What is Singleton Design Pattern?

Ensures a class has only one instance and provides a global point of access to it.

A singleton is a class that only allows a single instance of itself to be created, and usually gives simple access to that instance.

Most commonly, singletons don't allow any parameters to be specified when creating the instance, since a second request of an instance with a different parameter could be problematic! (If the same instance should be accessed for all requests with the same parameter then the factory pattern is more appropriate.)

There are various ways to implement the Singleton Pattern in C#. The following are the common characteristics of a Singleton Pattern.

A single constructor, that is private and parameterless.

The class is sealed.

A static variable that holds a reference to the single created instance, if any.

A public static means of getting the reference to the single created instance, creating one if necessary.

Advantages of Singleton Pattern

The advantages of a Singleton Pattern are:

Singleton pattern can be implemented interfaces.

It can be also inherit from other classes.

It can be lazy loaded.

It has Static Initialization.

It can be extended into a factory pattern.

It helps to hide dependencies.

It provides a single point of access to a particular instance, so it is easy to maintain.

Disadvantages of Singleton Pattern

The disadvantages of a Singleton Pattern are:

Unit testing is more difficult (because it introduces a global state into an application).

This pattern reduces the potential for parallelism within a program, because to access the singleton in a multi-threaded system, an object must be serialized (by locking).

Singleton class vs. Static methods

The following conpares Singleton class vs. Static methods:

A Static Class cannot be extended whereas a singleton class can be extended.

A Static Class can still have instances (unwanted instances) whereas a singleton class prevents it.

A Static Class cannot be initialized with a STATE (parameter), whereas a singleton class can be.

A Static class is loaded automatically by the CLR when the program or namespace containing the class is loaded.

How to Implement Singleton Pattern in your code

There are many way to implement a Singleton Pattern in C#.

No Thread Safe Singleton.

Thread-Safety Singleton.

Thread-Safety Singleton using Double-Check Locking.

Thread-Safe Singleton without using locks and no lazy instantiation.

Fully lazy instantiation.

Using .NET 4's Lazy<T> type.

**1. No Thread Safe Singleton**

Explanation of the following code:

The following code is not thread-safe.

Two different threads could both have evaluated the test (if instance == null) and found it to be true, then both creates instances, which violates the singleton pattern.

Note that in fact the instance may already have been created before the expression is evaluated, but the memory model doesn't guarantee that the new value of instance will be seen by other threads unless suitable memory barriers have been passed.

**public** **sealed** **class** Singleton

{

    //Private Constructor.

**private** Singleton()

    {

    }

**private** **static** Singleton instance = **null**;

**public** **static** Singleton Instance

    {

**get**

        {

**if** (instance == **null**)

            {

                instance = **new** Singleton();

            }

**return** instance;

        }

    }

}

**2. Thread Safety Singleton**

Explanation of the following code:

This implementation is thread-safe.

In the following code, the thread is locked on a shared object and checks whether an instance has been created or not.

This takes care of the memory barrier issue and ensures that only one thread will create an instance.

For example: Since only one thread can be in that part of the code at a time, by the time the second thread enters it, the first thread will have created the instance, so the expression will evaluate to false.

The biggest problem with this is performance; performance suffers since a lock is required every time an instance is requested.

**public** **sealed** **class** Singleton

{

    Singleton()

    {

    }

**private** **static** **readonly** **object** padlock = **new** **object**();

**private** **static** Singleton instance = **null**;

**public** **static** Singleton Instance

    {

**get**

        {

**lock** (padlock)

            {

**if** (instance == **null**)

                {

                    instance = **new** Singleton();

                }

**return** instance;

            }

        }

    }

}

**3. Thread Safety Singleton using Double Check Locking**

Explanation of the following code:

In the following code, the thread is locked on a shared object and checks whether an instance has been created or not with double checking.

**public** **sealed** **class** Singleton

{

    Singleton()

    {

    }

**private** **static** **readonly** **object** padlock = **new** **object**();

**private** **static** Singleton instance = **null**;

**public** **static** Singleton Instance

    {

**get**

        {

**if** (instance == **null**)

            {

**lock** (padlock)

                {

**if** (instance == **null**)

                    {

                        instance = **new** Singleton();

                    }

                }

            }

**return** instance;

        }

    }

}

**Façade:**

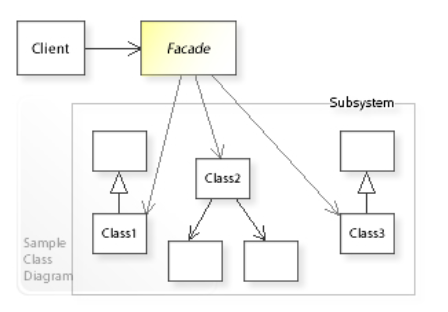
Facade is a **structural design pattern** that provides a simplified interface to a library, a framework, or any other complex set of classes.

If we try to understand this in simpler terms, then we can say that a room is a façade and just by looking at it from outside the door, one cannot predict what is inside the room and how the room is structured from inside. Thus, Façade is a general term for simplifying the outward appearance of a complex or large system.

In software terms, Facade pattern hides the complexities of the systems and provides a simple interface to the clients.

This pattern involves one wrapper class which contains a set of methods available for the client. This pattern is particularly used when a system is very complex or difficult to understand and when the system has multiple subsystems.

Let’s see the below UML diagram,

  
Image source: Wikipedia

Here, we can see that the client is calling the Façade class which interacts with multiple subsystems making it easier for the client to interact with them.

However, it is possible that façade may provide limited functionality in comparison to working with the subsystem directly, but it should include all those features which are actually required by the client.

For example, when someone calls the restaurant, suppose, for ordering pizza or some other food, then the operator on behalf of the restaurant gives the voice interface which is actually the façade for their customers.

Customers place their orders just by talking to the operator and they don’t need to bother about how they will prepare the pizza, what all operations will they perform, on what temperature they will cook, etc.

Similarly, in our code sample, we can see that the client is using the restaurant façade class to order pizza and bread of different types without directly interacting with the subclasses.

Now, it's time to dive into the real code.

This is the interface specific to the pizza.

**public** **interface** IPizza {

**void** GetVegPizza();

**void** GetNonVegPizza();

}

This is a pizza provider class which will get pizza for their clients. Here methods can have other private methods which client is not bothered about.

**public** **class** PizzaProvider: IPizza {

**public** **void** GetNonVegPizza() {

        GetNonVegToppings();

        Console.WriteLine("Getting Non Veg Pizza.");

    }

**public** **void** GetVegPizza() {

        Console.WriteLine("Getting Veg Pizza.");

    }

**private** **void** GetNonVegToppings() {

        Console.WriteLine("Getting Non Veg Pizza Toppings.");

    }

}

Similarly, this is the interface specific for the bread.

**public** **interface** IBread {

**void** GetGarlicBread();

**void** GetCheesyGarlicBread();

}

And this is a bread provider class.

**public** **class** BreadProvider: IBread {

**public** **void** GetGarlicBread() {

        Console.WriteLine("Getting Garlic Bread.");

    }

**public** **void** GetCheesyGarlicBread() {

        GetCheese();

        Console.WriteLine("Getting Cheesy Garlic Bread.");

    }

**private** **void** GetCheese() {

        Console.WriteLine("Getting Cheese.");

    }

}

Below is the restaurant façade class, which will be used by the client to order different pizzas or breads.

**public** **class** RestaurantFacade {

**private** IPizza \_PizzaProvider;

**private** IBread \_BreadProvider;

**public** RestaurantFacade() {

        \_PizzaProvider = **new** PizzaProvider();

        \_BreadProvider = **new** BreadProvider();

    }

**public** **void** GetNonVegPizza() {

        \_PizzaProvider.GetNonVegPizza();

    }

**public** **void** GetVegPizza() {

        \_PizzaProvider.GetVegPizza();

    }

**public** **void** GetGarlicBread() {

        \_BreadProvider.GetGarlicBread();

    }

**public** **void** GetCheesyGarlicBread() {

        \_BreadProvider.GetCheesyGarlicBread();

    }

}

Finally, below is the main method of our program,

**void** Main() {

    Console.WriteLine("----------------------CLIENT ORDERS FOR PIZZA----------------------------\n");

    var facadeForClient = **new** RestaurantFacade();

    facadeForClient.GetNonVegPizza();

    facadeForClient.GetVegPizza();

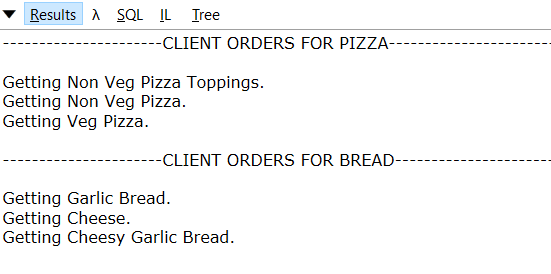
    Console.WriteLine("\n----------------------CLIENT ORDERS FOR BREAD----------------------------\n");

    facadeForClient.GetGarlicBread();

    facadeForClient.GetCheesyGarlicBread();

}

**OUTPUT**



Now, let’s see when we should use this pattern and what could be the real-life scenarios:

**WHEN TO USE THIS PATTERN**

Use this pattern to simplify the problem when there are multiple complex subsystems and interacting with them individually is really difficult/cumbersome.

**REAL LIFE USE CASE**

The shopkeeper is a façade for all the items in the shop.

Online travel portal is a façade for their customers for different holiday/travel packages.

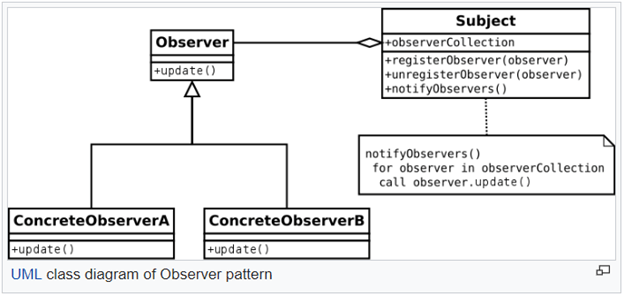
Customer care is a façade for their customers for different services.

**Observer Pattern:**

This pattern falls under the category of behavioral pattern. As the name suggests, it is based on the principle where one object observes another object.

So, in simple words, this pattern defines a one-to-many relationship between subject and observer objects so that if the subject changes its state, then all of the dependent observer objects will be notified of the change.

Let’s understand it in more details using the UML diagram, as shown below.

  
Image source: Wikipedia

**For example**

Consider a technical author like we have in C# Corner, who writes blogs, articles etc. So, if someone likes reading articles from a specific author, one can follow/subscribe the author for future articles. This is a real-life scenario of the Observer pattern.

In this case, all the subscribers who want to subscribe to the author are Observer Objects and the author is a Subject.

Now, let’s dive into the actual code to see how it works.

This is the interface which will be implemented by the subject.

**public** **interface** ISubject {

**void** registerObserver(Observer observer);

**void** unregisterObserver(Observer observer);

**void** notifyObservers();

}

This is the Subject class and as per our example mentioned above, the author will be the Subject. So here, we are assuming that the author has already written 1 article and those who will like the article can subscribe or unsubscribe (if they have already subscribed) accordingly.

Now, whenever the subject changes its state, all of its observers will be notified.

**public** **class** Subject: ISubject {

**private** List < Observer > Observers = **new** List < Observer > ();

**private** **int** articlesCount = 1;

**public** **int** Articles {

**get** {

**return** articlesCount;

        }

**set** {

**if** (value > articlesCount) {

                articlesCount++;

                notifyObservers();

            }

        }

    }

**public** **void** registerObserver(Observer observer) {

        Observers.Add(observer);

    }

**public** **void** unregisterObserver(Observer observer) {

        Observers.Remove(observer);

    }

**public** **void** notifyObservers() {

**foreach**(var observer **in** Observers) {

            observer.Update();

        }

    }

}

This is the interface which will be implemented by all the Observer objects. As per our example, all the subscribers/followers for the author will be the Observer objects.

