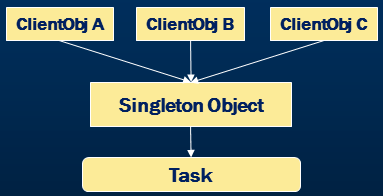
# Introduction to Design Patterns

**In this tutorial we will discuss**  
1. What are Design Patterns  
2. Evolution of Design Patterns   
3. The three categories of design patterns    
  
In many interviews you might have encountered lot of questions related to interfaces, abstract classes, delegates and other features related to object oriented programming along with the design pattern related questions.    
  
  
  
Design patterns solutions are evolved from the  features of object oriented programming.    
  
Once you understand the design patterns, it makes you very comfortable in attending interviews as well as applying these features with confidence in your projects or applications.  
    
Implementing design patterns in the applications are proven and tested. Writing the code aligning with design patterns will make your applications Reliable,Scalable and Maintainable.    
  
  
  
**What are Design Patterns**   
Design patterns are reusable solutions to the problems that we encounter in the day to day programming. They are generally targeted at solving the problems of object generation and integration. In other words, Design patterns acts as templates which can be applied to the real-world programming problems.  
  
**History and evolution of design Patterns**  
The four authors of the book famously know as Gang of four are the ones who brought the concepts of design patterns in their book “Elements of reusable Object-Oriented software” .   
   
Gang of Four has divided the book into two parts with first part explaining about the pros and cons of object oriented programming and the second part describes the evolution of 23 classic software design patterns.  
   
For more details, please refer to the following wikipedia article  
<https://en.wikipedia.org/wiki/Design_Patterns>  
  
**Types of Design Patterns**  
Gang of Four have categorised the design patterns in to 3 types based on different problems encountered in the real world applications. They are Creational, Structural and Behavioural.  
   
**Creational design patterns :**These patterns deal with object creation and initialization. Creational pattern gives the program more flexibility in deciding which objects need to be created for a given case.  
   
**Examples of Creational design patterns category :**Singleton , Factory and Abstract Factory etc.  
  
**Structural design patterns :**This pattern deals with class and object composition. In simple words, This pattern focuses on decoupling interface, implementation of classes and its objects.   
  
**Examples of Structural design patterns category :**Adapter,  Facade and Bridge etc.  
  
**Behavioural design patterns :**These patterns deal with communication between Classes and objects.   
  
**Examples of Behavioural design patterns :** Chain of Responsibility, Command and Interpreter etc.  
   
For understanding design patterns better it is very important to have basic knowledge about the following object oriented concepts

* Abstraction
* Inheritance
* Polymorphism
* Encapsulation
* Interfaces
* Classes
* Abstract classes

In Part 2 we will focus on the Singleton design pattern in detail with a simple example.

### Singleton Design Pattern

**Suggested Videos**  
[Part 1 - Introduction to Design Patters](https://www.youtube.com/watch?v=rI4kdGLaUiQ) - [Text](http://csharp-video-tutorials.blogspot.com/2017/05/introduction-to-design-patterns.html) - [Slides](http://csharp-video-tutorials.blogspot.com/2017/05/introduction-to-design-patterns_4.html)   
  
**In this tutorial we will discuss**   
1. What is Singleton Design Pattern  
2. Singleton as Creational Pattern  
3. Implementation Guidelines  
4. How do we implement a Singleton class   
  
  
This is continuation to [Part 1](https://www.youtube.com/watch?v=rI4kdGLaUiQ) of [Design Patterns Tutorial](https://www.youtube.com/playlist?list=PL6n9fhu94yhUbctIoxoVTrklN3LMwTCmd). So please watch [Part 1](https://www.youtube.com/watch?v=rI4kdGLaUiQ) before proceeding.   
  
**Singleton Pattern** belongs to **Creational type pattern**. As discussed in our previous video, Gang of four have defined five design patterns that belongs to creational design type category. Singleton is one among them and the rest are Factory, Abstract Factory, Builder and Prototype patterns. As the name implies, creational design type deals with object creation mechanisms. Basically, to simplify this, creational pattern explain us the creation of objects in a manner suitable to a given situation.   
  
