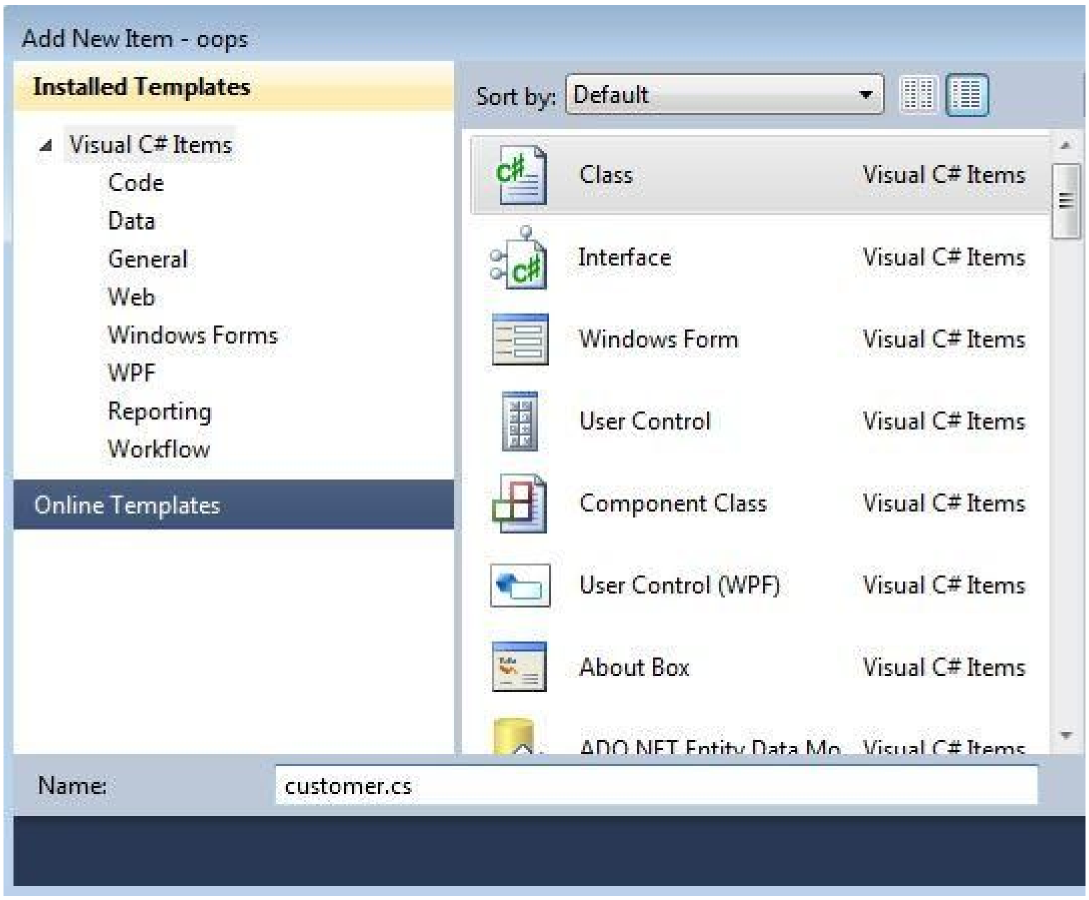
The baselist is the inherited class. By default, classes inherit from the System.Object type. A class can inherit and implement multiple interfaces but doesn't support multiple inheritances.

Step-by-step Tutorial for Creating a Class

1. Open Visual Studio 2010 from start menu.
2. Go to "File" > "New" > "Project...", select "Console Application" in the right pane and provide the name "oops" for the project.
3. Then in the Solution Explorer, you will notice some files that are automatically created as,



4. You can also write your own code in the default program.cs file that is created but it is a good programming practice to create a new class.
5. For adding a new class, right-click over the project name (oops) in the Solution Explorer, then click "Add" > "Class". Give the name to the class "customer" as in the following;


Add New Item - oops
Installed Templates
Visual C# Items
Code
Data
General
Web
Windows Forms
WPF
Reporting
Workflow
Online Templates
Sort by: Default
Class Visual C# Items
Interface Visual C# Items
Windows Form Visual C# Items
User Control Visual C# Items
Component Class Visual C# Items
User Control (WPF) Visual C# Items
About Box Visual C# Items
ADO.NET Entity Data Mo... Visual C# Items
Name: customer.cs

When you open the customer.cs class.you will find some default-generated code as in the following,

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace oops
{
   class customer
   {
   }
}
```

Note

The C# console application project must require a single entry point Main () function that is already generated in the program class. For example if you add a new customer class and want to define one or more Main () entry points here then .NET will throw an error of multiple entry points. So it is advisable to delete or exclude the program.cs file from the solution.

So here in this example the customer class defines fields such as CustID, Name and Address to hold information about a particular customer. It might also define some functionality that acts upon the data stored in these fields.

```
using System;
    namespace oops
    {
       class customer
       {
          // Member Variables
          public int CustID;
          public string Name;
          public string Address;

    //constuctor for initializing fields
          customer()
          {
             CustID=1101;
             Name="Tom";
             Address="USA";
          }


          //method for displaying customer records (functionality)
          public void displayData()
          {
             Console.WriteLine("Customer="+CustID);
             Console.WriteLine("Name="+Name);
             Console.WriteLine("Address="+Address);
          }
         // Code for entry point
       }
    }
```

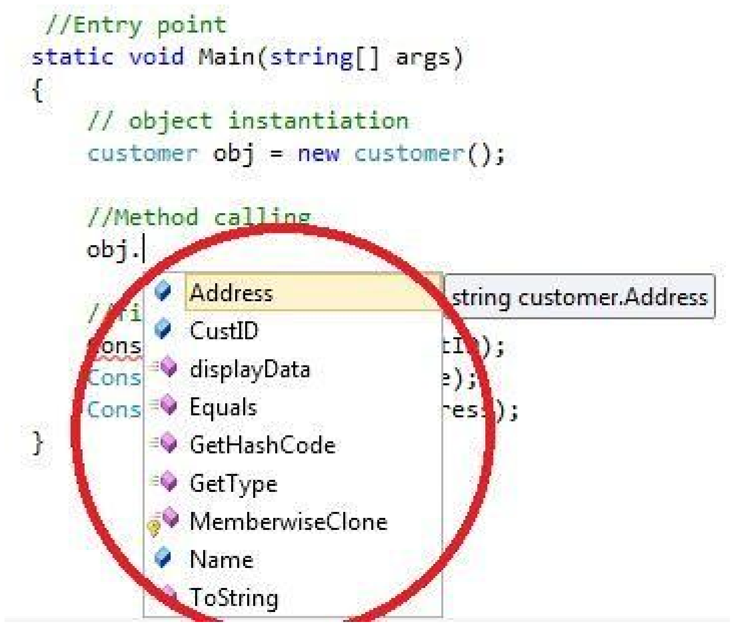
At line 9, we are defining a constructor of the customer class for initializing the class member fields. The constructor is a special function that is automatically called when the customer class object is created (instantiated). And at line 11 we are printing these fields to the console by creating a user defined method displayData().

You can then instantiate an object of this class to represent one specific customer, set the field value for that instance and use its functionality, as in:

```
class customer
    {
       // class members code
        //Entry point
       static void Main(string[] args)
       {
          // object instantiation
          customer obj = new customer();
          //Method calling
          obj.displayData();
           //fields calling
```

```
            Console.WriteLine(obj.CustID);
            Console.WriteLine(obj.Name);
            Console.WriteLine(obj.Address);
        }
    }
```

Here you use the keyword new to declare the customer class instance. This keyword creates the object and initializes it. When you create an object of the customer class, the .NET framework IDE provides a special feature called Intellisense that provides access to all the class member fields and functions automatically. This feature is invoked when the "." operator is put right after the object, as in the following;



```
 //Entry point
static void Main(string[] args)
{
    // object instantiation
    customer obj = new customer();

    //Method calling
    obj.
```

Image 1.1 Intellisense feature

Normally, as the program grows in size and the code becomes more complex, the Intellisense feature increases the convenience for the programmer by showing all member fields, properties and functions.

Multiple Class Declaration

Sometimes circumstances require multiple classes to be declared in a single namespace. So in that case it is not mandatory to add a separate class to the solution, instead you can attach the new class into the existing program.cs or another one as in the following;

```
using System;
    namespace oops
    {
       class Program
       {
           public void MainFunction()
           {
             Console.WriteLine("Main class");
```

```
        }
        static void Main(string[] args)
       {
           //main class instance
           Program obj = new Program();
           obj.MainFunction();
           //other class instace
           demo dObj=new demo();
           dObj.addition();
        }
     }

  class demo
     {
        int x = 10;
        int y = 20;
        int z;

    public void addition()
        {
           z = x + y;
           Console.WriteLine("other class in Namespace");
           Console.WriteLine(z);
        }
     }
  }
```

Here in this example, we are creating an extra class "demo" in the program.cs file at line 12 and finally we are instantiating the demo class with the program class inside the Main() entry in lines 6 to 11. So it doesn't matter how many classes we are defining in a single assembly.

Partial classes

Typically, a class will reside entirely in a single file. However, in situations where multiple developers need access to the same class, then having the class in multiple files can be beneficial. The partial keywords allow a class to span multiple source files. When compiled, the elements of the partial types are combined into a single assembly.

There are some rules for defining a partial class as in the following;

- A partial type must have the same accessibility.
- Each partial type is preceded with the "partial" keyword.
- If the partial type is sealed or abstract then the entire class will be sealed and abstract.

In the following example we are adding two files, partialPart1.cs and partialPart2.cs, and declare a partial class, partialclassDemo, in both classes.

partialPart1.cs

```
using System;
    namespace oops
    {
        public partial class partialclassDemo
        {
            public void method1()
            {
                Console.WriteLine("method from part1 class");
            }
        }
    }
```

partialPart2.cs