**public** **interface** IObserver {

**void** Update();

}

This is the Observer class which can be used to create different observers.

**public** **class** Observer: IObserver {

**public** **string** ObserverName;

**public** Observer(**string** name) {

        ObserverName = name;

    }

**public** **void** Update() {

        //Observer can update his system accordingly

        Console.WriteLine("Hello " + ObserverName + ", a new article has been published by the author.");

    }

}

Finally, this is the main method which can be used for running this program.

**void** Main() {

    var subject = **new** Subject();

    var observerA = **new** Observer("Observer A");

    var observerB = **new** Observer("Observer B");

    var observerC = **new** Observer("Observer C");

    Console.WriteLine("Intially suppose Subject has already written total " + subject.Articles + " article");

    Console.WriteLine("\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

    Console.WriteLine("Registering observers A and B for future articles...............");

    subject.registerObserver(observerA);

    subject.registerObserver(observerB);

    Console.WriteLine("New article published by Subject,so now observers A and B will be notified....\n");

    subject.Articles++;

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

    Console.WriteLine("Registering observer C for future articles and unregistering observer B from the future articles...............");

    subject.registerObserver(observerC);

    subject.unregisterObserver(observerB);

    Console.WriteLine("New article published by Subject,so now observers A and C will be notified....\n");

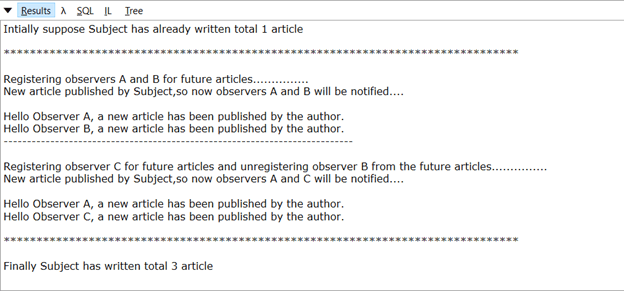
    subject.Articles++;

    Console.WriteLine("\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

    Console.WriteLine("Finally Subject has written total " + subject.Articles + " article \n");

}

**OUTPUT**



**When should we use this**

We should use it when multiple objects are dependent on the state of one object.

**Some common Use Cases**

Following someone on Instagram/Twitter and other such platforms.

App users gets notified for the updates.

Email Subscriptions and many more.

**Inversion of Control:**

public class clsDAL  
{  
    private clsSqlServer \_sql;   
   
    public clsDAL()  
    {  
        \_sql = new clsSqlServer();   
    }  
}

Consider the above example, where we have a DAL class. The default constructor of the DAL class creates an object of the SqlServer class. That means the DAL Class is responsible for creating an object of the SqlServer class. So there is tight coupling between the DAL class and SqlServer class.  
  
Three main problems in the above code:

The DAL class is responsible for creating an object of the SqlServer class.

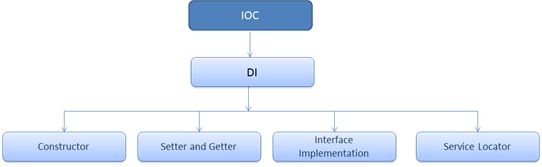
The SqlServer class is directly referenced in the DAL class.

The DAL class should be aware of the SqlServer class type.

Now we understand the problem. Now let's discuss the solution. The solution is to shift the object creation part from this class. We need to shift the object creation control from here i.e. Inversion of control.  
  
**Principles of IOC**

Main classes aggregating other classes should not depend on the direct implementation of the aggregated classes. Both the classes should depend on abstraction.

Abstraction should not depend on details, details should depend on abstraction.

**What is Dependency Injection?**  
  
Inversion of control is implemented by dependency injection because Inversion of control is a principle and dependency injection is a way of implementing IOC.  
  
**Ways of implementing IOC**  
  
Now let's discuss each method with an example.  
  
**Constructor Methodology**

public class clsDAL

{

    private ISql \_sql;    public clsDAL(ISql obj)

    {

         \_sql = obj;

    }

}

In this methodology we pass an object of SQL into the DAL class. Here in the above code you can see that there is a parameterized constructor in the DAL class. And the parameterized constructor accepts an object of SQL. The DAL class is not responsible for creating an object of SQL in this case. So there is no tight coupling between these classes. This method is not useful for the client who only can use a default constructor.  
  
**Setter and Getter**

public class clsDAL

{

    private ISql \_sql;    public Isql Sql

    {

        set

        {

             \_sql = value;

        }

    }

}

In this method we expose an object of SQL through the get/set methods of the DAL class. But it violates the encapsulation rule of OOP. Encapsulation means hiding internal details of an object. So here rather than hiding an object, we are exposing an object.  
  
**Interface Implementation**

interface ISqlDI

{

    void setConnection(ISql obj);

}

public class clsDAL : ISqlDI

{

    private ISql \_sql;    public void setConnection(ISql obj)

    {

        \_sql = obj;

    }

}

In the preceding code we have implemented an interface which has a setConnection method which sets the SQL object. And the DAL class implements a SQL interface. So with the help of the setConnection method the client can inject a SQL object in the DAL class.  
  
**Server Locator**  
  
static class LocateConnection  
{  
    public static ISql getConnection() { }  
}

 interface clsDAL

{

    private ISql \_sql;

    public clsDAL()

    {

        \_sql = LocateConnection.getConnection();

    }

}

In this method we create a static class and a static method inside this class. The DAL class calls this static method from its default constructor. So in this way the SQL object is injected into the DAL class.

Dependency Injection in C#

Dependency Injection (DI) is a software design pattern. It allows us to develop loosely-coupled code. The intent of Dependency Injection is to make code maintainable. Dependency Injection helps to reduce the tight coupling among software components. Dependency Injection reduces the hard-coded dependencies among your classes by injecting those dependencies at run time instead of design time technically. This article explains how to implement Dependency Injection in C# and .NET code.

We have the following ways to implement Dependency Injection.

Constructor Injection

This is the most commonly used dependency pattern in Object Oriented Programming. The constructor injection normally has only one parameterized constructor, so in this constructor dependency there is no default constructor and we need to pass the specified value at the time of object creation. We can use the injection component anywhere within the class. It addresses the most common scenario where a class requires one or more dependencies.

The following is an example:

**using** System;

**using** System.Collections.Generic;

**using** System.Linq;

**using** System.Text;

**using** System.Threading.Tasks;

**namespace** propertyinjuction

{

**public** **interface** text

    {

**void** print();

    }

**class** format : text

    {

**public** **void** print()

        {

            Console.WriteLine(" here is text format");

        }

    }

    // constructor injection

**public** **class** constructorinjection

    {

**private** text \_text;

**public** constructorinjection(text t1)

        {

**this**.\_text = t1;

        }

**public** **void** print()

        {

            \_text.print();

        }

    }

**class** constructor

    {

**static** **void** Main(**string**[] args)

        {

            constructorinjection cs = **new** constructorinjection(**new** format());

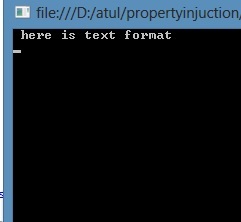
            cs.print();

            Console.ReadKey();

        }

    }

}



By passing the services that implemented the text interface the builder assembled the dependencies.

Property Injection

We use constructor injection, but there are some cases where I need a parameter-less constructor so we need to use property injection.

 The following is an example:

**public** **interface** INofificationAction

{

**void** ActOnNotification(**string** message);

}

**class** atul     {

       INofificationAction task = **null**;

**public** **void** notify(INofificationAction  at ,**string** messages)

       {

**this**.task = at;

       task.ActOnNotification(messages);

       }

   }

**class** EventLogWriter : INofificationAction

   {

**public** **void** ActOnNotification(**string** message)

       {

           // Write to event log here

       }

   }

**class** Program

   {

**static** **void** Main(**string**[] args)

       {

           //services srv = new services();

           //other oth = new other();

           //oth.run();

           //Console.WriteLine();

           EventLogWriter elw = **new** EventLogWriter();

           atul at = **new** atul();

           at.notify(elw, "to logg");

           Console.ReadKey();

       }

   }

You cannot control when the dependency is set at all, it can be changed at any point in the object's lifetime.

Method Injection

In method injection we need to pass the dependency in the method only. The entire class does not need the dependency, just the one method. I have a class with a method that has a dependency. I do not want to use constructor injection because then I would be creating the dependent object every time this class is instantiated and most of the methods do not need this dependent object.

The following is an example:

**using** System;

**using** System.Collections.Generic;

**using** System.Linq;

**using** System.Text;

**using** System.Threading.Tasks;

**namespace** propertyinjuction

{

**public** **interface** Iset

    {

**void** print();

    }

**public** **class** servic : Iset

    {

**public** **void** print()

        {

            Console.WriteLine("print........");

        }

    }

**public** **class** client

    {

**private** Iset \_set;

**public** **void** run(Iset serv)

        {

**this**.\_set = serv;

            Console.WriteLine("start");

**this**.\_set.print();

        }

    }

**class** method

    {

**public** **static** **void** Main()

        {

            client cn = **new** client();

            cn.run(**new** servic());

            Console.ReadKey();

        }

    }

}

**Abstract Factory Design Pattern In C#**

Abstract Factory pattern also falls under Creational Pattern of Gang of Four (GoF) Design Patterns.

**Background**

Before talking about Abstract Factory pattern, I just want to share a little about "Gang of Four (GoF)" to which the Abstract Factory pattern belongs.

Who are the Gang of Four?

The Gang of Four are the authors of the book, "Design Patterns: Elements of Reusable Object-Oriented Software". This important book describes various development techniques and pitfalls in addition to providing 23 object-oriented programming design patterns. The four authors are Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides.

Now, let's move on to the Abstract Factory design pattern.

In this article, I have tried to cover the concept of Abstract Factory design pattern and the ways of implementing Abstract Factory design pattern.

**What is it?**

An interface for creating families of related or dependent objects without specifying their concrete classes. We can say it is just an object maker which can create more than one type of object.

The object it produces is known to the client only by that object's interface, not by the object's actual concrete implementation.

**When to use it?**

We use it when we have a requirement to create a set of related objects, or dependent objects which must be used together as families of objects. Concrete classes should be decoupled from clients.

How does it differ from Factory Method?

First of all, both of them fall under Creational category and it means both will solve the problem relating to object creation. Factory Method and Abstract Factory design pattern are about creating objects.

Factory Method Design Pattern

Here, we define an interface which will expose a method which will create objects for us. Return type of that method is never a concrete type; rather, it will be some interface (or may be an abstract class).

Creates object through inheritance

Produce only one product

Implements code in the abstract creator that makes use of the concrete type that sub class produces

Abstract Factory Design Pattern

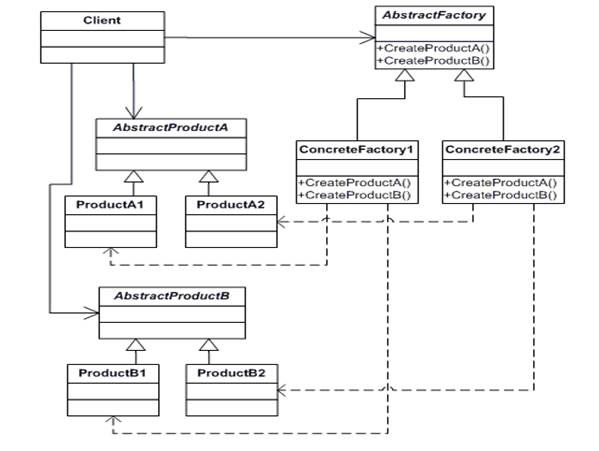
Here, we define an interface which will create families of related or dependent objects. In simple words, interface will expose multiple methods each of which will create some object. Again, here method return types will be generic interfaces. All these objects will together become part of some important functionality.