  
Singleton design pattern is used when we need to ensure that only one object of a particular class is Instantiated. That single instance created is responsible to coordinate actions across the application.   
  
   
  
If you look at the illustrated diagram above you will see different objects trying to invoke an object instantiated as singleton. This single instance of the object is responsible to invoke underneath methods or events.  
  
**Advantages and Guidelines for Singleton implementation.**  
  
Concurrent access to the resource is well managed by singleton design pattern.  
  
As part of the Implementation guidelines we need to ensure that only one instance of the class exists by declaring all constructors of the class to be private.  Also, to control the singleton access we need to provide a static property that returns a single instance of the object.  
  
**Singleton Class Implementation Example**

**Program.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

/// <summary>

/// First version of Singleton demo

/// </summary>

namespace SingletonDemo

{

    class Program

    {

        static void Main(string[] args)

        {

            /\*

             \* Assuming Singleton is created from employee class

             \* we refer to the GetInstance property from the Singleton class

             \*/

            Singleton fromEmployee = Singleton.GetInstance;

            fromEmployee.PrintDetails("From Employee");

            /\*

             \* Assuming Singleton is created from student class

             \* we refer to the GetInstance property from the Singleton class

             \*/

            Singleton fromStudent = Singleton.GetInstance;

            fromStudent.PrintDetails("From Student");

            Console.ReadLine();

        }

    }

}

**Singleton.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

/// <summary>

/// First Singleton version

/// </summary>

namespace SingletonDemo

{

    /\*

     \*  Sealed ensures the class being inherited and

     \*  object instantiation is restricted in the derived class

     \*/

    public sealed class Singleton

    {

        private static int counter = 0;

        /\*

         \* Private property initilized with null

         \* ensures that only one instance of the object is created

         \* based on the null condition

         \*/

        private static Singleton instance = null;

        /\*

         \* public property is used to return only one instance of the class

         \* leveraging on the private property

         \*/

        public static Singleton GetInstance

        {

            get

            {

                if (instance == null)

                    instance = new Singleton();

                return instance;

            }

        }

        /\*

         \* Private constructor ensures that object is not

         \* instantiated other than with in the class itself

         \*/

        private Singleton()

        {

            counter++;

            Console.WriteLine("Counter Value " + counter.ToString());

        }

        /\*

         \* Public method which can be invoked through the singleton instance

         \*/

        public void PrintDetails(string message)

        {

            Console.WriteLine(message);

        }

    }

}

Why is singleton class sealed

**Suggested Videos**  
[Part 1 - Introduction to Design Patters](https://www.youtube.com/watch?v=rI4kdGLaUiQ) - [Text](http://csharp-video-tutorials.blogspot.com/2017/05/introduction-to-design-patterns.html) - [Slides](http://csharp-video-tutorials.blogspot.com/2017/05/introduction-to-design-patterns_4.html)  
[Part 2 - Singleton Design Pattern](https://www.youtube.com/watch?v=YGGg9ecy0K4) - [Text](http://csharp-video-tutorials.blogspot.com/2017/05/singleton-design-pattern.html) - [Slides](http://csharp-video-tutorials.blogspot.com/2017/05/singleton-design-pattern_13.html)   
  
**In this tutorial we will discuss**

* A quick Recap of Singleton version which we have discussed in the previous session
* We will focus on why we used Sealed keyword
* Why we need to seal the singleton class

**In our previous video we discussed**

* Why we need to create a singleton class and how we can apply singleton design pattern to a class
* We have changed the class to restrict external instantiation by changing the public constructor to private and then we provided a public property to access this class
* We have proved that adding Private constructor will prevent us instantiating a new class
* We have further sealed down this class to avoid any inheritance

You might be wondering why we need to seal the class when a private constructor is present.    
  
Let’s first remove the sealed keyword and check that. Let’s create another class called DerivedSingleton and Inherit the singleton class. Let's compile the code and look at that it has thrown an error saying Singleton is inaccessible due to its protection level. This error is because of private constructor.  
  