```
using System;
    namespace oops
    {
        public partial class partialclassDemo
        {
            public void method2()
            {
                Console.WriteLine("method from part2 class");
            }
        }
    }
```

And finally we are creating an instance of the partialclassDemo in the program.cs file as the following:

Program.cs

```
using System;
    namespace oops
    {
        class Program
        {
            static void Main(string[] args)
            {
                //partial class instance
                partialclassDemo obj = new partialclassDemo();
                obj.method1();
                obj.method2();
            }
        }
    }
```

Static classes

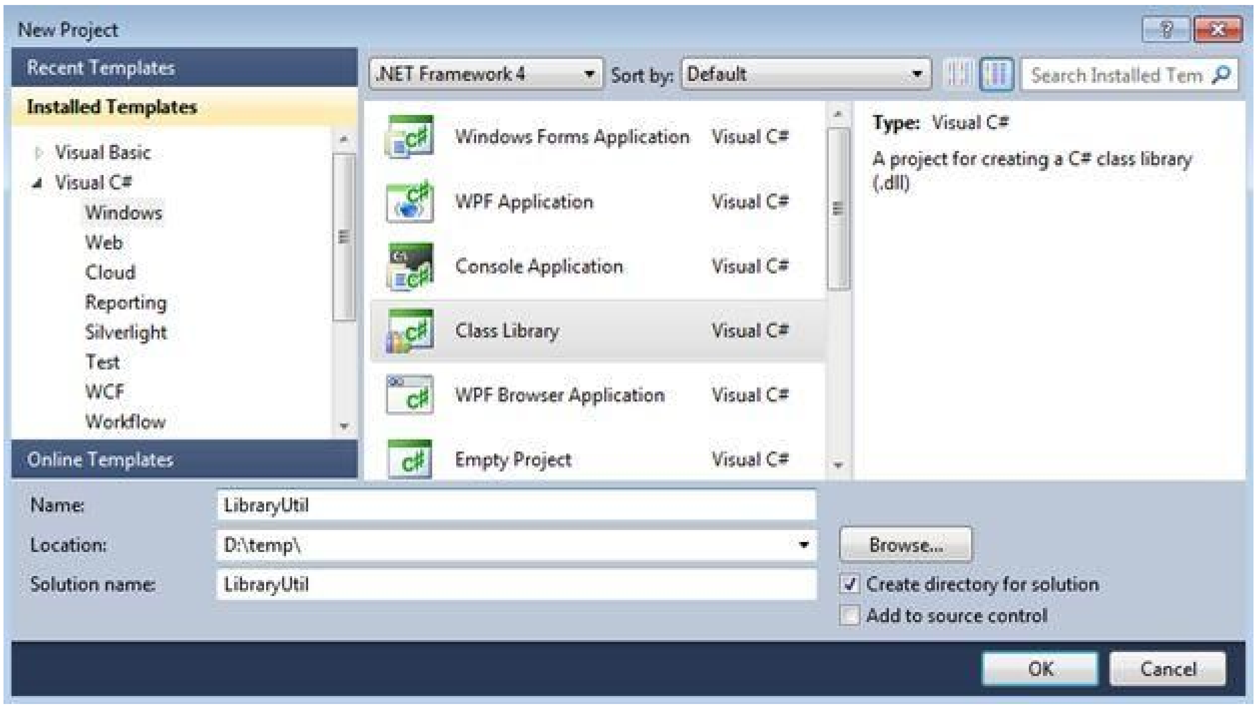
A static class is declared using the "static" keyword. If the class is declared as static then the compiler never creates an instance of the class. All the member fields, properties and functions must be declared as static and they are accessed by the class name directly not by a class instance object.

```
using System;
    namespace oops
    {
        static class staticDemo
        {
            //static fields
            static int x = 10, y;
            //static method
            static void calcute()
            {
                y = x * x;
                Console.WriteLine(y);
            }
            static void Main(string[] args)
            {
                //function calling directly
                staticDemo.calcute();
            }
        }
    }
```

Creating and accessing Class Component Library

.NET provides the capability of creating libraries (components) of a base application rather than an executable (".exe"). Instead the library project's final build version will be ".DLL" that can be referenced from other outside applications to expose its entire functionality.

First create a class library based application as,


New Project

Recent Templates

Installed Templates

- Visual Basic
- Visual C#
  - Windows
  - Web
  - Cloud
  - Reporting
  - Silverlight
  - Test
  - WCF
  - Workflow

Online Templates

.NET Framework 4 — Sort by: Default — Search Installed Tem

Windows Forms Application	Visual C#
WPF Application	Visual C#
Console Application	Visual C#
Class Library	Visual C#
WPF Browser Application	Visual C#
Empty Project	Visual C#

Type: Visual C#

A project for creating a C# class library (.dll)

Name: LibraryUtil

Location: D:\temp\

Solution name: LibraryUtil

[✓] Create directory for solution

[ ] Add to source control

Then we are implementing a math class library that is responsible of calculating square root and the addition of two numbers as:

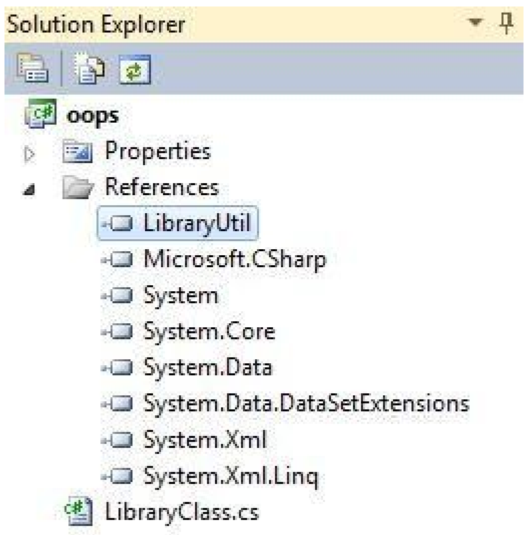
```
using System;
    namespace LibraryUtil
    {
       public class MathLib
       {
          public MathLib() { }
          public void calculareSum(int x, int y)
          {
             int z = x + y;
             Console.WriteLine(z);
          }

     public void calculareSqrt(double x)
          {
             double z = Math.Sqrt(x);
             Console.WriteLine(z);
          }
       }
    }
```

Build this code and you will notice that a DLL file was created, not an executable, in the root directory of the application *(path = D:\temp\LibraryUtil\LibraryUtil\bin\Debug\ LibraryUtil.dll).*

Now create another console based application where you utilize all the class library's functionality.

Then you have to add the class library dll file reference to access the declared class in the library dll. (Right-click on the Reference then "Add reference" then select the path of the dll file.)

When you add the class library reference then you will notice in the Solution Explorer that a new LibraryUtil is added as in the following;



Now add the namespace of the class library file in the console application and create the instance of the class declared in the library as in the following;

```
using System;
    using LibraryUtil; // add library namespace
    namespace oops
    {
        public class LibraryClass
        {
            static void Main()
            {
                //library class instance
                MathLib obj = new MathLib();
                //method populate
                obj.calculareSum(2, 5);
                obj.calculareSqrt(25);
            }
        }
    }
```

Finally run the application.

Constructor and Destructor

A constructor is a specialized function that is used to initialize fields. A constructor has the same name as the class. Instance constructors are invoked with the new operator and can't be called in the same manner as other member functions. There are some important rules pertaining to constructors as in the following;

- Classes with no constructor have an implicit constructor called the default constructor that is parameter less. The default constructor assigns default values to fields.
- A public constructor allows an object to be created in the current assembly or referencing assembly.
- Only the extern modifier is permitted on the constructor.
- A constructor returns void but does not have an explicitly declared return type.
- A constructor can have zero or more parameters.
- Classes can have multiple constructors in the form of default, parameter or both.

The following example shows one constructor for a customer class.

```
using System;
    namespace oops
    {
        class customer
        {
            // Member Variables
            public string Name;
            //constuctor for initializing fields
            public customer(string fname, string lname)
            {
                Name= fname +" "+ lname;
            }
            //method for displaying customer records
            public void AppendData()
            {
                Console.WriteLine(Name);
            }
             //Entry point
            static void Main(string[] args)
            {
                // object instantiation
                customer obj = new customer("Barack", "Obama");
                //Method calling
                obj.AppendData();
            }
        }
    }
```

Note

The moment a new statement is executed, the default constructor is called.

Static Constructor

A constructor can be static. You create a static constructor to initialize static fields. Static constructors are not called explicitly with the new statement. They are called when the class is first referenced. There are some limitations of the static constructor as in the following;

- Static constructors are parameter less.
- Static constructors can't be overloaded.
- There is no accessibility specified for Static constructors.

In the following example the customer class has a static constructor that initializes the static field and this constructor is called when the class is referenced in the Main () at line 26 as in the following:

```
using System;
    namespace oops
    {
        class customer
        {
            // Member Variables
            static private int x;
            //constuctor for static initializing fields
            static customer()
            {
                x = 10;
            }

      //method for get  static field
            static public void getData()
            {
                Console.WriteLine(x);
            }
             //Entry point
            static void Main(string[] args)
            {
               //static Method calling
                customer.getData();
            }
        }
    }
```

Destructors

The purpose of the destructor method is to remove unused objects and resources. Destructors are not called directly in the source code but during garbage collection. Garbage collection is nondeterministic. A destructor is invoked at an undetermined moment. More precisely a programmer can't control its execution; rather it is called by the Finalize () method. Like a constructor, the destructor has the same name as the class except a destructor is prefixed with a tilde (~). There are some limitations of destructors as in the following;

- Destructors are parameter less.
- A Destructor can't be overloaded.
- Destructors are not inherited.
- Destructors can cause performance and efficiency implications.

The following implements a destructor and dispose method. First of all we are initializing the fields via constructor, doing some calculations on that data and displaying it to the console. But at line 9 we are implementing the destructor that is calling a Dispose() method to release all the resources.

