**Section06 Singleton Design Pattern**

**Notes:-**

**1-Singleton is component which is instantiated only once**

**2-singelton is used for**

**A-some components it only make sense to have one in the system.**

**\*Database repository: you want single instance of context to access to the database**

**\*Object factory**

**B-constructor call is expensive**

**\*we only do it once**

**\*we provide everyone with the same instance**

**C-Need to take care of lazy initiate and thread safety**

**Lesson01 Singleton Implementation**

**Notes:-**

**1-Singleton is used to initialize by set the constructor private and using static property of the same class with using lazy to initiate whenever it call**

**//we define interface with implement inside the SingletonDatabase**

**namespace SingletonPatternSol.Interfaces{**

**public interface IDatabase{**

**int GetPopulation(string name);}}**

**using System;**

**using System.Collections.Generic;**

**using System.IO;**

**using System.Linq;**

**using System.Security.Cryptography;**

**using System.Text;**

**using System.Threading.Tasks;**

**using Autofac;**

**using MoreLinq;**

**using SingletonPatternSol.Interfaces;**

**using static System.Console;**

**namespace SingletonPatternSol.Models{**

**public class SingletonDatabase : IDatabase{**

**private Dictionary<string, int> capitals;**

**private static int instanceCount;**

**public static int Count => instanceCount;**

**private SingletonDatabase(){**

**WriteLine("Initializing database");**

**//by using the MoreLinq**

**//we read the capital.txt file into key , value pairs as dictionary**

**capitals = File.ReadAllLines(**

**Path.Combine(**

**new FileInfo(typeof(IDatabase).Assembly.Location).DirectoryName, "capitals.txt"))**

**.Batch(2).ToDictionary(**

**list => list.ElementAt(0).Trim(),**

**list => int.Parse(list.ElementAt(1)));}**

**public int GetPopulation(string name){return capitals[name];}**

**// laziness + thread safety not calling until it call through static property**

**private static Lazy<SingletonDatabase> instance = new Lazy<SingletonDatabase>(() =>{**

**instanceCount++;**

**return new SingletonDatabase();});**

**public static IDatabase Instance => instance.Value;}}**

**Lesson02 Problem of Singleton Design Pattern**

**Notes:-**

**1-the problem of the singleton is that you have to use the same static property instance as below**

**//we install the following nugget pacakges**

* **NUnit (3.12.0)**
* **NUnit3TestAdapter (3.17.0)**
* **Microsoft.NET.Test.Sdk (16.7.1)**