Now you might be thinking that when a private constructor is restricting the inheritance then why we need to apply sealed keyword to the class.   
  
Let’s just move this new class inside the Singleton class. By moving this class inside the Singleton class it has now become nested or child class of the main singleton class.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

/// <summary>

/// First Singleton version

/// </summary>

namespace SingletonDemo

{

    /\*

     \*  Sealed restricts the inheritance

     \*/

    public class Singleton

    {

        private static int counter = 0;

        private static object obj = new object();

        /\*

        \* Private constructor ensures that object is not

        \* instantiated other than with in the class itself

        \*/

        private Singleton()

        {

            counter++;

            Console.WriteLine("Counter Value " + counter.ToString());

        }

        private static Singleton instance = null;

        /\*

         \* public property is used to return only one instance of the class

         \* leveraging on the private property

         \*/

        public static Singleton GetInstance

        {

            get

            {

                if (instance == null)

                    instance = new Singleton();

                return instance;

            }

        }

        /\*

         \* Public method which can be invoked through the singleton instance

         \*/

        public void PrintDetails(string message)

        {

            Console.WriteLine(message);

        }

        /\*

         \* By removing sealed keyword we can inherit the singleton and instantiate multiple objects

         \* This violates singleton design principles.

         \*/

        public class DerivedSingleton : Singleton

        {

        }

    }

}   
  
**What is a nested class?**   
A class with in another class is called a nested class.   
  
Now that we have moved the derived class to nested class lets compile the program and check. Look at that we are able to compile this successfully.   
  
Now, let’s switch to main program and access the nested class.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

/// <summary>

/// First version of Singleton demo

/// </summary>

namespace SingletonDemo

{

    class Program

    {

        static void Main(string[] args)

        {

            /\*

            \* Assuming Singleton is created from student class

            \* we refer to the GetInstance property from the Singleton class

            \*/

            Singleton fromStudent = Singleton.GetInstance;

            fromStudent.PrintDetails("From Student");

            /\*

            \* Assuming Singleton is created from employee class

            \* we refer to the GetInstance property from the Singleton class

            \*/

            Singleton fromEmployee = Singleton.GetInstance;

            fromEmployee.PrintDetails("From Employee");

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

            /\*

             \* Instantiating singleton from a derived class. This violates singleton pattern principles.

             \*/

            Singleton.DerivedSingleton derivedObj = new Singleton.DerivedSingleton();

            derivedObj.PrintDetails("From Derived");

            Console.ReadLine();

        }

    }

}  
  
Lets run the program. Look at that the counter value has incremented to 2 proving that we are able to create multiple instances of the singleton using the nested derived class.  
  
This violates the principle of singleton.  Let’s go back to the Singleton and make the class as sealed. Let’s compile the program  
  
Look at that we have got an error when we compile the program saying we cannot derive a sealed class. With this we have proved that private constructor helps in preventing any external instantiations of objects and sealed will prevent the class inheritances.  
  
In the next session we will discuss how to handle thread safety in singleton as the current version can create multiple instances in multi-threaded environments.

Thread Safety in Singleton

**Suggested Videos**  
[Part 1 - Introduction to Design Patters](https://www.youtube.com/watch?v=rI4kdGLaUiQ) - [Text](http://csharp-video-tutorials.blogspot.com/2017/05/introduction-to-design-patterns.html) - [Slides](http://csharp-video-tutorials.blogspot.com/2017/05/introduction-to-design-patterns_4.html)  
[Part 2 - Singleton Design Pattern](https://www.youtube.com/watch?v=YGGg9ecy0K4) - [Text](http://csharp-video-tutorials.blogspot.com/2017/05/singleton-design-pattern.html) - [Slides](http://csharp-video-tutorials.blogspot.com/2017/05/singleton-design-pattern_13.html)   
[Part 3 - Why is singleton class sealed](https://www.youtube.com/watch?v=LypTOnfkfvA) - [Text](http://csharp-video-tutorials.blogspot.com/2017/05/why-is-singleton-class-sealed.html) - [Slides](http://csharp-video-tutorials.blogspot.com/2017/05/why-is-singleton-class-sealed_19.html)  
  
**In this tutorial we will discuss**

* Lazy Initialization in Singleton
* How to use Multithreads in Singleton
* How to implement a Thread Safe singleton class

**Lazy Initialization in Singleton :** GetInstance Property is responsible for the Singleton Instance creation. Singleton object is not instantiated until and unless **GetInstance** is invoked. Hence, there is a delay in instance creation till the GetInstance is accessed. This Delay in Instance creation is called Lazy Initialization.   
  