```
using System;
    namespace oops
    {
        class customer
        {
            // Member Variables
            public int x, y;
            //constuctor for  initializing fields
            customer()
            {
                Console.WriteLine("Fields inititalized");
                x = 10;
            }
            //method for get field
            public void getData()
            {
                y = x * x;
                Console.WriteLine(y);
            }
            //method to release resource explicitly
            public void Dispose()
            {
                Console.WriteLine("Fields cleaned");
                x = 0;
                y = 0;
            }
            //destructor
            ~customer()
            {
                Dispose();
            }
            //Entry point
            static void Main(string[] args)
            {
                //instance created
                customer obj = new customer();
                obj.getData();
            }
        }
    }
```

At line 12 when the instance is created, fields are initialized but it is not necessary that at the same time the destructor is also called. Its calling is dependent on garbage collection. If you want to see the destructor being called into action then put a breakpoint (by F9) at line 10 and compile the application. The CLR indicates its execution at the end of the program by highlighting line 10 using the yellow color.

Function Overloading

Function overloading allows multiple implementations of the same function in a class. Overloaded methods share the same name but have a unique signature. The number of parameters, types of parameters or both must be different. A function can't be overloaded on the basis of a different return type alone.

```
using System;
    namespace oops
    {
       class funOverload
       {
          public string name;
          //overloaded functions
          public void setName(string last)
          {
             name = last;
          }

      public void setName(string first, string last)
          {
             name = first + "" + last;
          }

      public void setName(string first, string middle, string last)
          {
             name = first + "" + middle + "" + last;
          }

      //Entry point
          static void Main(string[] args)
          {
             funOverload obj = new funOverload();
             obj.setName("barack");
             obj.setName("barack "," obama ");
             obj.setName("barack ","hussian","obama");
          }
       }
    }
```

At lines 3, 4 and 5 we are defining three methods with the same name but with different parameters. In the Main (), the moment you create an instance of the class and call the functions setName() via obj at lines 7, 8 and 9 then intellisense will show three signatures automatically.

Encapsulation

Encapsulation is the mechanism that binds together the code and the data it manipulates, and keeps both safe from outside interference and misuse. In OOP, code and data may be combined in such a way that a self-contained box is created. When code and data are linked together in this way, an object is created and encapsulation exists.

Within an object, code, data or both may be private or public to that object. Private code is known to and accessible only by another part of the object, that is private code or data may not be accessible by a piece of the program that exists outside the object. When the code and data is public, other portions of your program may access it even though it is defined within an object.

```
using System;
    namespace oops
    {
        class Encapsulation
        {
            /// <summary>
            /// Every member Variable and Function of the class are bind
            /// with the Encapsulation class object only and safe with
            /// the outside inference
            /// </summary>
            // Encapsulation Begin
            int x;
            //class constructor
            public Encapsulation(int iX)
            {
                this.x = iX;
            }


            //calculating the square
            public void MySquare()
            {
                int Calc = x * x;
                Console.WriteLine(Calc);
            }

        // End of Encapsulation
              //Entry point
            static void Main(string[] args)
            {
                //instance created
                customer obj = new customer(20);
                obj. MySquare();
            }
        }
    }
```

Inheritance

Inheritance is the process by which one object can acquire the properties of another object. Inheritance is a "is a kind of" relationship and it supports the concept of classification in which an object needs only define those qualities that make it unique within the class. Inheritance involves a base class and a derived class. The derived class inherits from the base class and also can override inherited members as well as add new members to extend the base class.

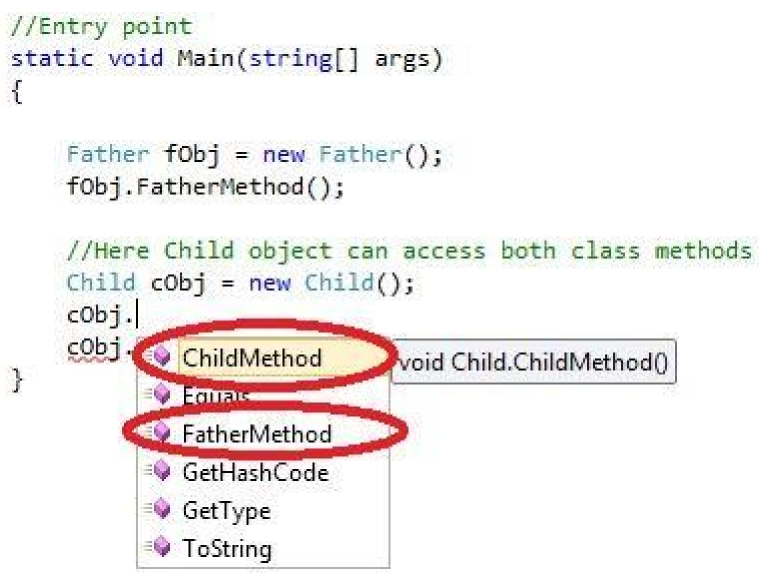
A base type represents the generalization, whereas a derived type represents a specification of an instance. Such as Employees that can have diverse types, such as hourly, salaried and temporary so in that case Employees is the general base class and hourly, salaried and temporary employee are specialized derived classes.

Classes can inherit from a single class and one or more interfaces. When inheriting from a class, the derived class inherits the members including the code of the base class. The important point to remember is that Constructors and Destructors are not inherited from the base class.

The syntax of inheritance is as in the following;

Class derivedClass :baseClass, Iterface1, Interface2 { body }

For example we are defining two classes, Father and Child. You notice at line 7, we are implementing inheritance by using a colon (:); at this moment all the properties belonging to the Father Class is accessible to the Child class automatically.



```
//Entry point
static void Main(string[] args)
{

    Father fObj = new Father();
    fObj.FatherMethod();

    //Here Child object can access both class methods
    Child cObj = new Child();
    cObj.|
    cObj.
}
```

```
using System;
    namespace oops
    {
       //Base Class
       public class Father
       {
          public void FatherMethod()
          {
             Console.WriteLine("this property belong to Father");
          }
       }

   //Derived class
       public class Child : Father
       {
          public void ChildMethod()
          {
             Console.WriteLine("this property belong to Child");
          }
       }


       class Inheritance
       {
          //Entry point
          static void Main(string[] args)
          {
             Father fObj = new Father();
             fObj.FatherMethod();

         //Here Child object can access both class methods
             Child cObj = new Child();
             cObj.FatherMethod();
```

```
            cObj.ChildMethod();
        }
    }
}
```

At line 11 , the Intellisense only shows the Father class functions but at line 15 to 16 the Child class object is able to access both class methods as in the following.

We can create a class in the VB.Net language or another .NET supported language and can inherit them in a C# .Net class and vice versa. But a class developed in C++ or other unmanaged environment can't be inherited in .NET.

Note

Cross-language and multiple inheritance is not supported by .NET.

Accessibility

Accessibility sets the visibility of the member to outside assemblies or derived types. The following table describes member accessibility;

Modifiers	Outside Assembly	Derived Class
private	No	No
public	Yes	Yes
protected	No	No
internal	Yes (this assembly only)	Yes (this assembly only)
internal protected	Yes (this assembly only)	Yes

Constructor in Inheritance

Constructors in a base class are not inherited in a derived class. A derived class has a base portion and derived portion. The base portion initializes the base portion, and the constructor of the derived class initializes the derived portion.

The following is the syntax of a constructor in inheritance:

Accessibility modifier classname(parameterlist1) : base(parameterlist2) { body }

So the base keyword refers to the base class constructor, while parameterlist2 determines which overloaded base class constructor is called.

In the following example, the Child class's constructor calls the single-argument constructor of the base Father class;

```
using System;
   namespace oops
   {
     //Base Class
     public class Father
     {
       //constructor
       public Father()
       {
         Console.WriteLine("Father class constructor");
       }

    public void FatherMethod()
       {
         Console.WriteLine("this property belong to Father");
       }
     }

  //Derived class
     public class Child : Father
     {
       public Child()
         : base()
       {
         Console.WriteLine("child class constructor");
       }

    public void ChildMethod()
       {
         Console.WriteLine("this property belong to Child");
       }
     }

  class Inheritance
     {
       //Entry point
       static void Main(string[] args)
       {
         //Here Child object can access both class methods
         Child cObj = new Child();
         cObj.FatherMethod();
         cObj.ChildMethod();
```

```
                Console.ReadKey();
            }
        }
    }
```

At line 4, we are defining a base Father Class constructor and in the derived class Child, at line 8 we are initializing it explicitly via base keyword. If we pass any parameter in the base class constructor then we have to provide them in the base block of the child class constructor.

Virtual Methods

By declaring a base class function as virtual, you allow the function to be overridden in any derived class. The idea behind a virtual function is to redefine the implementation of the base class method in the derived class as required. If a method is virtual in the base class then we have to provide the override keyword in the derived class. Neither member fields nor static functions can be declared as virtual.

```
using System;
    namespace oops
    {
        class myBase
        {
            //virtual function
            public virtual void VirtualMethod()
            {
                Console.WriteLine("virtual method defined in the base class");
             }
        }

  class myDerived : myBase
        {
            // redifing the implementation of base class method
            public override void VirtualMethod()
            {
                Console.WriteLine("virtual method defined in the Derive class");
            }
        }


  class virtualClass
        {
            static void Main(string[] args)
            {
                // class instance
                new myDerived().VirtualMethod();
                Console.ReadKey();
            }
        }
    }
```

Hiding Methods

If a method with the same signature is declared in both base and derived classes, but the methods are not declared as virtual and overriden respectively, then the derived class version is said to hide the base class version. In most cases, you would want to override methods rather than hide them. Otherwise .NET automatically generates a warning.

In the following example, we are defining a VirutalMethod() in the myBase class but not overriding it in the derived class, so in that case the compiler will generate a warning. The compiler will assume that you are hiding the base class method. So to overcome that problem, if you prefix the new keyword in the derived class method then the compiler will prefer the most derived version method. You can still access the base class method in the derived class by using the base keyword.