Creates object through composition

Produce families of products

Concrete factories implements factory method to create product

UML Class Diagram

****

The classes and objects participating in the above UML class diagram are as follow.

AbstractFactory  
This is an interface for operations which is used to create abstract product.

ConcreteFactory  
This is a class which implements the AbstractFactory interface operations to create concrete products.

AbstractProduct  
This declares an interface for a type of product object

Product  
This defines a product object to be created by the corresponding concrete factory also implements the AbstractProduct interface

Client  
This is a class which uses AbstractFactory and AbstractProduct interfaces to create a family of related objects.

Now, let’s understand this with a real world example,

The example here has an implementation of an Abstract Factory as an Interface IMobilePhone that has methods that can create a Smart Phone object and a Normal Phone object. The client codes against IMobilePhone and gets ISmartPhone and INormalPhone interfaces.

In case of "Nokia", it creates a family of Nokia objects (SmartPhone and NormalPhone) and in case of "Samsung", creates a family of Samsung objects (SmartPhone and NormalPhone).

The client doesn't care which object (Nokia SmartPhone and NormalPhone or Samsung SmartPhone and NormalPhone), IMobilePhone interface returns as it codes against ISmartPhone and INormalPhone interface.

**Memento Design Pattern Using C#:**

**What is the Memento Pattern?**  
According to the GoF's definition, what this pattern does is:  
  
Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.  
  
**A Real world example**  
Let's use an example of a computer game, where we have a player who will need to cross levels 1 to 5 to complete the game and will score different scores at each level. We will create a checkpoint for this player at level 1, with some initial score at a specific instance of time, and store a snapshot of its data. Then we will update the score, level and time of the player. Now with some business rule, say the player looses a life, he will again need to start from that checkpoint. So when he starts again from the checkpoint, we will restore the snapshot data and proceed further accordingly. In order to do this, we will divide the system into the following components.

**Originator or MainClass**This is the main class of the system that maintains the data of the player, in other words its level, score and so on and for which we want to create a snapshot of the data. In our case, it will be a PlayerStatistics class, with 3 properties named Level, Score and CheckPointTime. These are the properties or the data of the player, for which we want to create a snapshot.

**Memento**This class is used to store the snapshot of the Originator class data. In our case, it will be the CheckPointMemento class, only with properties, for which we want to store the data, in other words for each of the properties that we want to have in the snapshot, we create a property in the memento class.

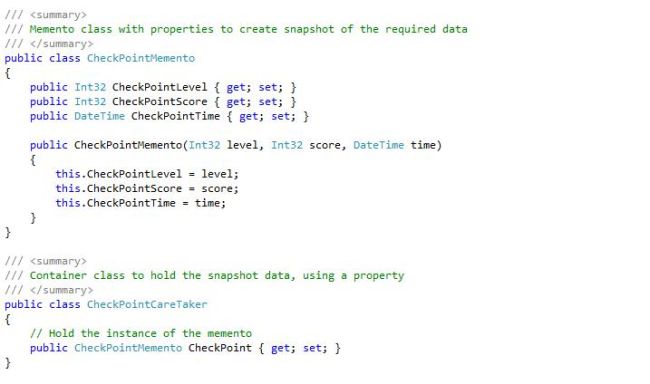
**CareTakerClass**This is like a container class, that holds the Memento class instance, that further holds the snapshot of the original data. In our case, it will be CheckPointCareTaker, and will hold the memento instance using a property defined in it.

So, to start with, we create the following classes:

A PlayerStatistics class and its properties.

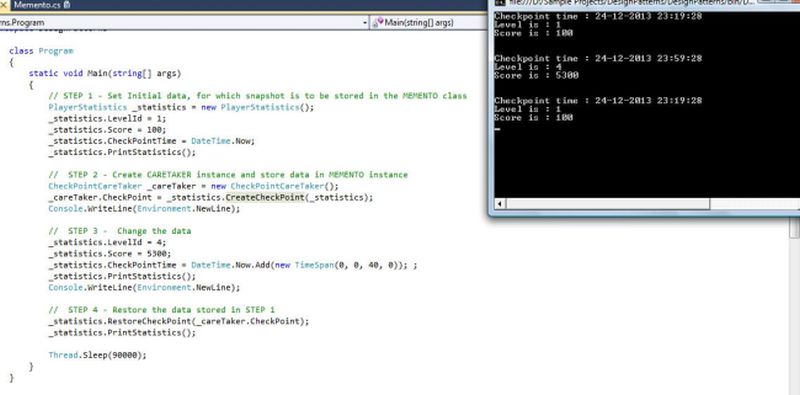
A CheckPointMemento class, with the properties for which we want to store the snapshot data of the PlayerStatistics class.

A CheckPointCareTakerclass, that will act as a container for CheckPointMemento class.

See the classes below. These are the memento and caretaker classes.  
  
  
  
Now we will create the main class PlayerStatistics and an important point to note here is that the PlayerStatistics class will have two methods, apart from the properties to hold the player data. These methods are :

**CreateCheckPoint()**This method will be used to return the instance of the memento that is storing the data of which we would like to create a snapshot. The instance returned by this method will be added to the container or the CheckPointCareTaker class, by setting the property.

**RestoreCheckPoint()**This method will receive the memento class instance or you can say the snapshot data that we added at the checkpoint. The data received will then be returned to the original properties.

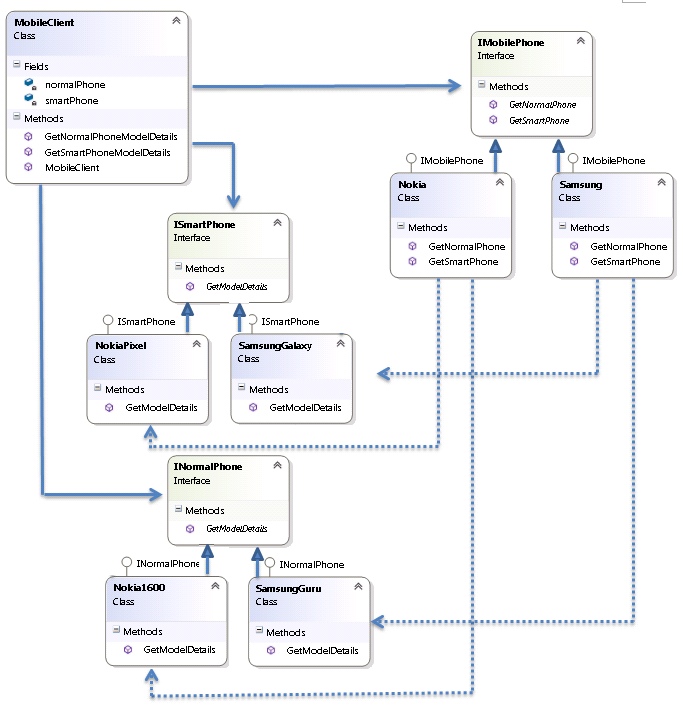
See the complete code for the Main/originator class, with the properties and the two methods explained above.  
  
  
  
Now we have set up the initial code. It's time to write the client code. So we start by setting some initial data for the player. Then we will store the data in a memento instance and add it to the caretaker container class. Further we will change the data and then finally retrieve the stored data from the memento instance. So our code will be like the following:  
  
  
  
**How is it working ?**  
Let's explain the client code now.

In Step 1, we simply set the properties of the player data and print them.

In Step 2, we call the CreateCheckPoint() method in the main class, and generate the instance of the memento class type. This instance is then assigned to the CheckPointCareTakerclass, that acts as a container here to hold the snapshot data.

In Step 3, we simply update the data and print it, like we did in Step 1.

In Step 4, we basically perform the undo operation and retrieve the original data that we had at the start. This is done by calling the RestoreCheckPoint method in the main class and passing it the data from the container or caretaker class. The data received is then set back to the properties and printed again.



**Who is what?**

The classes and objects participating in the above class diagram can be identified as shown below.

AbstractFactory- IMobilePhone

ConcreteFactory - Nokia, Samsung

AbstractProduct- ISmartPhone, INormalPhone

Product- NokiaPixel, Nokia1600, SamsungGalaxy, SamsungGuru

Client- MobileClient

**Here are the code blocks for each participant**

AbstractFactory

namespace AbstractFactoryDesignPatternInCSharp

{

    /// <summary>

    /// The 'AbstractFactory' interface.

    /// </summary>

**interface** IMobilePhone

    {

        ISmartPhone GetSmartPhone();

        INormalPhone GetNormalPhone();

    }

}

ConcreteFactoryNokia

namespace AbstractFactoryDesignPatternInCSharp

{

    /// <summary>

    /// The 'ConcreteFactory1' class.

    /// </summary>

**class** Nokia : IMobilePhone

    {

**public** ISmartPhone GetSmartPhone()

        {

**return** **new** NokiaPixel();

        }

**public** INormalPhone GetNormalPhone()

        {

**return** **new** Nokia1600();

        }

    }

}

Samsung

namespace AbstractFactoryDesignPatternInCSharp

{

    /// <summary>

    /// The 'ConcreteFactory2' class.

    /// </summary>

**class** Samsung : IMobilePhone

    {

**public** ISmartPhone GetSmartPhone()

        {

**return** **new** SamsungGalaxy();

        }

**public** INormalPhone GetNormalPhone()

        {

**return** **new** SamsungGuru();

        }

    }

}

AbstractProductISmartPhone

namespace AbstractFactoryDesignPatternInCSharp

{

    /// <summary>

    /// The 'AbstractProductA' interface

    /// </summary>

**interface** ISmartPhone

    {

        string GetModelDetails();

    }

}

INormalPhone

namespace AbstractFactoryDesignPatternInCSharp

{

    /// <summary>

    /// The 'AbstractProductB' interface

    /// </summary>

**interface** INormalPhone

    {

        string GetModelDetails();

    }

}

ProductNokiaPixel

namespace AbstractFactoryDesignPatternInCSharp

{

    /// <summary>

    /// The 'ProductA1' class

    /// </summary>

**class** NokiaPixel : ISmartPhone

    {

**public** string GetModelDetails()

        {

**return** "Model: Nokia Pixel\nRAM: 3GB\nCamera: 8MP\n";

        }

    }

}

SamsungGalaxy

**namespace** AbstractFactoryDesignPatternInCSharp

{

    /// <summary>

    /// The 'ProductA2' class

    /// </summary>

**class** SamsungGalaxy : ISmartPhone

    {

**public** **string** GetModelDetails()

        {

**return** "Model: Samsung Galaxy\nRAM: 2GB\nCamera: 13MP\n";

        }

    }

}  

Nokia1600

namespace AbstractFactoryDesignPatternInCSharp

{

    /// <summary>

    /// The 'ProductB1' class

    /// </summary>

**class** Nokia1600 : INormalPhone

    {

**public** string GetModelDetails()

        {

**return** "Model: Nokia 1600\nRAM: NA\nCamera: NA\n";

        }

    }

}

SamsungGuru

namespace AbstractFactoryDesignPatternInCSharp

{

    /// <summary>

    /// The 'ProductB2' class

    /// </summary>

**class** SamsungGuru : INormalPhone

    {

**public** string GetModelDetails()

        {

**return** "Model: Samsung Guru\nRAM: NA\nCamera: NA\n";

        }

    }

}

Client

namespace AbstractFactoryDesignPatternInCSharp

{

    /// <summary>

    /// The 'Client' class

    /// </summary>

**class** MobileClient

    {

        ISmartPhone smartPhone;

        INormalPhone normalPhone;

**public** Client(IMobilePhone factory)

        {

            smartPhone = factory.GetSmartPhone();

            normalPhone = factory.GetNormalPhone();

        }

**public** string GetSmartPhoneModelDetails()

        {

**return** smartPhone.GetModelDetails();

        }

**public** string GetNormalPhoneModelDetails()

        {

**return** normalPhone.GetModelDetails();

        }

    }

}

Factory Pattern Client Demo

using System;

namespace AbstractFactoryDesignPatternInCSharp

{

    /// <summary>

    /// Abstract Factory Pattern Demo

    /// </summary>

**class** Program

    {

**static** **void** Main()

        {

            IMobilePhone nokiaMobilePhone = **new** Nokia();

            MobileClient nokiaClient = **new** MobileClient(nokiaMobilePhone);

            Console.WriteLine("\*\*\*\*\*\*\*\*\* NOKIA \*\*\*\*\*\*\*\*\*\*");

            Console.WriteLine(nokiaClient.GetSmartPhoneModelDetails());

            Console.WriteLine(nokiaClient.GetNormalPhoneModelDetails());

            IMobilePhone samsungMobilePhone = **new** Samsung();

            MobileClient samsungClient = **new** MobileClient(samsungMobilePhone);

            Console.WriteLine("\*\*\*\*\*\*\* SAMSUNG \*\*\*\*\*\*\*\*\*\*");

            Console.WriteLine(samsungClient.GetSmartPhoneModelDetails());

            Console.WriteLine(samsungClient.GetNormalPhoneModelDetails());

            Console.ReadKey();

        }

    }

}

In Object Oriented Programming (OOP), SOLID is an acronym, introduced by Michael Feathers, for five design principles used to make software design more understandable, flexible, and maintainable. These principles are a subset of many principles promoted by Robert C. Martin.