  
  
  
**How to use Multithreads in Singleton :**The lazy initialization works perfectly well when we invoke the GetInstance in a Single threaded approach. However, there is a chance that we may end up creating multiple instances when multiple threads invoke the GetInstance at the same time.  
  
This Thread racing situation causes thread safety issues in Singleton Initialization and further the current code ends up in creating multiple instances of Singleton objects in memory.  
  
To achieve and replicate multiple threads accessing GetInstance, We have modified the main program by using Parallel.Invoke method of .NET Framework 4.0.  Please refer to Main program code below for more details.  
  
**How to implement a Thread Safe singleton class :** Locks are the best way to control thread race condition and they help us to overcome the present situation. Please refer to the Singleton.cs code for lock checks and double check locking.  
  
For more details on double check locking please refer to the below article  
<https://en.wikipedia.org/wiki/Double-checked_locking>

**Program.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

/// <summary>

/// First version of Singleton demo

/// </summary>

namespace SingletonDemo

{

    class Program

    {

        static void Main(string[] args)

        {

            Parallel.Invoke(

                () => PrintStudentdetails(),

                () => PrintEmployeeDetails()

                );

            Console.ReadLine();

        }

        private static void PrintEmployeeDetails()

        {

            /\*

             \* Assuming Singleton is created from employee class

             \* we refer to the GetInstance property from the Singleton class

             \*/

            Singleton fromEmployee = Singleton.GetInstance;

            fromEmployee.PrintDetails("From Employee");

        }

        private static void PrintStudentdetails()

        {

            /\*

                         \* Assuming Singleton is created from student class

                         \* we refer to the GetInstance property from the Singleton class

                         \*/

            Singleton fromStudent = Singleton.GetInstance;

            fromStudent.PrintDetails("From Student");

        }

    }

}

**Singleton.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

/// <summary>

/// First Singleton version

/// </summary>

namespace SingletonDemo

{

    /\*

     \*  Sealed restricts the inheritance

     \*/

    public sealed class Singleton

    {

        private static int counter = 0;

        private static readonly object obj = new object();

        /\*

        \* Private constructor ensures that object is not

        \* instantiated other than with in the class itself

        \*/

        private Singleton()

        {

            counter++;

            Console.WriteLine("Counter Value " + counter.ToString());

        }

        private static Singleton instance = null;

        /\*

         \* public property is used to return only one instance of the class

         \* leveraging on the private property

         \*/

        public static Singleton GetInstance

        {

            get

            {

                if (instance == null)

                {

                    lock (obj)

                    {

                        if (instance == null)

                            instance = new Singleton();

                    }

                }

                return instance;

            }

        }

        /\*

         \* Public method which can be invoked through the singleton instance

         \*/

        public void PrintDetails(string message)

        {

            Console.WriteLine(message);

        }

    }

}

### Lazy vs Eager loading in Singleton

**Suggested Videos**  
[Part 2 - Singleton Design Pattern](https://www.youtube.com/watch?v=YGGg9ecy0K4) - [Text](http://csharp-video-tutorials.blogspot.com/2017/05/singleton-design-pattern.html) - [Slides](http://csharp-video-tutorials.blogspot.com/2017/05/singleton-design-pattern_13.html)  
[Part 3 - Why is singleton class sealed](https://www.youtube.com/watch?v=LypTOnfkfvA) - [Text](http://csharp-video-tutorials.blogspot.com/2017/05/why-is-singleton-class-sealed.html) - [Slides](http://csharp-video-tutorials.blogspot.com/2017/05/why-is-singleton-class-sealed_19.html)   
[Part 4 - Thread Safety in Singleton](https://www.youtube.com/watch?v=QWrcOmLWi_Q) - [Text](http://csharp-video-tutorials.blogspot.com/2017/06/thread-safety-in-singleton.html) - [Slides](http://csharp-video-tutorials.blogspot.com/2017/06/thread-safety-in-singleton_5.html)  
  