```
using System;
    namespace oops
    {
        class myBase
        {
            //virtual function
            public virtual void VirtualMethod()
            {
                Console.WriteLine("virtual method defined in the base class");
            }
        }


   class myDerived : myBase
        {
            // hiding the implementation of base class method
           public new void VirtualMethod()
            {
                Console.WriteLine("virtual method defined in the Derive class");
      //still access the base class method
                base.VirtualMethod();
            }
        }

   class virtualClass
        {
            static void Main(string[] args)
            {
                // class instance
                new myDerived().VirtualMethod();
                Console.ReadKey();
            }
        }
    }
```

Abstract Classes

C# allows both classes and functions to be declared abstract using the abstract keyword. You can't create an instance of an abstract class. An abstract member has a signature but no function body and they must be overridden in any non-abstract derived class. Abstract classes exist primarily for inheritance. Member functions, properties and indexers can be abstract. A class with one or more abstract members must be abstract as well. Static members can't be abstract.

In this example, we are declaring an abstract class Employess with a method displayData() that does not have an implementation. Then we are implementing the displayData() body in the derived class. One point to be noted here is that we have to prefixe the abstract method with the override keyword in the derived class.

```
using System;
    namespace oops
    {
        //abstract class
        public abstract class Employess
        {
            //abstract method with no implementation
            public abstract void displayData();
        }

  //derived class
        public class test : Employess
        {
            //abstract class method implementation
            public override void displayData()
            {
                Console.WriteLine("Abstract class method");
            }
        }

  class abstractClass
        {
            static void Main(string[] args)
            {
                // class instance
                new test().displayData();
            }
        }
    }
```

Sealed Classes

Sealed classes are the reverse of abstract classes. While abstract classes are inherited and are refined in the derived class, sealed classes cannot be inherited. You can create an instance of a sealed class. A sealed class is used to prevent further refinement through inheritance.

Suppose you are a developer of a class library and some of the classes in the class library are extensible but other classes are not extensible so in that case those classes are marked as sealed.

```
using System;
    namespace oops
    {
        sealed class SealedClass
        {
            void myfunv();
        }

  public class test : SealedClass //wrong. will give compilation error
        {
        }
    }
```

Interface

An interface is a set of related functions that must be implemented in a derived class. Members of an interface are implicitly public and abstract. Interfaces are similar to abstract classes. First, both types must be inherited; second, you cannot create an instance of either. Although there are several differences as in the following;

- An Abstract class can contain some implementations but an interface can't.
- An Interface can only inherit other interfaces but abstract classes can inherit from other classes and interfaces.
- An Abstract class can contain constructors and destructors but an interface can't.
- An Abstract class contains fields but interfaces don't.

So the question is, which of these to choose? Select interfaces because with an interface, the derived type still can inherit from another type and interfaces are more straightforward than abstract classes.

```
using System;
    namespace oops
    {
        // interface
        public interface xyz
        {
          void methodA();
          void methodB();
        }
        // interface method implementation
        class test : xyz
        {
            public void methodA()
            {
                Console.WriteLine("methodA");
            }
            public void methodB()
            {
                Console.WriteLine("methodB");
            }
        }

    class interfaceDemo
        {
            static void Main(string[] args)
            {
                test obj = new test();
                obj.methodA();
                obj.methodB();
            }
        }
    }
```

An interface can be inherited from other interfaces as in the following:

```
public interface xyz
    {
       void methodA();
        void methodB();
    }

public interface abc : xyz
    {
        void methodC();
    }
```

Polymorphism

Polymorphism is the ability to treat the various objects in the same manner. It is one of the significant benefits of inheritance. We can decide the correct call at runtime based on the derived type of the base reference. This is called late binding.

In the following example, instead of having a separate routine for the hrDepart, itDepart and financeDepart classes, we can write a generic algorithm that uses the base type functions. The method LeaderName() declared in the base abstract class is redefined as per our needs in 2 different classes.

```
using System;
    namespace oops
    {
       public abstract class Employee
       {
          public virtual void LeaderName()
          {
          }
       }

   public class hrDepart : Employee
       {
          public override void LeaderName()
          {
             Console.WriteLine("Mr. jone");
          }
       }

   public class itDepart : Employee
       {
          public override void LeaderName()
          {
             Console.WriteLine("Mr. Tom");
          }
       }


   public class financeDepart : Employee
       {
          public override void LeaderName()
          {
             Console.WriteLine("Mr. Linus");
          }
       }

   class PolymorphismDemo
       {
          static void Main(string[] args)
          {
             hrDepart obj1 = new hrDepart();
             itDepart obj2 = new itDepart();
             financeDepart obj3 = new financeDepart();

         obj1.LeaderName();
             obj2.LeaderName();
             obj3.LeaderName();
             Console.ReadKey();
          }
```

```
        }
    }
```

Struct

C# struct also known as C# structure is a simple user-defined type, a lightweight alternative to a class. A stuct in C# is simply a composite data type consisting of a number elements of other types

Similar to classes, structures have behaviors and attributes. As a value type, structures directly contain their value so their object or instance is stored on the stack.

C# Struts support access modifiers, constructors, indexers, methods, fields, nested types, operators, and properties. There are few properties.

- A struct is used to improve the performance and clarity of code.
- A struct uses fewer resources in memory than a class.
- When we have small and frequent use of some work use structs over classes.
- Performance can suffer when using structures in situations where reference types are expected due to boxing and unboxing.
- You should pass structs to method as ref parameters in order to avoid the performance loss associated with copying data.
- Structs reside on the stack, so we should keep them small.
- Structs can't be inherited and we can say they are sealed.
- Structure implicitly inherits from System.ValueType.
- The default constructor of a structure initializes each field to a default value. You cannot replace the default constructor of a structure.
- You can't define destructor for structs.

How to define a struct in C#

```
public struct Student {
    int id;
    int zipcode;
    double salary;
}
```

Note: Struct members may not be declared protected.

Structs are simple to use and can prove to be useful at times. Just keep in mind that they're created on the stack and that you're not dealing with references to them but dealing directly with them. Whenever you have a need for a type that will be used often and is mostly just a piece of data, structs might be a good option.

How to use a struct in C#

```
using System;
namespace example_struct {
    class Program {
        public struct Student {
            int id;
            int zipcode;
            double salary;
            // all the members of the struct has to be initialized in this way
            public Student(int id, int zipcode, double salary) {
                this.id = id;
                this.zipcode = zipcode;
                this.salary = salary;
            }
            // all the members of the struct has to be initialized either in this way
            public Student(int id, int zipcode) {
                this.id = id;
                this.zipcode = zipcode;
                this.salary = 3400.00;
            }
            // if you left any member of a struct uninitialzed it will give error
            // code below will give error because the zipcode and salary field is left uninitialzed
            //public Student(int id)
            //{
            // this.id = id;
            //}
            // struct can also have copy constructor but have to be fully initialzed
            public Student(Student x) {
                this.id = x.id;
                this.zipcode = x.zipcode;
                this.salary = x.salary;
            }
            public void DisplayValues() {
                Console.WriteLine("ID: " + this.id.ToString());
                Console.WriteLine("Zipcode : " + this.zipcode.ToString());
                Console.WriteLine("Salary : " + this.salary.ToString());
            }
        }
        static void Main(string[] args) {
            Student stu = new Student(12, 201301, 4560.00);
            Student stu1 = new Student(stu);
            stu.DisplayValues();
            Console.WriteLine("Copy constructor values");
            stu1.DisplayValues();
            Console.ReadLine();
        }
    }
}
```

Properties

A property in C# is a member of a class that provides a flexible mechanism for classes to expose private fields. Internally, C# properties are special methods called accessors. A C# property have two accessors, get property accessor and set property accessor.

A get accessor returns a property value, and a set accessor assigns a new value. The value keyword represents the value of a property.

Properties in C# and .NET have various access levels that is defined by an access modifier. Properties can be read-write, read-only, or write-only. The read-write property implements both, a get and a set accessor.

A write-only property implements a set accessor, but no get accessor. A read-only property implements a get accessor, but no set accessor.

For example, the above program can be modified with a property X as follows.

```
class MyClass
{
   private int x;
   public int X
   {
      get
      {
         return x;
      }
      set
      {
         x = value;
      }
   }
}
class MyClient
{
   public static void Main()
   {
      MyClass mc = new MyClass();
      mc.X = 10;
      int xVal = mc.X;
      Console.WriteLine(xVal);//Displays 10
   }
}
```

The object of the class MyClass can access the property X as follows.

