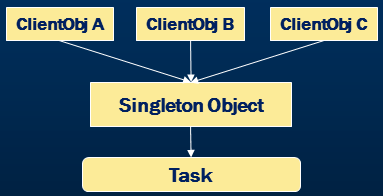
**Singleton Design Pattern**

**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

**Singleton Pattern** belongs to **Creational type pattern**.

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

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.

### Thread Safety in Singleton

**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

**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

**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

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.