In this tutorial we will discuss the **difference between Lazy Initialization and Eager Initialization**  
  
**Lazy Initialization :**The lazy initialization of an object improves the performance and avoids unnecessary computation till the point the object is accessed. Further, it reduces the memory footprint during the startup of the program. Reducing the memory print will help faster loading of the application.   
  
  
  
**Non-Lazy or Eager Loading :** Eager loading is nothing but to initialize the required object before it’s being accessed.  Which means, we instantiate the object and keep it ready and use it when we need it. This type of initialization is used in lower memory footprints. Also, in eager loading, the common language runtime takes care of the variable initialization and its thread safety. Hence, we don’t need to write any explicit coding for thread safety.   
  
  
  
**Singleton with Lazy keyword (.NET 4.0) :** Lazy keyword provides support for lazy initialization. In order to make a property as lazy, we need to pass the type of object to the lazy keyword which is being lazily initialized.   
  
By default, Lazy<T> objects are thread-safe.  In multi-threaded scenarios, the first thread which tries to access the Value property of the lazy object will take care of thread safety when multiple threads are trying to access the Get Instance at the same time.   
  
Therefore, it does not matter which thread initializes the object or if there are any thread race conditions that are trying to access this property.

**Program.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace SingletonDemo

{

    class Program

    {

        static void Main(string[] args)

        {

            Parallel.Invoke(

                () => PrintStudentDetails(),

                () => PrintEmployeeDetails()

            );

            Console.ReadLine();

        }

        private static void PrintEmployeeDetails()

        {

            Singleton fromEmployee = Singleton.GetInstance;

            fromEmployee.PrintDetails("From Employee");

        }

        private static void PrintStudentDetails()

        {

            Singleton fromStudent = Singleton.GetInstance;

            fromStudent.PrintDetails("From Student");

        }

    }

}

**Singleton.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace SingletonDemo

{

    public sealed class Singleton

    {

        private static int counter = 0;

        private Singleton()

        {

            counter++;

            Console.WriteLine("Counter Value " + counter.ToString());

        }

        private static readonly Lazy<Singleton> instance =

new Lazy<Singleton>(()=>new Singleton());

        public static Singleton GetInstance

        {

            get

            {

                return instance.Value;

            }

        }

        public void PrintDetails(string message)

        {

            Console.WriteLine(message);

        }

    }

}

Static Class vs Singleton

**Suggested Videos**  
[Part 3 - Why is singleton class sealed](https://www.youtube.com/watch?v=LypTOnfkfvA) - [Text](http://csharp-video-tutorials.blogspot.com/2017/05/why-is-singleton-class-sealed.html) - [Slides](http://csharp-video-tutorials.blogspot.com/2017/05/why-is-singleton-class-sealed_19.html)  
[Part 4 - Thread Safety in Singleton](https://www.youtube.com/watch?v=QWrcOmLWi_Q) - [Text](http://csharp-video-tutorials.blogspot.com/2017/06/thread-safety-in-singleton.html) - [Slides](http://csharp-video-tutorials.blogspot.com/2017/06/thread-safety-in-singleton_5.html)   
[Part 5 - Lazy vs Eager loading in Singleton](https://www.youtube.com/watch?v=xk-AKHUCdGc) - [Text](http://csharp-video-tutorials.blogspot.com/2017/06/lazy-vs-eager-loading-in-singleton.html) - [Slides](http://csharp-video-tutorials.blogspot.com/2017/06/lazy-vs-eager-loading-in-singleton_12.html)  
  
**Differences between Singleton and static classes**

1. Static is a keyword and Singleton is a design pattern
2. Static classes can contain only static members
3. Singleton is an object creational pattern with one instance of the class
4. Singleton can implement interfaces, inherit from other classes and it aligns with the OOPS concepts
5. Singleton object can be passed as a reference
6. Singleton supports object disposal
7. Singleton object is stored on heap
8. Singleton objects can be cloned

**Static class example - Temperature Converter**  
We are pretty sure that the formulas for foreign heat to Celsius conversion and vice versa will not change at all and hence we can use static classes with static methods that does the conversion for us. Please refer to the below code for more details.  
  