```
MyClass mc = new MyClass();
```

mc.X = 10; // calls set accessor of the property X, and pass 10 as value of the standard field 'value'.

This is used for setting value for the data member x.

Console.WriteLine(mc.X);// displays 10. Calls the get accessor of the property X.

Delegate

A delegate in C# is similar to a function pointer in C or C++. Using a delegate allows the programmer to encapsulate a reference to a method inside a delegate object. The delegate object can then be passed to code which can call the referenced method, without having to know at compile time which method will be invoked.

Call a Function directly - No Delegate
In most cases, when we call a function, we specify the function to be called directly. If the class MyClass has a function named Process, we'd normally call it like this (SimpleSample.cs):

```
using System;

namespace NoDelegate
{
   public class MyClass
   {
      public void Process()
      {
         Console.WriteLine("Process() begin");
         Console.WriteLine("Process() end");
      }
   }

   public class Test
   {
      static void Main(string[] args)
      {
         MyClass myClass = new MyClass();
         myClass.Process();
      }
   }
}
```

That works well in most situations. Sometimes, however, we don't want to call a function directly - we'd like to be able to pass it to somebody else so that they can call it. This is especially useful in an event-driven system such as a graphical user interface, when I want some code to be executed when the user clicks on a button, or when I want to log some information but can't specify how it is logged.

The very basic Delegate

An interesting and useful property of a delegate is that it does not know or care about the class of the object that it references. Any object will do; all that matters is that the method's argument types and return type match the delegate's. This makes delegates perfectly suited for "anonymous" invocation.

The signature of a single cast delegate is shown below:

delegate result-type identifier ([parameters]);

where:

- result-type: The result type, which matches the return type of the function.
- identifier: The delegate name.
- parameters: The Parameters, that the function takes.

Examples:

public delegate void SimpleDelegate () This declaration defines a delegate named SimpleDelegate, which will encapsulate any method that takes no parameters and returns no value.
public delegate int ButtonClickHandler (object obj1, object obj2) This declaration defines a delegate named ButtonClickHandler, which will encapsulate any method that takes two objects as parameters and returns an int.

A delegate will allow us to specify what the function we'll be calling *looks like* without having to specify *which* function to call. The declaration for a delegate looks just like the declaration for a function, except that in this case, we're declaring the signature of functions that this delegate can reference.

There are three steps in defining and using delegates:

- Declaration
- Instantiation
- Invocation

A very basic example (SimpleDelegate1.cs):

```
using System;

namespace BasicDelegate
{
   // Declaration
   public delegate void SimpleDelegate();

   class TestDelegate
   {
      public static void MyFunc()
      {
         Console.WriteLine("I was called by delegate ...");
      }

      public static void Main()
      {
         // Instantiation
         SimpleDelegate simpleDelegate = new SimpleDelegate(MyFunc);
```

```
        // Invocation
        simpleDelegate();
    }
}
```

Calling Static Functions

For our next, more advanced example (SimpleDelegate2.cs), declares a delegate that takes a single string parameter and has no return type:

```
using System;

namespace SimpleDelegate
{
    // Delegate Specification
    public class MyClass
    {
        // Declare a delegate that takes a single string parameter
        // and has no return type.
        public delegate void LogHandler(string message);

        // The use of the delegate is just like calling a function directly,
        // though we need to add a check to see if the delegate is null
        // (that is, not pointing to a function) before calling the function.
        public void Process(LogHandler logHandler)
        {
            if (logHandler != null)
            {
                logHandler("Process() begin");
            }

            if (logHandler != null)
            {
                logHandler ("Process() end");
            }
        }
    }

    // Test Application to use the defined Delegate
    public class TestApplication
    {
        // Static Function: To which is used in the Delegate. To call the Process()
        // function, we need to declare a logging function: Logger() that matches
        // the signature of the delegate.
        static void Logger(string s)
        {
            Console.WriteLine(s);
        }

        static void Main(string[] args)
        {
            MyClass myClass = new MyClass();

            // Crate an instance of the delegate, pointing to the logging function.
            // This delegate will then be passed to the Process() function.
            MyClass.LogHandler myLogger = new MyClass.LogHandler(Logger);
            myClass.Process(myLogger);
        }
    }
}
```

Calling Member Functions

In the simple example above, the **Logger( )** function merely writes the string out. A different function might want to log the information to a file, but to do this, the function needs to know what file to write the information to (SimpleDelegate3.cs)

```
using System;
using System.IO;

namespace SimpleDelegate
{
    // Delegate Specification
    public class MyClass
    {
        // Declare a delegate that takes a single string parameter
        // and has no return type.
        public delegate void LogHandler(string message);

        // The use of the delegate is just like calling a function directly,
        // though we need to add a check to see if the delegate is null
        // (that is, not pointing to a function) before calling the function.
        public void Process(LogHandler logHandler)
        {
            if (logHandler != null)
            {
                logHandler("Process() begin");
            }

            if (logHandler != null)
            {
                logHandler ("Process() end");
            }
        }
    }

    // The FileLogger class merely encapsulates the file I/O
    public class FileLogger
    {
        FileStream fileStream;
        StreamWriter streamWriter;

        // Constructor
        public FileLogger(string filename)
        {
            fileStream = new FileStream(filename, FileMode.Create);
            streamWriter = new StreamWriter(fileStream);
        }

        // Member Function which is used in the Delegate
        public void Logger(string s)
        {
            streamWriter.WriteLine(s);
        }

        public void Close()
        {
            streamWriter.Close();
            fileStream.Close();
        }
    }

    // Main() is modified so that the delegate points to the Logger()
    // function on the fl instance of a FileLogger. When this delegate
    // is invoked from Process(), the member function is called and
    // the string is logged to the appropriate file.
    public class TestApplication
    {
        static void Main(string[] args)
        {
            FileLogger fl = new FileLogger("process.log");

            MyClass myClass = new MyClass();

            // Crate an instance of the delegate, pointing to the Logger()
            // function on the fl instance of a FileLogger.
            MyClass.LogHandler myLogger = new MyClass.LogHandler(fl.Logger);
            myClass.Process(myLogger);
            fl.Close();
        }
    }
}
```

The cool part here is that we didn't have to change the Process() function; the code to all the delegate is the same regardless of whether it refers to a **static** or **member** function.

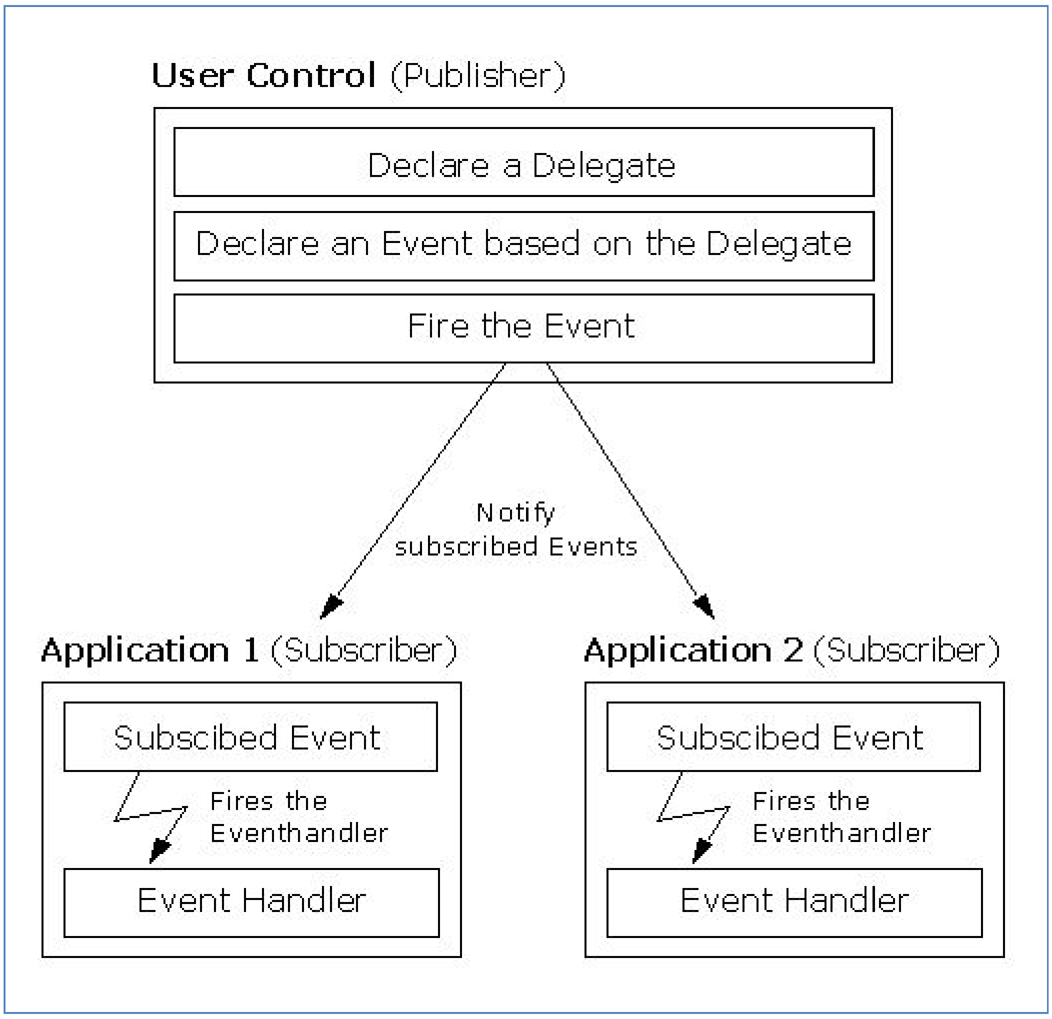
Multicasting
Being able to point to member functions is nice, but there are more tricks you can do with delegates. In C#, delegates are ***multicast***, which means that they can **point to more than one function at a time** (that is, they're based off the System.MulticastDelegate type). A multicast delegate maintains a list of functions that will all be called when the delegate is invoked. We can add back in the logging function from the first example, and call both delegates. Here's what the code looks like:

```
using System;
using System.IO;

namespace SimpleDelegate
{
    // Delegate Specification
    public class MyClass
    {
        // Declare a delegate that takes a single string parameter
        // and has no return type.
        public delegate void LogHandler(string message);

        // The use of the delegate is just like calling a function directly,
        // though we need to add a check to see if the delegate is null
        // (that is, not pointing to a function) before calling the function.
        public void Process(LogHandler logHandler)
        {
            if (logHandler != null)
            {
                logHandler("Process() begin");
            }

            if (logHandler != null)
            {
                logHandler ("Process() end");
            }
        }
    }

    // The FileLogger class merely encapsulates the file I/O
    public class FileLogger
    {
        FileStream fileStream;
        StreamWriter streamWriter;

        // Constructor
        public FileLogger(string filename)
        {
            fileStream = new FileStream(filename, FileMode.Create);
            streamWriter = new StreamWriter(fileStream);
        }