**using SingletonPatternSol.Interfaces;**

**using System.Collections.Generic;**

**namespace SingletonPatternSol.Testing{**

**public class DummyDatabase : IDatabase{**

**public int GetPopulation(string name){**

**return new Dictionary<string, int>{**

**["alpha"] = 1,**

**["beta"] = 2,**

**["gamma"] = 3**

**}[name];}}}**

**using SingletonPatternSol.Models;**

**using System.Collections.Generic;**

**namespace SingletonPatternSol.Testing{**

**public class SingletonRecordFinder{**

**public int TotalPopulation(IEnumerable<string> names){**

**int result = 0;**

**foreach (var name in names)**

**result += SingletonDatabase.Instance.GetPopulation(name);**

**return result;}}}**

**using SingletonPatternSol.Interfaces;**

**using System.Collections.Generic;**

**namespace SingletonPatternSol.Testing{**

**public class ConfigurableRecordFinder{**

**private IDatabase database;**

**public ConfigurableRecordFinder(IDatabase database){this.database = database;}**

**public int GetTotalPopulation(IEnumerable<string> names){**

**int result = 0;**

**foreach (var name in names)**

**result += database.GetPopulation(name);**

**return result;}}}**

**using NUnit.Framework;**

**using SingletonPatternSol.Models;**

**namespace SingletonPatternSol.Testing{**

**/// <summary>**

**/// IMPORTANT: be sure to turn off shadow copying for unit tests in R#!**

**/// </summary>**

**[TestFixture]**

**public class SingletonTests{**

**//to check if the singleton generate only 1 instance**

**[Test]**

**public void IsSingletonTest(){**

**var db = SingletonDatabase.Instance;**

**var db2 = SingletonDatabase.Instance;**

**Assert.That(db, Is.SameAs(db2));**

**Assert.That(SingletonDatabase.Count, Is.EqualTo(1));}**

**//to read from SingeltonDatabase and apply TotalPopulation**

**[Test]**

**public void SingletonTotalPopulationTest(){**

**// testing on a live database**

**var rf = new SingletonRecordFinder();**

**var names = new[] { "Seoul", "Mexico City" };**

**int tp = rf.TotalPopulation(names);**

**Assert.That(tp, Is.EqualTo(17500000 + 17400000));}**

**//to read from Dummy Database and assign as parameter IDatabase**

**//to load the GetTotalPopuplation that implement on both**

**//DummyDatabase and SingletonDatabase**

**[Test]**

**public void DependantTotalPopulationTest(){**

**var db = new DummyDatabase();**

**var rf = new ConfigurableRecordFinder(db);**

**Assert.That(**

**rf.GetTotalPopulation(new[] { "alpha", "gamma" }),**

**Is.EqualTo(4));}}}**

**Lesson03 Singleton using Dependency Injection**

**Notes:-**

**1-with using Autofac we can register all the types we wants as Singleton or new instance as below**

**using System;**

**namespace SingletonDISol{**

**public class Foo{**

**public EventBroker Broker;**

**public Foo(EventBroker broker){**

**Broker = broker ?? throw new ArgumentNullException(paramName: nameof(broker));}}**

**public class EventBroker{}}**

**public class Program{**

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

**//it will register EventBroker as singelton and Foo and new instance**

**var builder = new ContainerBuilder();**

**builder.RegisterType<EventBroker>().SingleInstance();**

**builder.RegisterType<Foo>();**

**using (var c = builder.Build()){**

**var foo1 = c.Resolve<Foo>();**

**var foo2 = c.Resolve<Foo>();**

**//we see that the foo1 , foo2 is differnt but the broker is singleton so its refer to the same**

**WriteLine(ReferenceEquals(foo1, foo2));**

**WriteLine(ReferenceEquals(foo1.Broker, foo2.Broker));}}}**

**Lesson04 MonoState**

**Notes:-**

**1-the issue also with Singleton is the difficult with testing because you cannot call constructor directly but within the static field that refer to the same class instance**

**2-there is alternative way of the Singleton is called MonoState which talk about that allow you to create instance from the constructor but with using static fields inside it so if one of the class instances change value it will reflect to the others and so on.**

**namespace MonoStatePro{**

**public class CEO{**

**//we see that with static fields all instance refer to the same property becouse they are static**

**private static string name;**

**private static int age;**

**public string Name{get { return name; }set { name = value; }}**

**public int Age{get { return age; }set { age = value; }}**

**public override string ToString(){return $"{nameof(Name)} : {Name} , {nameof(Age)} : {Age}";}}}**

**using static System.Console;**

**namespace MonoStatePro{**

**class Program{**

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

**var ceo = new CEO();**

**ceo.Name = "Adam";**

**ceo.Age = 22;**

**//we see that the both instances refer to the same static fields , so you can initiate as you want of //instances but all these instances refer to the same static fields**

**var ceo2 = new CEO();**

**WriteLine(ceo2);**

**ReadLine();}}}**

**Lesson05 Thread Singleton**

**Notes:-**

**1-instead of having single instance of the entire application, you can have instance per thread (by using ThreadLocal)**