  
  
**Real world usage of Singleton :** Listed are few real world scenarios for singleton usage

1. Exception/Information logging
2. Connection pool management
3. File management
4. Device management such as printer spooling
5. Application Configuration management
6. Cache management
7. And Session based shopping cart are some of the real world usage of singleton design pattern

**Program.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace StaticDemo

{

    class Program

    {

        static void Main(string[] args)

        {

            double celcius = 37; double fahrenheit = 98.6;

            Console.WriteLine("Value of {0} celcius to fahrenheit is {1}",

                celcius, Converter.ToFahrenheit(celcius));

            Console.WriteLine("Value of {0} fahrenheit to celcius is {1}",

                fahrenheit, Converter.ToCelcius(fahrenheit));

            Console.ReadLine();

        }

    }

}

**Converter.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace StaticDemo

{

    public static class Converter

    {

        public static double ToFahrenheit(double celcius)

        {

            return (celcius \* 9 / 5) + 32;

        }

        public static double ToCelcius(double fahrenheit)

        {

            return (fahrenheit - 32) \* 5 / 9;

        }

    }

}

Exception Logging using Singleton Design Pattern

**Suggested Videos**  
[Part 4 - Thread Safety in Singleton](https://www.youtube.com/watch?v=QWrcOmLWi_Q) - [Text](http://csharp-video-tutorials.blogspot.com/2017/06/thread-safety-in-singleton.html) - [Slides](http://csharp-video-tutorials.blogspot.com/2017/06/thread-safety-in-singleton_5.html)  
[Part 5 - Lazy vs Eager loading in Singleton](https://www.youtube.com/watch?v=xk-AKHUCdGc) - [Text](http://csharp-video-tutorials.blogspot.com/2017/06/lazy-vs-eager-loading-in-singleton.html) - [Slides](http://csharp-video-tutorials.blogspot.com/2017/06/lazy-vs-eager-loading-in-singleton_12.html)   
[Part 6 - Static Class vs Singleton](https://www.youtube.com/watch?v=zlg4jCY2g4o) - [Text](http://csharp-video-tutorials.blogspot.com/2017/06/static-class-vs-singleton.html) - [Slides](http://csharp-video-tutorials.blogspot.com/2017/06/static-class-vs-singleton_17.html)  
  
  
  
In this tutorial we will discuss how to create a simple employee web application using ASP.NET MVC and we will create a custom logger library using Singleton design pattern which logs exceptions to an external file   
  
  
  
Logger Library

**ILog.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Logger

{

    public interface ILog

    {

        void LogException(string message);

    }

}

**Log.cs**

using System;

using System.Collections.Generic;

using System.IO;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Logger

{

    public sealed class Log : ILog

    {

        private Log()

        {

        }

        private static readonly Lazy<Log> instance = new Lazy<Log>(() => new Log());

        public static Log GetInstance

        {

            get

            {

                return instance.Value;

            }

        }

        public void LogException(string message)

        {

            string fileName = string.Format("{0}\_{1}.log", "Exception", DateTime.Now.ToShortDateString());

            string logFilePath = string.Format(@"{0}\{1}", AppDomain.CurrentDomain.BaseDirectory, fileName);

            StringBuilder sb = new StringBuilder();

            sb.AppendLine("----------------------------------------");

            sb.AppendLine(DateTime.Now.ToString());

            sb.AppendLine(message);

            using (StreamWriter writer = new StreamWriter(logFilePath, true))

            {

                writer.Write(sb.ToString());

                writer.Flush();