SOLID Principles

There are five SOLID principles:

Single Responsibility Principle (SRP)

Open Closed Principle (OCP)

Liskov Substitution Principle (LSP)

Interface Segregation Principle (ISP)

Dependency Inversion Principle (DIP)

Single Responsibility Principle (SRP)

**Definition:** A class should have only one reason to change.

In layman terminology, this means that a class should not be loaded with multiple responsibilities and a single responsibility should not be spread across multiple classes or mixed with other responsibilities. The reason is that more changes requested in the future, the more changes the class need to apply.

**Understanding**

Single Responsibility Principle is one of the five SOLID principles which guide developers as they write code or design an application.

In simple terms, a module or class should have a very small piece of responsibility in the entire application. Or as it states, a class/module should have not more than one reason to change.

If a class has only a single responsibility, it is likely to be very robust. It’s easy to verify its working as per logic defined. And it’s easy to change in class as it has single responsibility.

The Single Responsibility Principle provides another benefit. Classes, software components and modules that have only one responsibility are much easier to explain, implement and understand than ones that give a solution for everything.

This also reduces number of bugs and improves development speed and most importantly makes developer’s life lot easier.

**Implementation**

Let’s take a scenario of Garage service station functionality. It has 3 main functions; open gate, close gate and performing service. Below example violates SRP principle. The code below, violates SRP principle as it mixes open gate and close gate responsibilities with the main function of servicing of vehicle.

Public class GarageStation

{

Public void DoOpenGate()

{

*//Open the gate functinality*

}

Public void PerformService(Vehicle vehicle)

{

*//Check if garage is opened*

*//finish the vehicle service*

}

Public void DoCloseGate()

{

*//Close the gate functinality*

}

}

We can correctly apply SRP by refactoring of above code by introducing interface. A new interface called IGarageUtility is created and gate related methods are moved to different class called GarageStationUtility.

Public class GarageStation

{

IGarageUtility \_garageUtil;

Public GarageStation(IGarageUtility garageUtil)

{

this.\_garageUtil=garageUtil;

}

Public void OpenForService()

{

\_garageUtil.OpenGate();

}

Public void DoService()

{

*//Check if service station is opened and then*

*//finish the vehicle service*

}

Public void CloseGarage()

{

\_garageUtil.CloseGate();

}

}

Public class GarageStationUtility : IGarageUtility

{

Public void OpenGate()

{

*//Open the Garage for service*

}

Public void CloseGate()

{

*//Close the Garage functionlity*

}

}

public interface IGarageUtility

{

voidOpenGate();

voidCloseGate();

}

Open Closed Principle (OCP)

**Definition:** Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.

Bertrand Meyer is generally credited for having originated the definition of open/closed principle in his book Object-Oriented Software Construction.

**Understanding**

This principle suggests that the class should be easily extended but there is no need to change its core implementations.

The application or software should be flexible to change. How change management is implemented in a system has a significant impact on the success of that application/ software. The OCP states that the behaviors of the system can be extended without having to modify its existing implementation.

New features should be implemented using the new code, but not by changing existing code. The main benefit of adhering to OCP is that it potentially streamlines code maintenance and reduces the risk of breaking the existing implementation.

**Implementation**

Let’s take an example of bank accounts like regular savings, salary saving, corporate etc. for different customers. As for each customer type, there are different rules and different interest rates. The code below violates OCP principle if the bank introduces a new Account type. Said code modifies this method for adding a new account type.

Public class Account

{

*// members and function declaration*

Public decimal CalcInt(AccType accType)

{

If(accType==”Regular”)*// savings*

{

Inte=bal\*0.4;

If(bal<1000)Inte-=bal\*0.2;

If(bal<50000)Inte+=amt\*0.4;

}

elseif(accType==”Salary”)*// salary savings*

{

inte=bal\*0.5;

}

elseif(accType==”Corporate”)*// Corporate*

{

inte=bal\*0.3;

}

}

}

We can apply OCP by using interface, abstract class, abstract methods and virtual methods when you want to extend functionality. Here I have used interface for example only but you can go as per your requirement.

Interface IAccount

{

*// members and function declaration, properties*

decimal CalcInt();

}

Public Class RegularSavingAcc : IAccount

{

*//regular savings account*

Public decimal CalcInt()

{

Inte=bal\*0.4;

If(bal<1000)inte-=bal\*0.2;

If(bal<50000)inte+=amt\*0.4;

}

}

Public Class SalarySavingAcc : IAccount

{*//Salary savings account*

Public decimal CalcInt()

{

Inte=bal\*0.5;

}

}

Public Class CorporateAcc : IAccount

{

Public decimal CalcInt()

{

Inte=bal\*0.3;

}

}

In the above code three new classes are created; regular saving account, SalarySavingAccount, and CorporateAccount, by extending them from IAccount.

This solves the problem of modification of class and by extending interface, we can extend functionality.

Above code is implementing both OCP and SRP principle, as each class has single is doing a single task and we are not modifying class and only doing an extension.

Liskov Substitution Principle (LSP)

**Definition by Robert C. Martin:** Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.

The **Liskov substitution principle (LSP)** is a definition of a subtyping relation, called **(strong) behavioral subtyping,** that was initially introduced by Barbara Liskov in a 1987 conference keynote address titled **Data abstraction and hierarchy.**

**Understanding**

LSP states that the child class should be perfectly substitutable for their parent class. If class C is derived from P then C should be substitutable for P.

We can check using LSP that inheritance is applied correctly or not in our code.

LSP is a fundamental principle of SOLID Principles and states that if program or module is using base class then derived class should be able to extend their base class without changing their original implementation.

**Implementation**

Let’s consider the code below where LSP is violated. We cannot simply substitute a Triangle, which results in printing shape of a triangle, with Circle.

namespaceDemo

{

publicclassProgram

{

staticvoidMain(string[]args)

{

Triangletriangle=newCircle();

Console.WriteLine(triangle.GetColor());

}

}

publicclassTriangle

{

publicvirtual string GetShape()

{

return" Triangle ";

}

}

publicclassCircle:Triangle

{

public override string GetShape()

{

return"Circle";

}

}

}

To correct above implementation, we need to refactor this code by introducing interface with method called GetShape.

namespaceDemo

{

classProgram

{

staticvoidMain(string[]args)

{

Shapeshape=newCircle();

Console.WriteLine(shape.GetShape());

shape =newTriangle();

Console.WriteLine(shape.GetShape());

}

}

public abstract classShape

{

public abstract string GetShape();

}

publicclassTriangle:Fruit

{

public override string GetShape()

{

return"Triangle";

}

}

publicclassCircle:Triangle

{

public override string GetShape()

{

return"Circle";

}

}

}

Interface Segregation Principle (ISP)

**Definition:** No client should be forced to implement methods which it does not use, and the contracts should be broken down to thin ones.

**Understanding**

Interface segregation principle is required to solve the design problem of the application. When all the tasks are done by a single class or in other words, one class is used in almost all the application classes then it has become a fat class with overburden. Inheriting such class will results in having sharing methods which are not relevant to derived classes but it’s there in the base class so that will inherit in the derived class.

Using ISP, we can create separate interfaces for each operation or requirement rather than having a single class to do the same work.

**Implementation**

In below code, ISP is broken as process method is not required by OfflineOrder class but is forced to implement.

public interface IOrder

{

Void AddToCart();

Void CCProcess();

}

publicclassOnlineOrder:IOrder

{

publicvoidAddToCart()

{

*//Do Add to Cart*

}

publicvoidCCProcess()

{

*//process through credit card*

}

}

publicclassOfflineOrder:IOrder

{

publicvoidAddToCart()

{

*//Do Add to Cart*

}

publicvoidCCProcess()

{

*//Not required for Cash/ offline Order*

thrownewNotImplementedException();

}

}

We can resolve this violation by dividing IOrder Interface.

public interface IOrder

{

voidAddToCart();

}

public interface IOnlineOrder

{

voidCCProcess();

}

publicclassOnlineOrder:IOrder,IOnlineOrder

{

Public void AddToCart()

{

*//Do Add to Cart*

}

Public void CCProcess()

{

*//process through credit card*

}

}

Public class OfflineOrder:IOrder

{

publicvoidAddToCart()

{

*//Do Add to Cart*

}

}

Dependency Inversion Principle (DIP)

This principle is about dependencies among components. The definition of DIP is given by Robert C. Martin is as follows:

High-level modules should not depend on low-level modules. Both should depend on abstractions.

Abstractions should not depend on details. Details should depend on abstractions.

**Understanding**

The principle says that high-level modules should depend on abstraction, not on the details, of low-level modules. In simple words, the principle says that there should not be a tight coupling among components of software and to avoid that, the components should depend on abstraction.

The terms Dependency Injection (DI) and Inversion of Control (IoC) are generally used as interchangeably to express the same design pattern. The pattern was initially called IoC, but Martin Fowler (known for designing the enterprise software) anticipated the name as DI because all frameworks or runtime invert the control in some way and he wanted to know which aspect of control was being inverted.

Inversion of Control (IoC) is a technique to implement the Dependency Inversion Principle in C#. Inversion of control can be implemented using either an abstract class or interface. The rule is that the lower level entities should join the contract to a single interface and the higher-level entities will use only entities that are implementing the interface. This technique removes the dependency between the entities.

Note:

In below implementation, I have used interface as a reference, but you can use abstract class or interface as per your requirement.

**Implementation**

In below code, we have implemented DIP using IoC using injection constructor. There are different ways to implement Dependency injection. Here, I have use injection thru constructor but you inject the dependency into class's constructor (Constructor Injection), set property (Setter Injection), method (Method Injection), events, index properties, fields and basically any members of the class which are public.

public interface IAutomobile

{

voidIgnition();

voidStop();

}

publicclassJeep:IAutomobile

{

*#region IAutomobile Members*

publicvoidIgnition()

{

Console.WriteLine("Jeep start");

}

publicvoidStop()

{

Console.WriteLine("Jeep stopped.");

}

*#endregion*

}

publicclass SUV :IAutomobile

{

*#region IAutomobile Members*

publicvoidIgnition()

{

Console.WriteLine("SUV start");

}

publicvoidStop()

{

Console.WriteLine("SUV stopped.");

}

*#endregion*

}

publicclassAutomobileController

{

IAutomobile m\_Automobile;

publicAutomobileController(IAutomobile automobile)

{

this.m\_Automobile= automobile;

}

publicvoidIgnition()

{

m\_Automobile.Ignition();

}

publicvoidStop()

{

m\_Automobile.Stop();

}

}

classProgram

{

staticvoidMain(string[]args)

{

IAutomobile automobile =newJeep();

*//IAutomobile automobile = new SUV();*

AutomobileController automobileController=newAutomobileController(automobile);

automobile.Ignition();

automobile.Stop();

Console.Read();

}

}

In the above code, IAutomobile interface is in an abstraction layer and AutomobileController as the higher-level module. Here, we have integrated all in a single code but in real-world, each abstraction layer is a separate class with additional functionality. Here products are completely decoupled from the consumer using IAutomobile interface. The object is injected into the constructor of the AutomobileController class in reference to the interface IAutomobile. The constructor where the object gets injected is called injection constructor.