**using System.Threading;**

**namespace PreThreadPro{**

**public sealed class PerThreadSingleton{**

**public int Id;**

**private PerThreadSingleton(){Id = Thread.CurrentThread.ManagedThreadId;}**

**//we apply the singelton per thread so every thread has its own single instance**

**private static ThreadLocal<PerThreadSingleton> threadInstance**

**= new ThreadLocal<PerThreadSingleton>(() => new PerThreadSingleton());**

**public static PerThreadSingleton Instance => threadInstance.Value;}}**

**using System.Threading.Tasks;**

**using static System.Console;**

**namespace PreThreadPro{**

**class Program{**

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

**//we see that every thread has its own instance**

**var t1 = Task.Factory.StartNew(() =>{WriteLine($"t1: " + PerThreadSingleton.Instance.Id);});**

**var t2 = Task.Factory.StartNew(() =>{**

**WriteLine($"t2: " + PerThreadSingleton.Instance.Id);**

**WriteLine($"t2 again: " + PerThreadSingleton.Instance.Id);});**

**Task.WaitAll(t1, t2);**

**ReadLine();}}}**

**Lesson06 Ambient Context**

**Notes:-**

**1`-this approach is not thread safe when you working on multiple threads**

**namespace AmbientContextPro{**

**public struct Point{**

**private int X, Y;**

**public Point(int x, int y){X = x;Y = y;}**

**public override string ToString(){return $"{nameof(X)}: {X}, {nameof(Y)}: {Y}";}}}**

**using System;**

**namespace AmbientContextPro{**

**public class Wall{**

**public Point Start, End;**

**public int Height;**

**public const int UseAmbient = Int32.MinValue;**

**public Wall(Point start, Point end){**

**Start = start;**

**End = end;**

**Height = BuildingContext.Current.WallHeight;}**

**public override string ToString(){**

**return $"{nameof(Start)}: {Start}, {nameof(End)}: {End}, " +**

**$"{nameof(Height)}: {Height}";}}}**

**using System.Collections.Generic;**

**using System.Text;**

**namespace AmbientContextPro{**

**public class Building{**

**public readonly List<Wall> Walls = new List<Wall>();**

**public override string ToString(){**

**var sb = new StringBuilder();**

**foreach (var wall in Walls)**

**sb.AppendLine(wall.ToString());**

**return sb.ToString();}}}**

**//on the Building Context we apply the Ambient Context by make stack property of the same class //instance with contain the height property and using Current to get the latest one**

**using System;**

**using System.Collections.Generic;**

**namespace AmbientContextPro{**

**// non-thread-safe global context**

**//the idea of the Ambient Context is to provide shard property accross group of instance**

**//so we use IDisposable it will pop the entered shared instance from the stack**

**public sealed class BuildingContext : IDisposable{**

**public int WallHeight = 0;**

**public int WallThickness = 300; // etc.**

**private static Stack<BuildingContext> stack**

**= new Stack<BuildingContext>();**

**static BuildingContext(){**

**// ensure there's at least one state**

**stack.Push(new BuildingContext(0));}**

**public BuildingContext(int wallHeight){**

**WallHeight = wallHeight;**

**stack.Push(this);}**

**//it will get the most shared instance**

**public static BuildingContext Current => stack.Peek();**

**public void Dispose(){**

**// not strictly necessary**

**if (stack.Count > 1)**

**stack.Pop();}}}**

**//on the main entry point**

**using static System.Console;**

**namespace AmbientContextPro{**

**class Program{**

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

**var house = new Building();**

**//each group of instance has its own shared instance (ground floor)**

**house.Walls.Add(new Wall(new Point(0, 0), new Point(5000, 0)/\*, e\*/));**

**house.Walls.Add(new Wall(new Point(0, 0), new Point(0, 4000)/\*, e\*/));**

**//each group of instance has its own shared instance (first floor)**

**using (new BuildingContext(3500)){**

**house.Walls.Add(new Wall(new Point(0, 0), new Point(5000, 0) /\*, e\*/));**

**house.Walls.Add(new Wall(new Point(0, 0), new Point(0, 4000) /\*, e\*/));}**

**//each group of instance has its own shared instance (ground again)**

**house.Walls.Add(new Wall(new Point(5000, 0), new Point(5000, 4000)/\*, e\*/));**

**WriteLine(house);}}}**