        // Member Function which is used in the Delegate
        public void Logger(string s)
        {
            streamWriter.WriteLine(s);
        }

        public void Close()
        {
            streamWriter.Close();
            fileStream.Close();
        }
    }

    // Test Application which calls both Delegates
    public class TestApplication
    {
         // Static Function which is used in the Delegate
        static void Logger(string s)
        {
            Console.WriteLine(s);
        }

        static void Main(string[] args)
        {
            FileLogger fl = new FileLogger("process.log");

            MyClass myClass = new MyClass();

            // Crate an instance of the delegates, pointing to the static
            // Logger() function defined in the TestApplication class and
            // then to member function on the fl instance of a FileLogger.
            MyClass.LogHandler myLogger = null;
            myLogger += new MyClass.LogHandler(Logger);
            myLogger += new MyClass.LogHandler(fl.Logger);

            myClass.Process(myLogger);
            fl.Close();
        }
    }
}
```

Events

The Event model in C# finds its roots in the event programming model that is popular in asynchronous programming. The basic foundation behind this programming model is the idea of "publisher and subscribers." In this model, you have ***publishers*** who will do some logic and publish an "event." Publishers will then send out their event only to ***subscribers*** who have subscribed to receive the specific event.

In C#, any object can *publish* a set of events to which other applications can *subscribe*. When the publishing class raises an event, all the subscribed applications are notified. The following figure shows this mechanism.



Conventions

The following important conventions are used with events:

- Event Handlers in the .NET Framework return void and take two parameters.
- The first paramter is the source of the event; that is the publishing object.
- The second parameter is an object derived from EventArgs.
- Events are properties of the class publishing the event.
- The keyword event controls how the event property is accessed by the subscribing classes.

Simple Event

Let's modify our logging example from above to use an event rather than a delegate:

```
using System;
using System.IO;

namespace SimpleEvent
{
   /* ========= Publisher of the Event ============== */
   public class MyClass
   {
      // Define a delegate named LogHandler, which will encapsulate
      // any method that takes a string as the parameter and returns no value
      public delegate void LogHandler(string message);

      // Define an Event based on the above Delegate
      public event LogHandler Log;

      // Instead of having the Process() function take a delegate
      // as a parameter, we've declared a Log event. Call the Event,
      // using the OnXXXX Method, where XXXX is the name of the Event.
      public void Process()
      {
         OnLog("Process() begin");
         OnLog("Process() end");
      }

      // By Default, create an OnXXXX Method, to call the Event
      protected void OnLog(string message)
      {
         if (Log != null)
         {
            Log(message);
         }
      }
   }

   // The FileLogger class merely encapsulates the file I/O
   public class FileLogger
   {
      FileStream fileStream;
      StreamWriter streamWriter;

      // Constructor
      public FileLogger(string filename)
      {
         fileStream = new FileStream(filename, FileMode.Create);
         streamWriter = new StreamWriter(fileStream);
      }

      // Member Function which is used in the Delegate
      public void Logger(string s)
      {
         streamWriter.WriteLine(s);
      }

      public void Close()
      {
         streamWriter.Close();
```

```
            fileStream.Close();
        }
    }

    /* ========= Subscriber of the Event ============== */
    // It's now easier and cleaner to merely add instances
    // of the delegate to the event, instead of having to
    // manage things ourselves
    public class TestApplication
    {
        static void Logger(string s)
        {
            Console.WriteLine(s);
        }

        static void Main(string[] args)
        {
            FileLogger fl = new FileLogger("process.log");
            MyClass myClass = new MyClass();

            // Subscribe the Functions Logger and fl.Logger
            myClass.Log += new MyClass.LogHandler(Logger);
            myClass.Log += new MyClass.LogHandler(fl.Logger);

            // The Event will now be triggered in the Process() Method
            myClass.Process();

            fl.Close();
        }
    }
}
```

The Second Change Event Example

Suppose you want to create a Clock class that uses events to notify potential subscribers whenever the local time changes value by one second. Here is the complete, documented example:

```
using System;
using System.Threading;

namespace SecondChangeEvent
{
   /* ======================= Event Publisher =============================== */

   // Our subject -- it is this class that other classes
   // will observe. This class publishes one event:
   // SecondChange. The observers subscribe to that event.
   public class Clock
   {
      // Private Fields holding the hour, minute and second
      private int _hour;
      private int _minute;
      private int _second;

      // The delegate named SecondChangeHandler, which will encapsulate
      // any method that takes a clock object and a TimeInfoEventArgs
      // object as the parameter and returns no value. It's the
      // delegate the subscribers must implement.
      public delegate void SecondChangeHandler (
         object clock,
    TimeInfoEventArgs timeInformation
  );

  // The event we publish
  public event SecondChangeHandler SecondChange;

  // The method which fires the Event
  protected void OnSecondChange(
    object clock,
    TimeInfoEventArgs timeInformation
  )
  {
    // Check if there are any Subscribers
    if (SecondChange != null)
    {
      // Call the Event
      SecondChange(clock,timeInformation);
    }
  }

  // Set the clock running, it will raise an
  // event for each new second
  public void Run()
  {
    for(;;)
    {
      // Sleep 1 Second
      Thread.Sleep(1000);

      // Get the current time
      System.DateTime dt = System.DateTime.Now;

      // If the second has changed
      // notify the subscribers
      if (dt.Second != _second)
      {
        // Create the TimeInfoEventArgs object
        // to pass to the subscribers
        TimeInfoEventArgs timeInformation =
          new TimeInfoEventArgs(
          dt.Hour,dt.Minute,dt.Second);

        // If anyone has subscribed, notify them
        OnSecondChange (this,timeInformation);
      }

      // update the state
      _second = dt.Second;
      _minute = dt.Minute;
      _hour = dt.Hour;

    }
  }
}

// The class to hold the information about the event
// in this case it will hold only information
// available in the clock class, but could hold
// additional state information
public class TimeInfoEventArgs : EventArgs
{
  public TimeInfoEventArgs(int hour, int minute, int second)
  {
    this.hour = hour;
    this.minute = minute;
    this.second = second;
  }
  public readonly int hour;
  public readonly int minute;
  public readonly int second;
}

/* ======================= Event Subscribers =============================== */

// An observer. DisplayClock subscribes to the
// clock's events. The job of DisplayClock is
// to display the current time
public class DisplayClock
{
  // Given a clock, subscribe to
  // its SecondChangeHandler event
  public void Subscribe(Clock theClock)
  {
    theClock.SecondChange +=
      new Clock.SecondChangeHandler(TimeHasChanged);
  }

  // The method that implements the
  // delegated functionality
  public void TimeHasChanged(
    object theClock, TimeInfoEventArgs ti)
  {
    Console.WriteLine("Current Time: {0}:{1}:{2}",
      ti.hour.ToString(),
      ti.minute.ToString(),
      ti.second.ToString());
  }
}

// A second subscriber whose job is to write to a file
public class LogClock
{
  public void Subscribe(Clock theClock)
  {
    theClock.SecondChange +=
      new Clock.SecondChangeHandler(WriteLogEntry);
  }

  // This method should write to a file
  // we write to the console to see the effect
  // this object keeps no state
  public void WriteLogEntry(
    object theClock, TimeInfoEventArgs ti)
  {
    Console.WriteLine("Logging to file: {0}:{1}:{2}",
      ti.hour.ToString(),
      ti.minute.ToString(),
      ti.second.ToString());
  }
}

/* ======================= Test Application =============================== */
```

```
// Test Application which implements the
// Clock Notifier - Subscriber Sample
public class Test
{
  public static void Main()
  {
    // Create a new clock
    Clock theClock = new Clock();

    // Create the display and tell it to
    // subscribe to the clock just created
    DisplayClock dc = new DisplayClock();
    dc.Subscribe(theClock);

    // Create a Log object and tell it
    // to subscribe to the clock
    LogClock lc = new LogClock();
    lc.Subscribe(theClock);

    // Get the clock started
    theClock.Run();
  }
```

Generics

C# collection types are designed to store, manage and manipulate similar data more efficiently. Data manipulation includes adding, removing, finding, and inserting data in the collection. Collection types implement the following common functionality:

- Adding and inserting items to a collection
- Removing items from a collection
- Finding, sorting, searching items
- Replacing items
- Copy and clone collections and items
- Capacity and Count properties to find the capacity of the collection and number of items in the collection

.NET supports two types of collections, generic collections and non-generic collections. Prior to .NET 2.0, it was just collections and when generics were added to .NET, generics collections were added as well.

Generic collections with work generic data type.

The following table lists and matches these classes.

Non-generic		**Generic**
ArrayList	------------->	List
HashTable	------------->	Dictionary

SortedList -------------> SortedList
Stack -------------> Stack
Queue -------------> Queue

1. Non-Generic
In non-generic collections, each element can represent a value of a different type. The collection size is not fixed. Items from the collection can be added or removed at runtime.

C# ArrayList

ArrayList class is a collection that can be used for any types or objects.

1. Arraylist is a class that is similar to an array, but it can be used to store values of various types.
2. An Arraylist doesn't have a specific size.
3. Any number of elements can be stored.

```
using System.Collections;

protected void Button1_Click(object sender, EventArgs e)
{
    ArrayList al = new ArrayList();
    string str = "kiran teja jallepalli";
    int x = 7;
    DateTime d = DateTime.Parse("8-oct-1985");
    al.Add(str);
    al.Add(x);
    al.Add(d);

    foreach (object o in al)
    {
      Response.Write(o);
      Response.Write("<br>");
    }
}
```

Output

```
kiran teja jallepalli
7
10/8/1985 12:00:00 AM
```

Foreach Loop

It executes for each and every item that exists in the arraylist object. Every time the loop rotates it reads one item from the arraylist and assignes it to the variable.

Note: Arraylist allocates memory for 4 items, whenever an object is created. When a fifth item is added, memory for another 4 items are added. it reduces the memory allocated for the object.

Capacity: is a property that returns the number of items for which memory is allocated.

C# HashTable

HashTable is similar to arraylist but represents the items as a combination of a key and value.