DI is a software design pattern that allows us to develop loosely coupled code. Using DI, we can reduce tight coupling between software components. DI also allows us to better accomplish future changes and other difficulties in our software. The purpose of DI is to make code sustainable.

**Difference between Authentication and Authorization**

|  |  |
| --- | --- |
| **Authentication** | **Authorization** |
| User identity is confirmed | Here, the user is given permission to access the system / resources after validation |
| User and user server is verified | Here it is validated if the user is allowed to access via some defined rules |
| Login details, usernames, passwords, OTPs required | Checks the security level and privilege of the user, thus determining what the user can or cannot have access to |
| Data is available via Token IDs | Data provided via Access token |
| User can partially change the authentication details as per the requirement | User cannot modify the Authorization permissions as it is given to a user by the owner/manager of the system, and only has the authority to change it. |

NET Core vs .NET Framework

Microsoft maintains both runtimes for building applications with .NET while sharing many of the same APIs. This shared API is called the .NET Standard.

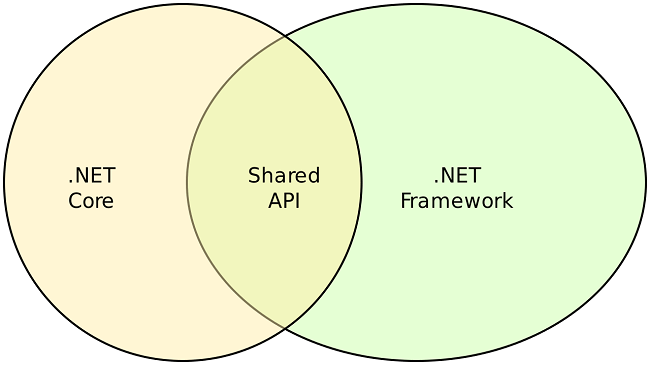


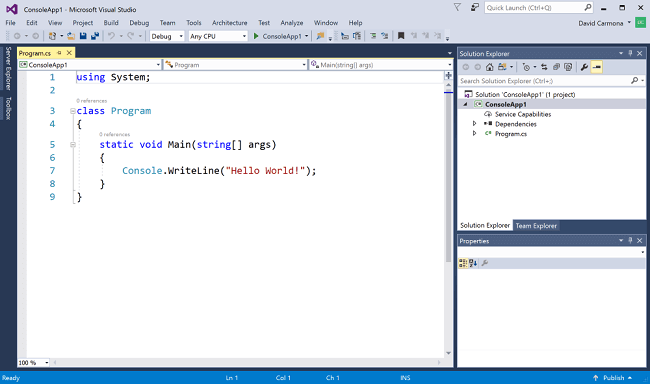
Image via [*Wikipedia*](https://en.m.wikipedia.org/wiki/File:.NET_Framework-Core_relationship.svg)

Developers use the .NET Framework to create Windows desktop and server-based applications. This includes ASP.NET web applications. On the other hand, .NET Core is used to create server applications that run on Windows, Linux and Mac. It does not currently support creating desktop applications with a user interface. Developers can write applications and libraries in VB.NET, C# and F# in both runtimes.

[C# is an object-oriented language](https://docs.microsoft.com/en-us/dotnet/articles/csharp/index) similar to other C-style languages. The learning curve should not be a problem for developers already working with C and similar languages.

**When to Use .NET Core**

A cross-platform and open-source framework, .NET Core is best when developing applications on any platform. .NET Core is used for cloud applications or refactoring large enterprise applications into microservices.



Screenshot via [*Microsoft.com*](https://www.microsoft.com/net/core#windowsvs2017)

You should use .NET Core when:

**There are cross-platform needs.**Use it when the application needs to run across multiple platforms such as Windows, Linux and macOS. Those operating systems are supported as development workstations (and the list of [supported operating systems](https://github.com/dotnet/core/blob/master/roadmap.md) is growing):

Visual Studio is compatible on Windows with a new limited version [on macOS](https://www.visualstudio.com/vs/visual-studio-mac/)

Visual Studio Code can be used on Windows, Linux and macOS

All supported platforms allow the use of the command line

**Using Microservices.** [Microservices](https://stackify.com/what-are-microservices/), a form of service-oriented architecture, are software applications consisting of small, modular business services. Each service can run a unique process, be deployed independently and be created in different programming applications. .NET Core allows a mix of technologies, is lightweight and scalable for each microservice

**Working with Docker containers.**Containers and microservices architecture are often used together. Because it is lightweight and modular, .NET Core works very well with containers. You can deploy cross-platform server apps to Docker containers. .NET Framework works with containers, but the image size is larger

**You have high-performance and scalable system needs.**Microsoft recommends running .NET Core with ASP.NET Core for the best performance and scale. This becomes important when using hundreds of microservices. In such a case, a lower number of servers and virtual machines is best. The efficiency and scalability gained should translate to a better user experience in addition to cost savings

**You are running multiple .NET versions side-by-side.** To install applications with dependencies on different versions of frameworks in .NET, developers need to use .NET Core. Multiple services are executable on the same server with different versions of .NET

**You want command line interface (CLI) control.**Some developers prefer working in lightweight editors and command line control. .NET Core has a CLI for all supported platforms and requires minimal installation on production machines. And, there still is the opportunity to switch to an IDE, such as Visual Studio IDE

**When Not to Use .NET Core**

.NET Core does not have some of the .NET features nor support for all libraries and extensions. As such, you may encounter a few situations in which .NET Core may not be the best option (though continued development will likely eliminate this drawback). Consider the following scenarios:

**Windows Forms and WPF applications are not supported**– You still have to use Mono to make a .NET desktop application for macOS

**ASP.NET WebForms don’t exist** – Though Microsoft provides [strategies for migrating ASP.NET Web Forms apps](https://docs.microsoft.com/en-us/dotnet/architecture/porting-existing-aspnet-apps/migrate-web-forms)

**You need to create a WCF service** – .NET Core does not currently support WCF. Instead, you would need to make a REST API with ASP.NET Core MVC

**Missing 3rd-party library support**– .NET Core provides a compatibility shim between .NET Framework and .NET Core. But, you may still have issues with compatibility if the class library uses any .NET Framework APIs that are not supported (though this will help bridge a lot of class libraries to .NET Core)

**Missing .NET Framework features** – Some .NET Framework functionality is still missing in .NET Core. For example, Entity Framework Core is not the exact same as Entity Framework v6

**You need to access Windows-specific APIs** – If your application needs to work with the Windows Registry (WMI or other Windows specific APIs), it won’t work with .NET Core. It is designed to be more sandboxed away from the OS

**Partial support for VB.NET and F#** – Microsoft and the community continue to work on this but it’s not yet 100%

**Developers Should Use .NET Framework When…**

.NET Framework is distributed with Windows. Generally, it is used to build Windows desktop and large-scale enterprise applications using .NET workflow and data connection tools.

The .NET Framework provides services that include:

Memory management

Type and memory safety

Security

Networking

Application deployment

Data structures

APIs

.NET Framework can be used with Docker and Windows Containers and is most feasible when:

**It is already being used** – Instead of migrating, extend the application. For example, developers can write a new web service in ASP.NET Core

**You’re using third-party libraries or NuGet packages not available in .NET Core** – Despite .NET Core’s popularity, you’ll need to use the .NET Framework when working with libraries that aren’t compatible with .NET Core. [NuGet is the free and open source package manager](https://www.nuget.org/) for .NET and other Microsoft development platforms. The NuGet ecosystem includes client tools that provide the ability to produce and consume packages. It also has a central package repository for package authors and consumers. It is available as a Visual Studio extension

**You’re using technologies not yet available in .NET Core** – .NET Core does not support all .NET Framework technologies. These not-yet-available technologies include:

ASP.NET Web Forms applications (no plans to port)

ASP.NET Web Pages applications (plans to port)

ASP.NET SignalR server/client implementation (plans to port)

WCF services implementation (no plans to migrate, but it is being considered)

Workflow related services (no plans to port) including Windows Workflow Foundation (WF), Workflow Services (WCF + WF in a single service), and WCF Data Services (formerly known as “ADO.NET Data Services”)

Windows Presentation Foundation (WPF) and Windows Forms (no plans to port)

**The platform does not support .NET Core** – Again, not all Microsoft and third-party platforms support it, such as some of Azure’s services. You may encounter some issues even with supported services, which comes with the territory. With .NET Core increasingly gaining traction, it’s becoming easier to find tutorials and workarounds for issues you may encounter. For instance, we encountered a [502.5 Process Failure](https://stackify.com/net-core-azure-app-service-wont-start/) when trying to start an Azure App Service. So, we published a post offering guidance for others who encounter the same issue

**When Not to Run .NET Framework**

There are also a few situations in which you shouldn’t run the .NET Framework. These include when:

Multiple OS platforms are required

High performance and scalability are needed

.NET Core works

Open source framework is required

Key Differences between .Net Core and .Net Framework

Let us discuss some of the major differences between .Net Core vs .Net Framework:

.Net Framework is a software development framework designed and maintained by the tech giant Microsoft. It is Windows-based and primarily runs on Windows devices. It is used for the development of standalone desktop as well as web applications. The framework provides all the basic requirements for the development of applications – UI, DB connectivity, Services, APIs, etc.

.Net Core is an open-source Development Platform designed and maintained by Microsoft and the .Net community. .Net Core has been designed, keeping in mind various needs and purposes, focusing on Web Development, Windows Phone Development, and Windows Store Apps Development.

(Singleton vs Transient vs Scoped)

Scoped

In this service, with every HTTP request, we get a new instance.

The same instance is provided for the entire scope of that request.

eg., if we have a couple of parameter in the controller, both object contains the same instance across the request

This is a better option when you want to maintain a state within a request.

services.AddScoped<IAuthService,AuthService>();

C#

Copy

Chart

Description automatically generated

Transient

A new service instance is created for each object in the HTTP request.

This is a good approach for the multithreading approach because both objects are independent of one another.

The instance is created every time they will use **more memory** and **resources**and can have a **negative** impact on performance

Utilize for the **lightweight** service with little or **no state**.

services.AddTransient<ICronJobService,CronJobService>();

C#

Copy

Graphical user interface, application, table

Description automatically generated

Singleton

Only one service instance was created throughout the lifetime.

Reused the same instance in future, wherever the service is required

Since it's a single lifetime service creation, memory leaks in these services will build up over time.

Also, it has memory efficient as they are created once reused everywhere.

services.AddSingleton<ILoggingService, LoggingService>();

C#

Copy

Graphical user interface, application, table

Description automatically generated

When to use which Service

Singleton approach => We can use this for logging service, feature flag(to on and off module while deployment), and email service

Scoped approach => This is a better option when you want to maintain a state within a request.

Transient approach =>  Use this approach for the lightweight service with little or no state.

Dependency Injection in Asp.Net Core (Singleton vs Transient vs Scoped)

March 10, 2021 [Nitish Kaushik](https://nitishkaushik.com/about)

Dependency Injection in Asp.Net Core is very important for the architecture of the application. Asp.Net Core framework provides built-in support for the Dependency Injection it means there is no need to install any third-party library to implement Dependency Injection.