            }

        }

    }

}

Create and MVC Application and Create EmployeePortal DB with Employee Table

**Employee Table**

CREATE TABLE [dbo].[Employee] (

    [Id]             INT          IDENTITY (1, 1) NOT NULL,

    [Name]           VARCHAR (50) NOT NULL,

    [JobDescription] VARCHAR (50) NOT NULL,

    [Number]         VARCHAR (50) NOT NULL,

    [Department]     VARCHAR (50) NOT NULL,

    PRIMARY KEY CLUSTERED ([Id] ASC)

);

Generate Model using ADO.Net entity model generator using the above Table. Post generation, Add an Employee controller and use generated model which further creates views for Employee which facilitates CRUD operations on the employee.

**Sample EmployeeController.cs**

using Logger;

using System;

using System.Collections.Generic;

using System.Data;

using System.Data.Entity;

using System.Linq;

using System.Net;

using System.Web;

using System.Web.Mvc;

using Web.Models;

namespace Web.Controllers

{

    public class EmployeesController : Controller

    {

        private ILog \_ILog;

        private EmployeePortalEntities db = new EmployeePortalEntities();

        public EmployeesController()

        {

            \_ILog = Log.GetInstance;

        }

        protected override void OnException(ExceptionContext filterContext)

        {

            \_ILog.LogException(filterContext.Exception.ToString());

            filterContext.ExceptionHandled = true;

            this.View("Error").ExecuteResult(this.ControllerContext);

        }

        // GET: Employees

        public ActionResult Index()

        {

            return View(db.Employees.ToList());

        }

        // GET: Employees/Details/5

        public ActionResult Details(int? id)

        {

            if (id == null)

            {

                return new HttpStatusCodeResult(HttpStatusCode.BadRequest);

            }

            Employee employee = db.Employees.Find(id);

            if (employee == null)

            {

                return HttpNotFound();

            }

            return View(employee);

        }

        // GET: Employees/Create

        public ActionResult Create()

        {

            return View();

        }

        // POST: Employees/Create

        // To protect from overposting attacks, please enable the specific properties you want to bind to, for

        // more details see http://go.microsoft.com/fwlink/?LinkId=317598.

        [HttpPost]

        [ValidateAntiForgeryToken]

        public ActionResult Create([Bind(Include = "Id,Name,JobDescription,Number,Department")] Employee employee)

        {

            if (ModelState.IsValid)

            {

                db.Employees.Add(employee);

                db.SaveChanges();

                return RedirectToAction("Index");

            }

            return View(employee);

        }

        // GET: Employees/Edit/5

        public ActionResult Edit(int? id)

        {

            if (id == null)

            {

                return new HttpStatusCodeResult(HttpStatusCode.BadRequest);

            }

            Employee employee = db.Employees.Find(id);

            if (employee == null)

            {

                return HttpNotFound();

            }

            return View(employee);

        }

        // POST: Employees/Edit/5

        // To protect from overposting attacks, please enable the specific properties you want to bind to, for

        // more details see http://go.microsoft.com/fwlink/?LinkId=317598.

        [HttpPost]

        [ValidateAntiForgeryToken]

        public ActionResult Edit([Bind(Include = "Id,Name,JobDescription,Number,Department")] Employee employee)

        {

            if (ModelState.IsValid)

            {

                db.Entry(employee).State = EntityState.Modified;

                db.SaveChanges();

                return RedirectToAction("Index");

            }

            return View(employee);

        }

        // GET: Employees/Delete/5

        public ActionResult Delete(int? id)

        {

            if (id == null)

            {

                return new HttpStatusCodeResult(HttpStatusCode.BadRequest);

            }

            Employee employee = db.Employees.Find(id);

            if (employee == null)

            {

                return HttpNotFound();

            }

            return View(employee);

        }

        // POST: Employees/Delete/5

        [HttpPost, ActionName("Delete")]

        [ValidateAntiForgeryToken]

        public ActionResult DeleteConfirmed(int id)

        {

            Employee employee = db.Employees.Find(id);

            db.Employees.Remove(employee);

            db.SaveChanges();

            return RedirectToAction("Index");

        }

        protected override void Dispose(bool disposing)

        {

            if (disposing)

            {

                db.Dispose();

            }

            base.Dispose(disposing);

        }

    }

}

* Run the application and all the exceptions will be logged under the file created by the logger class library.
* This Proves that singleton design pattern comes handy in the situations where we need to have a single instance of the object.
* Now, to consider another example, we can design Cache Management to use and leverage on Singleton design pattern as we can handle reads and writes to external caches using the Single Cache instance object.