```
using System.Collections;


protected void Button2_Click(object sender, EventArgs e)
{
   Hashtable ht = new Hashtable();
   ht.Add("ora", "oracle");
   ht.Add("vb", "vb.net");
   ht.Add("cs", "cs.net");
   ht.Add("asp", "asp.net");

   foreach (DictionaryEntry d in ht)
   {
     Response.Write (d.Key + " " + d.Value);

     Response.Write("<br>");

   }
}
```

Output

```
vb  vb.net
asp asp.net
cs  cs.net
ora oracle
```

DictonaryEntry: is a class whose object represents the data in a combination of key & value pairs.

C# SortedList

1. Is a class that has the combination of arraylist and hashtable.
2. Represents the data as a key and value pair.

3. Arranges all the items in sorted order.

```
using System.Collections;

protected void Button3_Click(object sender, EventArgs e)
{
    SortedList sl = new SortedList();
    sl.Add("ora", "oracle");
    sl.Add("vb", "vb.net");
    sl.Add("cs", "cs.net");
    sl.Add("asp", "asp.net");

    foreach (DictionaryEntry d in sl)
    {
        Response.Write(d.Key + " " + d.Value);
        Response.Write("<br>");

    }
}
```

Output

```
asp  asp.net
cs   cs.net
ora  oracle
vb   vb.net
```

C# Stack

```
protected void Button4_Click(object sender, EventArgs e)
{
    Stack stk = new Stack();
    stk.Push("cs.net");
    stk.Push("vb.net");
    stk.Push("asp.net");
    stk.Push("sqlserver");

    foreach (object o in stk)
    {
        Response.Write(o + "<br>");
    }
}
```

Output

```
sqlserver
asp.net
```

vb.net
cs.net

C# Queue

```
using System.Collections;

protected void Button5_Click(object sender, EventArgs e)
{
   Queue q = new Queue();
   q.Enqueue("cs.net");
   q.Enqueue("vb.net");
   q.Enqueue("asp.net");
   q.Enqueue("sqlserver");

   foreach (object o in q)
   {
     Response.Write(o + "<br>");
   }
}
```

Output

```
cs.net
vb.net
asp.net
sqlserver
```

2. Generic Collections

Generic Collections work on the specific type that is specified in the program whereas non-generic collections work on the object type.

a. Specific type
b. Array Size is not fixed
c. Elements can be added / removed at runtime.

C# List

```
using System.Collections.Generic;

protected void Button1_Click(object sender, EventArgs e)
{
      List<int> lst = new List<int>();
      lst.Add(100);
      lst.Add(200);
```

```
        lst.Add(300);
        lst.Add(400);
        foreach (int i in lst)
        {
            Response.Write(i+"<br>");
        }
}
```

C# Dictonary

```
using System.Collections.Generic;

 protected void Button1_Click(object sender, EventArgs e)
 {
     Dictionary<int, string> dct = new Dictionary<int, string>();
     dct.Add(1, "cs.net");
     dct.Add(2, "vb.net");
     dct.Add(3, "vb.net");
     dct.Add(4, "vb.net");
     foreach (KeyValuePair<int, string> kvp in dct)
     {
         Response.Write(kvp.Key + " " + kvp.Value);
         Response.Write("<br>");
     }
  }
```

C# SortedList

```
using System.Collections.Generic;

protected void Button3_Click(object sender, EventArgs e)
{
SortedList<string, string> sl = new SortedList<string, string>();
sl.Add("ora", "oracle");
sl.Add("vb", "vb.net");
sl.Add("cs", "cs.net");
sl.Add("asp", "asp.net");

foreach (KeyValuePair<string, string> kvp in sl)
{
Response.Write(kvp.Key + " " + kvp.Value);
Response.Write("<br>");
}
}
```

C# Stack

```
using System.Collections.Generic;

protected void Button4_Click(object sender, EventArgs e)
```

```
{
Stack<string> stk = new Stack<string>();
stk.Push("cs.net");
stk.Push("vb.net");
stk.Push("asp.net");
stk.Push("sqlserver");

foreach (string s in stk)
{
Response.Write(s + "<br>");
}
}
```

C# Queue

```
using System.Collections.Generic;

protected void Button1_Click(object sender, EventArgs e)
  {
  Queue<string> q = new Queue<string>();

  q.Enqueue("cs.net");
  q.Enqueue("vb.net");
  q.Enqueue("asp.net");
  q.Enqueue("sqlserver");

  foreach (string s in q)
  {
  Response.Write(s + "<br>");
  }
  }
```

Extension Methods

Extension methods, as the name suggests, are additional methods. Extension methods allow you to inject additional methods without modifying, deriving or recompiling the original class, struct or interface. Extension methods can be added to your own custom class, .NET framework classes, or third party classes or interfaces.

In the following example, IsGreaterThan() is an extension method for int type, which returns true if the value of the int variable is greater than the supplied integer parameter.

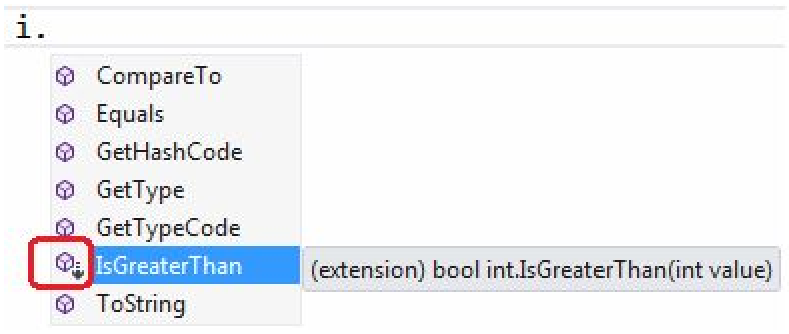
Example:

```
namespace ExtensionMethods
{
    public static class IntExtensions
     {
        public static bool IsGreaterThan(this int i, int value)
        {
            return i > value;
        }
    }
}
```

The IsGreaterThan() method is not a method of int data type (Int32 struct). It is an extension method written by the programmer for the int data type.

The IsGreaterThan() extension method will be available throughout the application by including the namespace in which it has been defined.

The extension methods have a special symbol in intellisense of the visual studio, so that you can easily differentiate between class methods and extension methods.



Extension Method Symbol in visual studio intellisense

Use of Extension method:

```
using ExtensionMethods;

class Program
{
    static void Main(string[] args)
    {
        int i = 10;

        bool result = i.IsGreaterThan(100);

        Console.WriteLine(result);
    }
}
```

Output: False

Async Await

Asynchronous programming is very helpful in this condition. By using Asynchronous programming, the Application can continue with the other work that does not depend on the completion of the whole task.

We will get all the benefits of traditional Asynchronous programming with much less effort by the help of async and await keywords.

Suppose, we are using two methods as Method1 and Method2 respectively and both the methods are not dependent on each other and Method1 is taking a long time to complete its task. In Synchronous programming, it will execute the first Method1 and it will wait for completion of this method and then it will execute Method2. Thus, it will be a time intensive process even though both the methods are not depending on each other.

We can run all the methods parallelly by using the simple thread programming but it will block UI and wait to complete all the tasks. To come out of this problem, we have to write too many codes in traditional programming but if we will simply use the async and await keywords, then we will get the solutions in much less code.

Also, we are going to see more examples, if any third Method as Method3 has a dependency of method1, then it will wait for the completion of Method1 with the help of await keyword.

Async and await are the code markers, which marks code positions from where the control should resume after a task completes.

Example :

```
class Program {
   static void Main(string[] args)
   {
      Method1();
      Method2();
      Console.ReadKey();
   }

   public static async Task Method1()
   {
      await Task.Run(() =>
      {
         for (int i = 0; i < 5; i++)
         {
            Console.WriteLine(" Method 1");
         }
      });
   }


   public static void Method2()
   {
      for (int i = 0; i < 5; i++)
      {
         Console.WriteLine(" Method 2");
      }
   }
}
```

In the code given above, Method1 and Method2 are not dependent on each other and we are calling from the Main method.

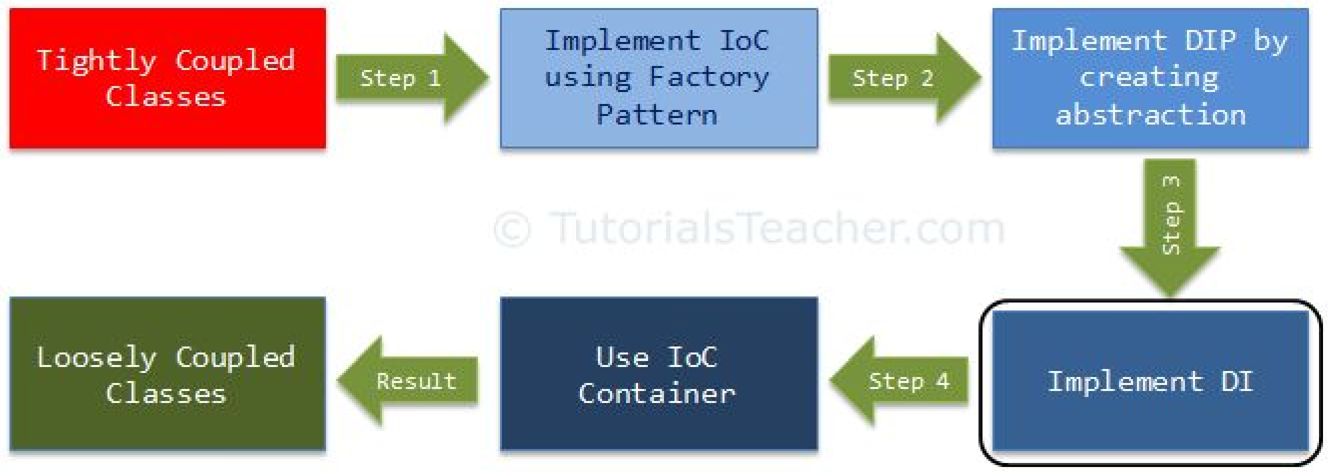
Here, we can clearly see Method1 and Method2 are not waiting for each other.

Output