[View or Download sample code](https://github.com/nitishwebgentle/Dependency-Injection-in-Asp.Net-Core.git)

In this Post, You will learn the Dependency Inject in .Net Core with example.

What is Dependency Injection in Asp.Net Core?

*Dependency Injection is a principle in software architecture that provides a loosely coupled communication between two classes.*

For example the Communication between –

Controller and Repositories

Controller and Services (Email sender, Logger, etc.)

Business layer and Data layer

Etc.

The concept of Dependency Injection is same for Asp.Net Core MVC and Asp.Net Core Web API.

Benefits of using Dependency Injection in Asp.Net Core?

There are lots of benefits of using Dependency Injection in your application –

IOC Implementation – Dependency Injection is used to implement the Inversion Of Control principle.

Dependency Injection provides a Loosely Coupled Communication between two layers (classes).

Writing the Unit Test Cases are super easy with Dependency Injection.

Help to implement SOLID principles as D in SOLID stands for Dependency Injection.

Dependency Injection provides a Flexible Architecture of the application that helps to update the consumed classes without updating the caller classes.

Abstraction – As the Controllers are completely unaware of the implementation of services.

Etc.

How to configure Dependency Injection in Asp.Net Core?

Asp.Net Core provides the built-in support for Dependency Injection.

Dependencies are registered in containers and the container in asp.net core is [IServiceProvider](https://docs.microsoft.com/en-us/dotnet/api/system.iserviceprovider" \t "_blank).

The best place to registers the dependency in the asp.net core is the ConfigureServices method of the Startup class.

public void ConfigureServices(IServiceCollection services)

{

services.AddControllers();

// Register the dependencies here

}

Example of Dependency Injection in Asp.Net Core –

Let’s [create a new Asp.Net Core Web API or MVC application](https://nitishkaushik.com/create-asp-net-core-mvc-application-using-visual-studio-and-cli/).

Add a BooksController in the Controllers folder. Click [here](https://nitishkaushik.com/controller-in-asp-net-core-mvc/) to learn more about Controller in Asp.Net Core.

using Microsoft.AspNetCore.Mvc;

namespace SampleAspNetCore.Controllers

{

[Route("api/[controller]")]

[ApiController]

public *class* BooksController : ControllerBase

{

}

}

Create a new Models folder and add a new BookModel class inside this Models folder.

Graphical user interface, text, application

Description automatically generatedBookModel in the Models folder

Add the following properties in the BookModel class.

public *class* BookModel

{

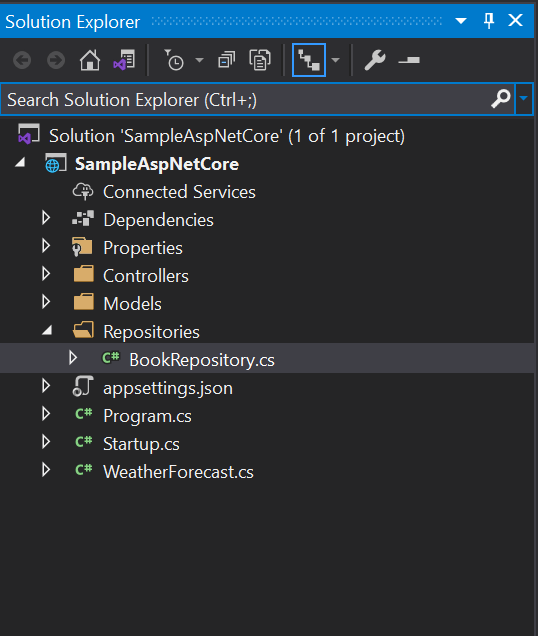
public int Id { get; set; }

public string Name { get; set; }

public int Price { get; set; }

}

Now, Create a new folder with the name Repositories at the root level or a new class library project in the solution (As per your choice or application architecture). And create a new BookRepository class in this folder/ classLib project. I am going to create a new folder at the root level.

Repository layer in asp.net core

We are not using any database in this example. So let’s create some in methods and in-memory data in this BookRepository class.

using SampleAspNetCore.Models;

using System.Collections.Generic;

namespace SampleAspNetCore.Repositories

{

public *class* BookRepository

{

/// <summary>

/// Hold in-memory books data

/// </summary>

private List<BookModel> books = new List<BookModel>();

/// <summary>

/// Add a new book in the books list

/// </summary>

/// <param name="book"></param>

/// <returns>Id of new book</returns>

public int AddBook(BookModel book)

{

book.Id = books.Count + 1; // Create the incremental Id

books.Add(book);

return book.Id;

}

/// <summary>

/// Gets all books

/// </summary>

/// <returns>All books</returns>

public List<BookModel> GetAllBooks()

{

return books;

}

}

}

We need to use this BookRepository class in the BookController to Add and Get the books data.

*If we do not want to follow the concept of Dependency Injection then we can simply create the object of BookRepository using new keyword and use both the methods of this BookRepository class.*

But, If we are following the concept of Dependency Injection then we need to create an interface IBookRepository for BookRepository. We will only expose this IBookRepository to the controller and because of this the controller will never know about the implementation i.e. BookRepository.

using SampleAspNetCore.Models;

using System.Collections.Generic;

namespace SampleAspNetCore.Repositories

{

public interface IBookRepository

{

int AddBook(BookModel book);

List<BookModel> GetAllBooks();

}

}

Make sure to inherit the BookRepository class from IBookRepositoryinterface.

public *class* BookRepository : IBookRepository

Inject the Dependency using Constructor in Asp.Net Core –

The Repository layer is ready. Now we need to make some changes in the controller class and inject the dependency using constructor.

using Microsoft.AspNetCore.Mvc;

using SampleAspNetCore.Models;

using SampleAspNetCore.Repositories;

namespace SampleAspNetCore.Controllers

{

[Route("api/[controller]")]

[ApiController]

public *class* BooksController : ControllerBase

{

private readonly IBookRepository \_bookRepository;

/// <summary>

/// Constructor Injection

/// </summary>

/// <param name="bookRepository"></param>

public BooksController(IBookRepository bookRepository)

{

\_bookRepository = bookRepository;

}

[HttpPost("")]

public IActionResult AddBook([FromBody] BookModel book)

{

int id = \_bookRepository.AddBook(book);

return Ok(id);

}

[HttpGet("")]

public IActionResult GetAllBooks()

{

var books = \_bookRepository.GetAllBooks();

return Ok(books);

}

}

}

*In the above code, you can check that we are not using BookRepository anywhere in the HomeController. We have created the types using IBookRepository interface.*

If you will run this application at this stage then you will get an error. This is because that at this time even the application also does not know about the implementation of the IBookRepository interface. We can tell the implementation to the application by registering our service in the container.

Dependency Injection Lifetime in Asp.Net Core –

Because we are not creating the repository class object directly in the controller, so Asp.Net Core framework provides us few methods to handle the lifetime of the object.

Singleton

Scoped

Transient

Let’s learn about the implementation and difference among Singleton and Scoped and Transient services.

Singleton Service lifetime in Dependency Injection –

Singleton services can be registered using AddSingleton<> method.

There will be only one instance of the Singleton service throughout the application.

Let’s open the Startup class and register the BookRepository using Singleton service.

public void ConfigureServices(IServiceCollection services)

{

services.AddControllers();

services.AddSingleton<IBookRepository, BookRepository>();

// Register other dependencies here

}

Now, we have registered the dependency using the Singleton service and everything is ready to test. Let’s run the application and send a request from any client tool ([postman](https://www.postman.com/downloads/), [fiddler](https://www.telerik.com/fiddler), etc.)

Test the Singleton Service –

Send a POST request at http://localhost:xxxx/api/books with the following JSON in the body.

{

"name": "C#",

"price": 300

}

A screenshot of a computer

Description automatically generated with medium confidenceSingle service request

Just send a few more requests by updating the JSON data in the body. and finally, use the GET call on the same URL.

A screenshot of a computer

Description automatically generated with medium confidenceGet all books

Observations –

The same instance of the service is shared among all the HTTP requests because the entire data is stored in memory and we can access it using a separate HTTP request.

Once you will restart the application then in-memory data will get lost because there will always be a new instance for each run of the application.

Example with multiple instances –

Let’s validate the Singleton service with multiple instances of same service.

Let’s create one more instance of the IBookRepository in the BooksController.

private readonly IBookRepository \_bookRepository;

private readonly IBookRepository \_bookRepository2;

/// <summary>

/// Constructor Injection

/// </summary>

/// <param name="bookRepository"></param>

/// <param name="bookRepository2"></param>

public BooksController(IBookRepository bookRepository, IBookRepository bookRepository2)

{

\_bookRepository = bookRepository;

\_bookRepository2 = bookRepository2;

}

Just for testing let’s use both the instances in the AddBook method of BooksController and this time we will perform the following two operations in this same method.

Add the new book using the first instance i.e. \_bookRepository

Get all books using the second instance i.e. \_bookRepository2

Let’s update the AddBook method of BooksController.

[HttpPost("")]

public IActionResult AddBook([FromBody] BookModel book)

{

\_bookRepository.AddBook(book); // Add book using first instance

var books = \_bookRepository2.GetAllBooks(); // Get all books using secind instance

return Ok(books);

}

Run the application again send the post request with some data. This time once you will send the POST request (to add a new book), You will receive all the books in the output.

Keep sending at least five requests with updated body JSON and every time in the output you will see all previously added books are available.

**Write the output of the HTTP requests with your thoughts in the comment section below this post.**

*In case you are using Asp.Net Core MVC then to test the multiple instances, you can add the books using the first instance and use the second instance to display all the books by injecting them directly in the View.cshtml*

Scoped Service lifetime in Dependency Injection –

Scoped services can be registered using AddScoped<> method.

A new instance of the service will be created for the new HTTP Request.

Example: Let’s say the controller A is using a Scoped service S two times in the same HTTP Request, then there will be only one (1) instance of this S service for this request.

Let’s register the **Scoped Dependency** in the Startup class.

public void ConfigureServices(IServiceCollection services)

{

services.AddControllers();

//services.AddSingleton<IBookRepository, BookRepository>();

services.AddScoped<IBookRepository, BookRepository>();