# Dependency injection and IOC

**Inversion of Control (инверсия управления)** — это некий абстрактный принцип, набор рекомендаций для написания слабо связанного кода. Суть которого в том, что каждый компонент системы должен быть как можно более изолированным от других, не полагаясь в своей работе на детали конкретной реализации других компонентов.  
**Dependency Injection (внедрение зависимостей)** — это одна из реализаций этого принципа (помимо этого есть еще [Factory Method](http://en.wikipedia.org/wiki/Factory_pattern), [Service Locator](http://en.wikipedia.org/wiki/Service_locator_pattern)).  
**IoC-контейнер** — это какая-то библиотека, фреймворк, программа если хотите, которая позволит вам упростить и автоматизировать написание кода с использованием данного подхода на столько, на сколько это возможно. Их довольно много, пользуйтесь тем, чем вам будет удобно, я продемонстрирую все на примере [Ninject](http://www.ninject.org/).

## Constructor injection

interface IChild

{

void Help(string message);

}

class Parent

{

IChild child = null;

public Parent(IChild child)

{

this.child = child;

}

public void Notify(string message)

{

if (child == null)

{

//decide which child parent should choose

}

else

{

child.Help(message);

}

}

}

class Child1 : IChild

{

public void Help(string message)

{

Console.WriteLine("I am child 1 -"+ message);

}

}

class Child2 : IChild

{

public void Help(string message)

{

Console.WriteLine("I am child 2 -" + message);

}

}

class Program

{

static void Main(string[] args)

{

Child1 child1 = new Child1();

Child2 child2 = new Child2();

Parent parent = new Parent(child1);

parent.Notify("Azaza");

}

}

## Method injection

interface IChild

{

void Help(string message);

}

class Parent

{

public void Notify(IChild child, string message)

{

if (child == null)

{

//decide which child parent should choose

}

else

{

child.Help(message);

}

}

}

class Child1 : IChild

{

public void Help(string message)

{

Console.WriteLine("I am child 1 -"+ message);

}

}

class Child2 : IChild

{

public void Help(string message)

{

Console.WriteLine("I am child 2 -" + message);

}

}

## Property injection

class Parent

{

public IChild Child { set; get; }

public void Notify(string message)

{

if (Child== null)

{

//decide which child parent should choose

}

else

{

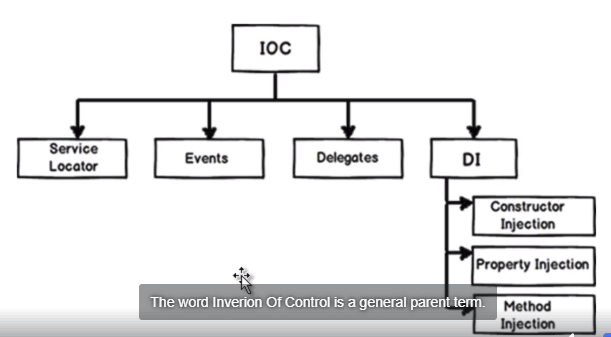
Child.Help(message);

}

}

}

## IOC



interface IDal

{

void Add();

}

class Customer

{

IDal dal = null;

public Customer(IDal dal)

{

this.dal = dal;

}

public void Add()

{

dal.Add();

}

}

class SqlServer : IDal

{

public void Add()

{

}

}

class Oracle : IDal

{

public void Add()

{

}

}

class Program

{

static void Main(string[] args)

{

Customer customer = new Customer(new SqlServer());

customer.Add();

}

}