```
Method 1
Method 2
Method 2
Method 1
Method 1
Method 2
Method 2
Method 1
Method 1
Method 2
```

Dependency Enjection

We created and used abstraction to make the classes loosely coupled. Here, we are going to implement Dependency Injection and strategy pattern together to move the dependency object creation completely out of the class. This is our third step in making the classes completely loose coupled.

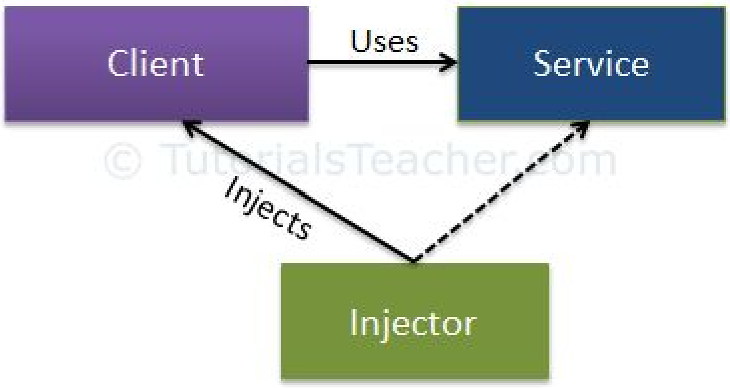


Dependency Injection (DI) is a design pattern used to implement IoC. It allows the creation of dependent objects outside of a class and provides those objects to a class through different ways. Using DI, we move the creation and binding of the dependent objects outside of the class that depends on them.

The Dependency Injection pattern involves 3 types of classes.

1. **Client Class:** The client class (dependent class) is a class which depends on the service class
2. **Service Class:** The service class (dependency) is a class that provides service to the client class.
3. **Injector Class:** The injector class injects the service class object into the client class.

The following figure illustrates the relationship between these classes:



As you can see, the injector class creates an object of the service class, and injects that object to a client object. In this way, the DI pattern separates the responsibility of creating an object of the service class out of the client class.

Types of Dependency Injection
As you have seen above, the injector class injects the service (dependency) to the client (dependent). The injector class injects dependencies broadly in three ways: through a constructor, through a property, or through a method.

Constructor Injection: In the constructor injection, the injector supplies the service (dependency) through the client class constructor.

Property Injection: In the property injection (aka the Setter Injection), the injector supplies the dependency through a public property of the client class.

Method Injection: In this type of injection, the client class implements an interface which declares the method(s) to supply the dependency and the injector uses this interface to supply the dependency to the client class.

Let's take an example from the previous chapter to maintain the continuity. In the previous section of DIP, we used Factory class inside the CustomerBusinessLogic class to get an object of the CustomerDataAccess object,

Example:

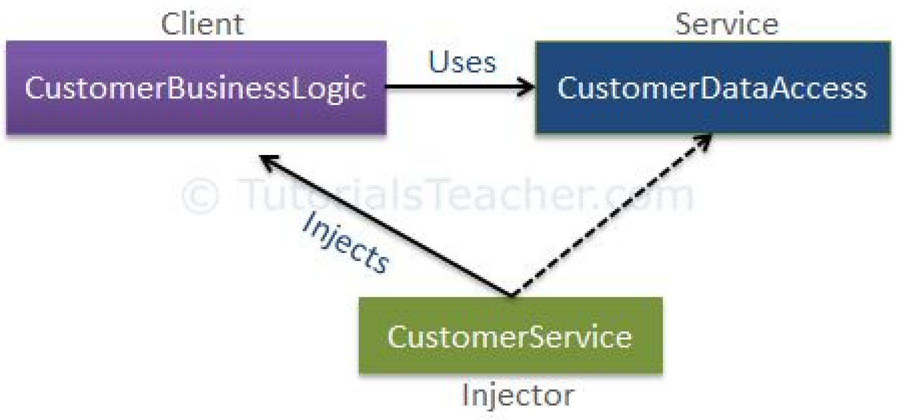
```
public interface ICustomerDataAccess
{
   string GetCustomerName(int id);
}

public class CustomerDataAccess: ICustomerDataAccess
{
   public CustomerDataAccess() {
   }

   public string GetCustomerName(int id) {
      return "Dummy Customer Name";
   }
}

public class DataAccessFactory
{
   public static ICustomerDataAccess GetCustomerDataAccessObj()
   {
      return new CustomerDataAccess();
```

```
    }
}

public class CustomerBusinessLogic
{
    ICustomerDataAccess _custDataAccess;

    public CustomerBusinessLogic()
    {
        _custDataAccess = DataAccessFactory.GetCustomerDataAccessObj();
    }

    public string GetCustomerName(int id)
    {
        return _custDataAccess.GetCustomerName(id);
    }
}
```

The problem with the above example is that we used DataAccessFactory inside the CustomerBusinessLogic class. So, suppose there is another implementation of ICustomerDataAccess and we want to use that new class inside CustomerBusinessLogic.

Then, we need to change the source code of the CustomerBusinessLogic class as well. The Dependency injection pattern solves this problem by injecting dependent objects via a constructor, a property, or an interface.

The following figure illustrates the DI pattern implementation for the above example.



As you see, the CustomerService class becomes the injector class, which sets an object of the service class (CustomerDataAccess) to the client class (CustomerBusinessLogic) either through a constructor, a property, or a method to achieve loose coupling. Let's explore each of these options.

Constructor Injection
As mentioned before, when we provide the dependency through the constructor, this is called a constructor injection.

Consider the following example where we have implemented DI using the constructor.

Example: Constructor Injection

```
public class CustomerBusinessLogic
{
    ICustomerDataAccess _dataAccess;
    public CustomerBusinessLogic(ICustomerDataAccess custDataAccess)
    {
        _dataAccess = custDataAccess;
    }

    public CustomerBusinessLogic()
    {
        _dataAccess = new CustomerDataAccess();
    }

    public string ProcessCustomerData(int id)
    {
        return _dataAccess.GetCustomerName(id);
    }
}

public interface ICustomerDataAccess
{
    string GetCustomerData(int id);
}

public class CustomerDataAccess: ICustomerDataAccess
{
    public CustomerDataAccess()
    {
    }

    public string GetCustomerName(int id)
    {
        //get the customer name from the db in real application
        return "Dummy Customer Name";
    }
}
```

In the above example, CustomerBusinessLogic includes the constructor with one parameter of type ICustomerDataAccess.

Now, the calling class must inject an object of ICustomerDataAccess.

Example: Inject Dependency

```
public class CustomerService
{
    CustomerBusinessLogic _customerBL;

    public CustomerService()
    {
        _customerBL = new CustomerBusinessLogic(new CustomerDataAccess());
    }

    public string GetCustomerName(int id) {
        return _customerBL.GetCustomerName(id);
    }
}
```

As you can see in the above example, the CustomerService class creates and injects the CustomerDataAccess object into the CustomerBusinessLogic class. Thus, the CustomerBusinessLogic class doesn't need to create an object of CustomerDataAccessusing the new keyword or using factory class. The calling class (CustomerService) creates and sets the appropriate DataAccess class to the CustomerBusinessLogic class. In this way, the CustomerBusinessLogic and CustomerDataAccess classes become "more" loosely coupled classes.

Property Injection
In the property injection, the dependency is provided through a public property. Consider the following example.

Example: Property Injection

```
public class CustomerBusinessLogic
{
    public CustomerBusinessLogic()
    {
    }

    public string GetCustomerName(int id)
    {
        return DataAccess.GetCustomerName(id);
    }

    public ICustomerDataAccess DataAccess { get; set; }
}

public class CustomerService
{
    CustomerBusinessLogic _customerBL;

    public CustomerService()
    {
        _customerBL = new CustomerBusinessLogic();
        _customerBL.DataAccess = new CustomerDataAccess();
    }

    public string GetCustomerName(int id) {
        return _customerBL.GetCustomerName(id);
    }
}
```

As you can see above, the CustomerBusinessLogic class includes the public property named DataAccess, where you can set an instance of a class that implements ICustomerDataAccess. So, CustomerService class creates and sets CustomerDataAccess class using this public property.

Method Injection
In the method injection, dependencies are provided through methods. This method can be a class method or an interface method.

The following example demonstrates the method injection using an interface based method.

Example: Interface Injection

```
interface IDataAccessDependency
{
    void SetDependency(ICustomerDataAccess customerDataAccess);
}
```

```
public class CustomerBusinessLogic : IDataAccessDependency
{
    ICustomerDataAccess _dataAccess;

    public CustomerBusinessLogic()
    {
    }

    public string GetCustomerName(int id)
    {
        return _dataAccess.GetCustomerName(id);
    }

    public void SetDependency(ICustomerDataAccess customerDataAccess)
    {
        _dataAccess = customerDataAccess;
    }
}

public class CustomerService
{
    CustomerBusinessLogic _customerBL;

    public CustomerService()
    {
        _customerBL = new CustomerBusinessLogic();
        ((IDataAccessDependency)_customerBL).SetDependency(new CustomerDataAccess());
    }

    public string GetCustomerName(int id) {
        return _customerBL.GetCustomerName(id);
    }
}
```

In the above example, the CustomerBusinessLogic class implements the IDataAccessDependency interface, which includes the SetDependency() mehtod. So, the injector class (CustomerService) will now use this method to inject the dependent class (CustomerDataAccess) to the client class.

Thus, you can use DI and strategy pattern to create loose coupled classes.