// Register other dependencies here

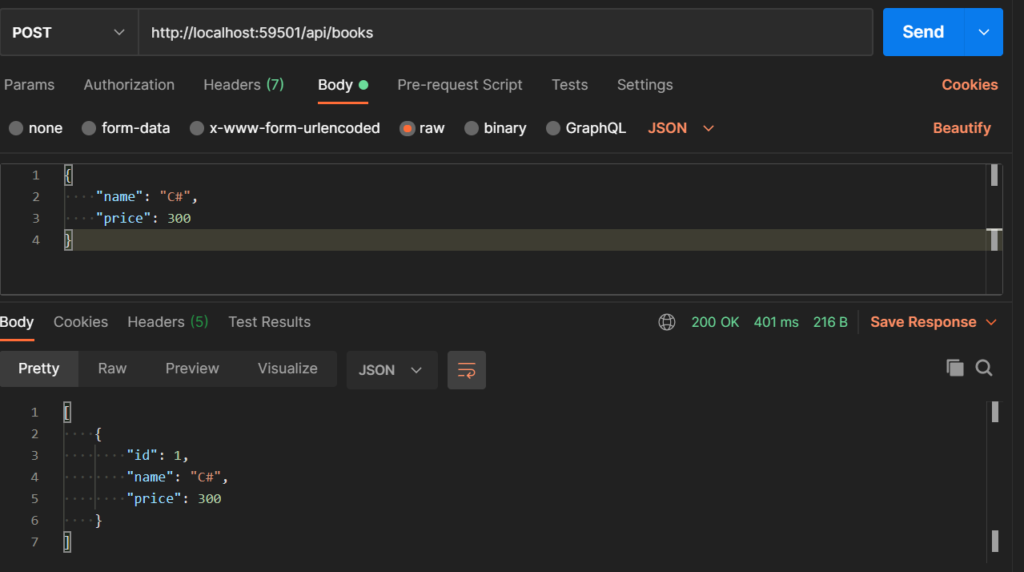
}

*Note: You can register n number of services in the ConfigureServices method.*

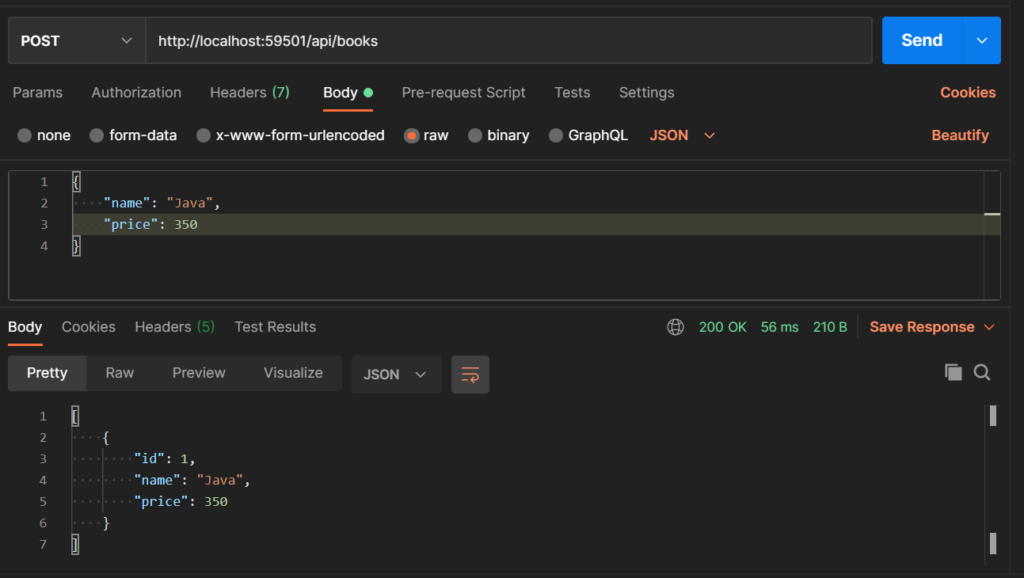
Test the Scoped lifetime in Dependency Injection –

We have already setup everything, Now without making any further changes in the code let’s run the application and test the Scoped lifetime.

There is no change in the request also. You can send the previously (That one we were using with Singleton lifetime) request.

Scoped lifetime in Dependency Injection

Update the JSON body and send another request.

Scoped lifetime in Dependency Injection example -2

Observations:

We are getting only the current request data in the response of the HTTP Request. It means the instances are not shared across HTTP Requests.

The instances are being shared for the same HTTP Request because we are using \_bookRepository to add the new book and \_bookRepository2 to get all the books.

Transient Service lifetime in Dependency Injection –

Transient services can be registered using AddTransient<> method.

A new instance of the service will be created every-time it is requested.

Example: Let’s say the Controller A is using a Transient service S 2 times in the same HTTP Request, then there will be 2 **separate instances** of this S service.

Let’s register the **Transient Dependency** in the Startup class.

public void ConfigureServices(IServiceCollection services)

{

services.AddControllers();

//services.AddSingleton<IBookRepository, BookRepository>();

//services.AddScoped<IBookRepository, BookRepository>();

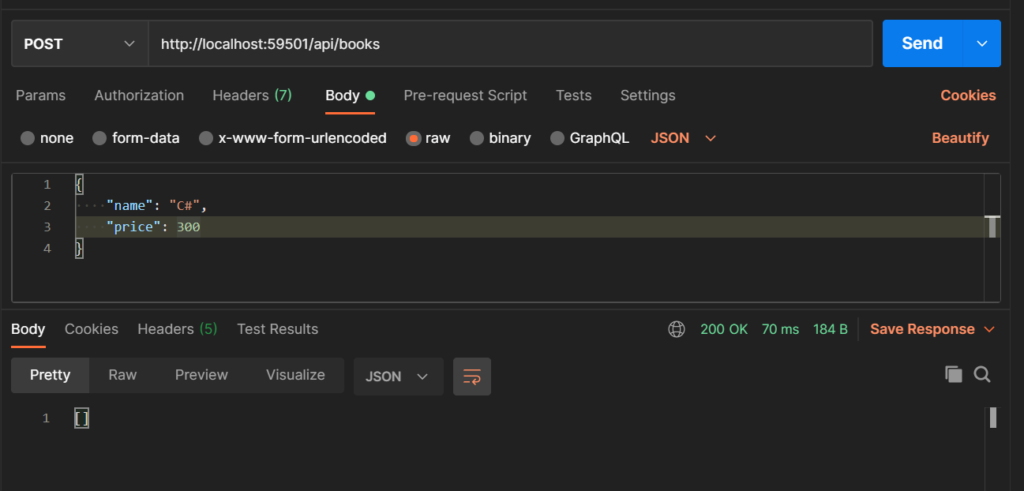
services.AddTransient<IBookRepository, BookRepository>();

// Register other dependencies here

}

Test the Scoped lifetime in Dependency Injection –

Because we have not made any change in the Controller and Service so let’s send the HTTP Request with same data.

Transient lifetime in Dependency Injection

Observations:

As per our code structure, there is no output of this HTTP Request in the Transient service. This is because we are using two separate instances of the BookRepository in the same HTTP Request. And as per the Transient service, a separate instance will be created every time it is requested.

Transient service does not share data between multiple instances hence it is a valid case in most of the scenarios.

Difference between SQL and NoSQL Database

When it comes to comparing among the SQL and NoSQL networks, the SQL is table-based databases, but NoSQL databases are in document forms, key-value pair, or graph databases. According to the performance, SQL requires specialist database hardware, but NoSQL can run on existing hardware.

Although both databases are excellent solutions, there are several major distinctions between them that users should consider before making a choice. The following is a comparison table showing the differences between the SQL and NoSQL databases:

| **Basis Of Comparison** | **SQL Database** | **NoSQL Database** |
| --- | --- | --- |
| **Structure** | A table-based structure is a preferable choice for programs that demand multi-row operations, such as accounting systems, or for legacy applications designed with a relational structure in mind. | Document-based structure with Key-value pairs, graph databases, and wide-column stores. |
| **Query Language** | SQL is a Structured Query Language | There is no declarative query language that NoSQL uses. It differentiates as per the different databases. |
| **Suitable** | The database is best suitable for Complex queries but not good for hierarchical data storage. | The database is not suitable for complex queries. However, it suits well for the hierarchical data storage. |
| **Scalability** | These databases are vertically scalable. So, with boosting CPU, RAM, or SSD, you may increase the demand on a single server. | NoSQL databases are horizontally scalable. This means by sharding or adding multiple servers to this database, you can handle greater traffic. |
| **Design for** | Traditional RDBMS examine and retrieve data for further analysis using SQL syntax and queries. The OLAP systems take advantage of them. | The NoSQL database system is made up of many database technologies. These were created in line with the requirements placed on recent application development. |
| **Schema** | SQL databases consists of a predefined schema | NoSQL databases comprises of the dynamic schema for unstructured data. |
| **Properties Followed** | SQL accompanies ACID properties that stand as Atomicity, Consistency, Isolation and Durability. | NoSQL accompanies Brewers CAP theorem that stands as Consistency, Availability and Partition Tolerance. |
| **Examples** | MySQL, Sqlite, MS-SQL, PostgreSQL, etc. | Big Table, MongoDB, Hbase, Cassandra, Raven DB, etc. |

Lastly, the providers provide excellent support for all SQL databases. There are also many independent consultants present to assist you with SQL databases for exceptionally large-scale deployments. However, one must still rely on support from the community for some NoSQL databases. Just a few outside professionals are accessible to design and execute your high size NoSQL deployments.

When to use SQL Database?

SQL is the finest database to employ for relational data, notably when the link among data sets is easily understandable and accessible. It’s also the greatest way to check for data security and integrity. SQL enables rising ad-hoc queries, and, in most circumstances, SQL databases are vertically expandable if you require flexibility in data access.

Some SQL databases have specific characteristics that support NoSQL workloads (for example, native JavaScript Object Notation (JSON) data types). If you don’t require the horizontal scalability of NoSQL data storage, these databases can handle a variety of non-relational applications.

Use Cases

      SQL is the simplest language used for interacting with a relational database management system.

      Understanding and modifying behavioural-related programs

      Creating unique dashboards

When to Use NoSQL Database?

For massive amounts of data or continuously updating data sets, NoSQL is the best database to employ. If you have variable database schemas or requirements that resist fitting into the relational data model. “Document databases” (e.g., CouchDB, Amazon DocumentDB, MongoDB) are a suitable choice if you’re dealing with immense quantities of unstructured data.

Use Cases

      Data that necessitates a flexible schema

      When ACID assistance isn’t required

      Data logging from a variety of source

**C# | Delegates:**

A delegate is an object which refers to a method or you can say it is a reference type variable that can hold a reference to the methods. Delegates in C# are similar to the function pointer in C/C++. It provides a way which tells which method is to be called when an event is triggered.

For example, if you click on a Button on a form (Windows Form application), the program would call a specific method. In simple words, it is a type that represents references to methods with a particular parameter list and return type and then calls the method in a program for execution when it is needed.

**Important Points About Delegates:**

Provides a good way to encapsulate the methods.

Delegates are the library class in System namespace.

These are the type-safe pointer of any method.

Delegates are mainly used in implementing the call-back methods and events.

Delegates can be chained together as two or more methods can be called on a single event.

It doesn’t care about the class of the object that it references.

Delegates can also be used in “anonymous methods” invocation.

Anonymous Methods(C# 2.0) and Lambda expressions(C# 3.0) are compiled to delegate types in certain contexts. Sometimes, these features together are known as anonymous functions.

**Declaration of Delegates:**

Delegate type can be declared using the delegate keyword. Once a delegate is declared, delegate instance will refer and call those methods whose return type and parameter-list matches with the delegate declaration.

Syntax:

[modifier] delegate [return\_type] [delegate\_name] ([parameter\_list]);

modifier: It is the required modifier which defines the access of delegate and it is optional to use.

delegate: It is the keyword which is used to define the delegate.

return\_type: It is the type of value returned by the methods which the delegate will be going to call. It can be void. A method must have the same return type as the delegate.

delegate\_name: It is the user-defined name or identifier for the delegate.

parameter\_list: This contains the parameters which are required by the method when called through the delegate.

Example:

// "public" is the modifier

// "int" is return type

// "GeeksForGeeks" is delegate name

// "(int G, int F, int G)" are the parameters

public delegate int GeeksForGeeks(int G, int F, int G);

Note: A delegate will call only a method which agrees with its signature and return type. A method can be a static method associated with a class or can be an instance method associated with an object, it doesn’t matter.

**Instantiation & Invocation of Delegates:**

After declaring a delegate, a delegate object is created with the help of new keyword. Once a delegate is instantiated, a method call made to the delegate is pass by the delegate to that method. The parameters passed to the delegate by the caller are passed to the method, and the return value, if any, from the method, is returned to the caller by the delegate. This is known as invoking the delegate.

Syntax:

[delegate\_name] [instance\_name] = new [delegate\_name](calling\_method\_name);

Example:

GeeksForGeeks GFG = new GeeksForGeeks (Geeks);

// here,

// "GeeksForGeeks" is delegate name.

// "GFG" is instance\_name

// "Geeks" is the calling method.

Below program illustrate the use of Delegate:

CSharp

// C# program to illustrate the use of Delegates

using System;

namespace GeeksForGeeks {

// declare class "Geeks"

class Geeks {

// Declaring the delegates

// Here return type and parameter type should

// be same as the return type and parameter type

// of the two methods

// "addnum" and "subnum" are two delegate names

public delegate void addnum(int a, int b);

public delegate void subnum(int a, int b);

// method "sum"

public void sum(int a, int b)

{ Console.WriteLine("(100 + 40) = {0}", a + b); }

// method "subtract"

public void subtract(int a, int b)

{

Console.WriteLine("(100 - 60) = {0}", a - b);

}

// Main Method

public static void Main(String []args)

{

// creating object "obj" of class "Geeks"

Geeks obj = new Geeks();

// creating object of delegate, name as "del\_obj1"

// for method "sum" and "del\_obj2" for method "subtract" &

// pass the parameter as the two methods by class object "obj"

// instantiating the delegates

addnum del\_obj1 = new addnum(obj.sum);

subnum del\_obj2 = new subnum(obj.subtract);

// pass the values to the methods by delegate object

del\_obj1(100, 40);

del\_obj2(100, 60);

// These can be written as using

// "Invoke" method

// del\_obj1.Invoke(100, 40);

// del\_obj2.Invoke(100, 60);

}

}

}

Output:

(100 + 40) = 140

(100 - 60) = 40

Explanation: In the above program, there are two delegates addnum and subnum. We are creating the object obj of the class Geeks because both the methods(addnum and subnum) are instance methods. So they need an object to call. If methods are static then there is no need to create the object of the class.

Multicasting of a Delegate

Multicasting of delegate is an extension of the normal delegate(sometimes termed as Single Cast Delegate). It helps the user to point more than one method in a single call.

Properties:

Delegates are combined and when you call a delegate then a complete list of methods is called.

All methods are called in First in First Out(FIFO) order.

‘+’ or ‘+=’ Operator is used to add the methods to delegates.

‘–’ or ‘-=’ Operator is used to remove the methods from the delegates list.

Note: Remember, multicasting of delegate should have a return type of Void otherwise it will throw a runtime exception. Also, the multicasting of delegate will return the value only from the last method added in the multicast. Although, the other methods will be executed successfully.

Below program demonstrates the use of Multicasting of a delegate:

CSharp

// C# program to illustrate the

// Multicasting of Delegates

using System;

class rectangle {

// declaring delegate

public delegate void rectDelegate(double height, double width);

// "area" method

public void area(double height, double width)

{

Console.WriteLine("Area is: {0}", (width \* height));

}

// "perimeter" method

public void perimeter(double height, double width)

{

Console.WriteLine("Perimeter is: {0} ", 2 \* (width + height));

}

// Main Method

public static void Main(String []args)

{

// creating object of class

// "rectangle", named as "rect"

rectangle rect = new rectangle();

// these two lines are normal calling

// of that two methods

// rect.area(6.3, 4.2);

// rect.perimeter(6.3, 4.2);

// creating delegate object, name as "rectdele"

// and pass the method as parameter by

// class object "rect"

rectDelegate rectdele = new rectDelegate(rect.area);

// also can be written as

// rectDelegate rectdele = rect.area;

// call 2nd method "perimeter"

// Multicasting

rectdele += rect.perimeter;

// pass the values in two method

// by using "Invoke" method

rectdele.Invoke(6.3, 4.2);

Console.WriteLine();

// call the methods with

// different values

rectdele.Invoke(16.3, 10.3);

}

}

Output:

Area is: 26.46

Perimeter is: 21

Area is: 167.89

Perimeter is: 53.2

C# | Predicate Delegate

A Predicate delegate is an in-built generic type delegate. This delegate is defined under System namespace. It works with those methods which contain some set of criteria and determine whether the passed parameter fulfill the given criteria or not. This delegate takes only one input and returns the value in the form of true or false. Now, first of all, we see how custom delegates work in this situation. As shown in the below example.

Syntax:

public delegate bool Predicate <in P>(P obj);

Here, P is the type of the object and obj is the object which is going to compare against the criteria defined within the method represented by Predicate delegate.

Example:

// C# program to illustrate delegates

using System;

class GFG {

// Declaring the delegate

public delegate bool my\_delegate(string mystring);

// Method

public static bool myfun(string mystring)

{

if (mystring.Length < 7)

{

return true;

}

else

{

return false;

}

}

// Main method

static public void Main()

{

// Creating object of my\_delegate

my\_delegate obj = myfun;

Console.WriteLine(obj("Hello"));

}

}

Output:

True

Now, we use the same above program with Predicate delegate as shown below.

Example: In the below example, we use a predicate delegate instead of a custom delegate. It reduces the size of the code and makes the program more readable. Here, the Predicate delegate contains a single input parameter and return output in boolean type. And here, we directly assign a myfun method to the Predicate delegate.

// C# program to illustrate Predicate delegates

using System;

class GFG {

// Method

public static bool myfun(string mystring)

{

if (mystring.Length < 7)

{

return true;

}

else {

return false;

}

}

// Main method

static public void Main()

{

// Using predicate delegate

// here, this delegate takes

// only one parameter

Predicate<string> val = myfun;

Console.WriteLine(val("GeeksforGeeks"));

}

}

Output:

False

Important Points:

You can also use a Predicate delegate with an anonymous method as shown in the below example:

Example:

Predicate<string> val = delegate(string str)

{

if (mystring.Length < 7)

{

return true;

}

else

{

return false;

};

val("Geeks");

You can also use a Predicate delegate with the lambda expressions as shown in the below example:

Example:

Predicate<string> val = str = > str.Equals(str.ToLower());

val("Geeks");

**C# | Action Delegate**

Action delegate is an in-built generic type delegate. This delegate saves you from defining a custom delegate as shown in the below examples and make your program more readable and optimized. It is defined under System namespace. It can contain minimum 1 and maximum of 16 input parameters and does not contain any output parameter. The Action delegate is generally used for those methods which do not contain any return value, or in other words, Action delegate is used with those methods whose return type is void. It can also contain parameters of the same type or of different types.

Syntax:

// One input parameter

public delegate void Action < in P > (P obj);

// Two input parameters

public delegate void Action < in P1, in P2 >(P1 arg1, P2 arg2);

Here, P, P1, and P2 are the type of the input parameters & arg1 and agr2 are the parameters of the method that Action delegate encapsulates.

Example: Below program illustrate how we create a custom delegate.

// C# program to illustrate delegates

using System;

class GFG {

// Declaring the delegate

public delegate void my\_delegate(int p, int q);

// Method

public static void myfun(int p, int q)

{

Console.WriteLine(p - q);

}

// Main method

static public void Main()

{

// Creating object of my\_delegate

my\_delegate obj = myfun;

obj(10, 5);

}

}

Output:

5

Example: It demonstrates the use of Action delegate.

// C# program to illustrate Action delegates

using System;

class GFG {

// Method

public static void myfun(int p, int q)

{

Console.WriteLine(p - q);

}

// Main method

static public void Main()

{

// Using Action delegate

// Here, Action delegate

// contains two input parameters

Action<int, int> val = myfun;

val(20, 5);

}

}

Example:

Action<string> val = delegate(string str)

{

Console.WriteLine(str);

};

val("GeeksforGeeks");

You can also use a Action delegate with the lambda expressions as shown in the below example:

Example:

Action<string> val = str = > Console.WriteLine(str);

val("GeeksforGeeks");

C# | Func delegatee we have to follow the following steps:

Step 1: Declare a custom delegate with the format which is exactly equal to the method.

Step 2: Create the object of custom delegate.

Step 3: Invoke the method.

By using these steps, we create a custom delegate as shown in the below program. But the problem is that for creating a delegate we need to follow the above procedure. To overcome this situation C# provides a built-in delegate that is Func. Using Func delegate you need not follow the following procedure to create a delegate.

Example:

// C# program to illustrate how to

// create custom delegates

using System;

class GFG {

// Declaring the delegate

public delegate int my\_delegate(int s, int d,

int f, int g);

// Method

public static int mymethod(int s, int d,

int f, int g)

{

return s \* d \* f \* g;

}

// Main method

static public void Main()

{

// Creating object of my\_delegate

my\_delegate obj = mymethod;

Console.WriteLine(obj(12, 34, 35, 34));

}

}

Output:

485520

A Func is a built-in generic type delegate. This delegate saves you from defining a custom delegate like as shown in the above example and make your program more readable and optimized. As we know that, Func is a generic delegate so it is defined under System namespace. It can contain minimum 0 and maximum of 16 input parameters in it and contain only one out parameter. The last parameter of the Func delegate is the out parameter which is considered as return type and used for the result. Func is generally used for those methods which are going to return a value, or in other words, Func delegate is used for value returning methods. It can also contain parameters of the same type or of different types.

Syntax:

// Zero normal parameters and one result parameter

public delegate TResult Func<out PResult>();

// One normal parameter and one result parameter

public delegate TResult Func<in P, out PResult>(P arg);

// Two normal parameters and one result parameter

public delegate TResult Func<in P1, in P2, out PResult>(P1 arg1, P2 arg2);

// Sixteen normal parameters and one result parameter

public delegate TResult Func<in P1, in P2, in P3, in P4, in P05, in P6, in P7, in P8, in P9, in P10, in P11, in P12, in P13, in P14, in P15, in P16, out PResult>(P1 arg1, P2 arg2, P3 arg3, P4 arg4, P5 arg5, P6 arg6, P7 arg7, P8 arg8, P9 arg9, P10 arg10, P11 arg11, P12 arg12, P13 arg13, P14 arg14, P15 arg15, P16 arg16);

Here, P1, P2….P16 are the type of input parameters, PResult is the type of output parameter, and arg1….arg16 are the parameter of the method that the Func delegate encapsulates.

Example 1: Here, we use a Func delegate to create a delegate only in a single line without using the above procedure. This Func delegate contains four input parameters and one output parameter.

// C# program to illustrate Func delegate

using System;

class GFG {

// Method

public static int mymethod(int s, int d, int f, int g)

{

return s \* d \* f \* g;

}

// Main method

static public void Main()

{

// Using Func delegate

// Here, Func delegate contains

// the four parameters of int type

// one result parameter of int type

Func<int, int, int, int, int> val = mymethod;

Console.WriteLine(val(10, 100, 1000, 1));

}

}

Output:

1000000

Example 2:

// C# program to illustrate Func delegate

using System;

class GFG {

// Method

public static int method(int num)

{

return num + num;

}

// Main method

static public void Main()

{

// Using Func delegate

// Here, Func delegate contains

// the one parameters of int type

// one result parameter of int type

Func<int, int> myfun = method;

Console.WriteLine(myfun(10));

}

}

Output: 20

Important Points:

The last parameter in Func Delegate is always an out parameter which is considered as a return type. It is generally used for the result.

You can also use a Func delegate with an anonymous method. As shown in the below example:

Example:

Func<int, int, int> val = delegate(int x, int y, int z)

{

return x + y + z;

};

Func<int, int, int, int> val = (int x, int y, int z) = > x + y + z;

**Below are some differences between the Delegates and Interfaces in C#:**

| **Delegate** | **Interface** |
| --- | --- |
| It could be a method only. | It contains both methods and properties. |
| It can be applied to one method at a time. | If a class implements an interface, then it will implement all the methods related to that interface. |
| If a delegate available in your scope you can use it. | Interface is used when your class implements that interface, otherwise not. |
| Delegates can me implemented any number of times. | Interface can be implemented only one time. |
| It is used to handling events. | It is not used for handling events. |
| It can access anonymous methods. | It can not access anonymous methods. |
| When you access the method using delegates you do not require any access to the object of the class where the method is defined. | When you access the method you need the object of the class which implemented an interface. |
| It does not support inheritance. | It supports inheritance. |
| It can wrap static methods and sealed class methods | It does not wrap static methods and sealed class methods.. |
| It created at run time. | It created at compile time. |
| It can implement any method that provides the same signature with the given delegate. | If the method of interface implemented, then the same name and signature method override. |
| It can wrap any method whose signature is similar to the delegate and does not consider which from class it belongs. | A class can implement any number of interfaces, but can only override those methods which belongs to the interfaces. |