Source: C# Corner (www.c-sharpcorner.com) PRINT

Article

Singleton Design Pattern with Java

By **Aman Gupta** on **Updated date Sep 23, 2020**

Design Pattern

In the last article, we learned the use of static keywords here in Java. So here is the practical implementation.

Design Pattern provides a simple solution to common problems that are faced in day to day life by software developers.Three basic types of design patterns exist, which are listed below,

- Creational Pattern
- Structural Pattern and,
- Behavioral Pattern

Along with these, there are normally 23 design patterns that are classified above in three patterns.

Thus the pattern described here is like a blueprint that you can customize to solve a particular design problem and they only differ by complexity and scalability of the code.
So here in this article, we will learn about the Singleton design pattern. Let’s get started.

Singleton Design Pattern

As the name suggests “single” i.e has only one instance and provides a global instance.

Singleton design pattern restricts the instantiation of a class and ensures that one object of a class exists in JVM(Java Virtual Machine).

Let's have a class “CsharpCorner” and we create a reference for this class.

Creating more than one instance means both reference variables will have different values stored. Thus it is “NOT” Singleton object. Below is the scenario, here c1 and c2 are two reference variables carrying different values.

```
package SingletonDesign;

public class CsharpCorner {
    public static void main(String[] args) {
        CsharpCorner c1 = new CsharpCorner();
        CsharpCorner c2 = new CsharpCorner();
    }
}
```

Singleton design pattern is divided into two types,

1. Eager Loading: creation at load time.
2. Lazy Loading: the creation of an object whenever needed.

Both eager and lazy design patterns differ in the way of returning the object, if an object is null it returns a new singleton object or other particular objects.

So we are now starting with eager loading and for which we need to remember some rules for constructing eager loading objects.

Eager loading singleton object

To make any class Singleton the first thing that comes to the mind is the Singleton keyword but no singleton is a concept in Java which can only be achieved by learning some below-mentioned steps.

- Default constructor as private.
- Create a private static class type variable and initialize it with null.
- Initialise variable(in step 2) to constructor of class type(here Singleton).
- Create a public static getObject or getInstance method and return a class object.

Here is the example which implements these rules for object creation,

```
package SingletonDesign;

class Singleton {
    //Eager Loading Singleton object
    //rule 1.2
    private static Singleton obj = null;

    // rule 1.1
    private Singleton() {
        //rule 1.3
        obj = new Singleton();
    }

    //rule 1.4
    public static Singleton getInstance() {
        return obj;
    }
}
public class CsharpCorner {
    public static void main(String[] args) {
        Singleton c1 = Singleton.getInstance();
        Singleton c2 = Singleton.getInstance();
    }
}
```

In the above example, the very first step is to create a private default constructor as a private constructor does not allow new objects to be created.

The second step is to create a private static class type variable, this variable is to be static as it’s been used inside the constructor for object creation.

The third step is to create the object using a class type variable obtained from the second step.

In the fourth and last step, we create a getInstance() method to return the object of the singleton class. Since getInstance() is a static method that every time we call will create only one instance of the class.

Point to remember

The above example is called eager loading, when we observe we find some drawback in this; the reference object “obj” is of type static, which means this object will be created and is available in memory when class is loaded, so this is a global variable because whenever a class is loaded that object is available in local memory. That’s the drawback, as, if this object is bulky then it is just a waste of memory and processing power. To overcome this we use the concept of lazy loading.

Note
If either we try to make more objects of Singleton class this is not possible now as one object is created earlier and we can’t get more than one instance in the singleton design pattern.

Lazy loading singleton object

To make any class Singleton the first thing that comes to mind is the Singleton keyword but no singleton is a concept in Java which can only be achieved by learning some below-mentioned steps.

- Default constructor as private.
- Create a private static class type variable and initialize it with null.
- Create a public static getObject or getInstance method and return a class object.

Here is the example which implements these rules for object creation,

```
package SingletonDesign;

class Singleton {
    // Lazy Loading Singleton
    //rule 2.2
    private static Singleton obj;

    // rule 2.1
    private Singleton() {

    }

    //rule 2.3
    public static Singleton getInstance() {
        if (obj == null) {
            obj = new Singleton();
        }
        return obj;
    }
}
public class CsharpCorner {
    public static void main(String[] args) {
        Singleton c1 = Singleton.getInstance();
    }
}
```

This technique overcomes the drawback of eager loading by creating an object at the time of calling the getInstance() method.

In the above example, the very first step is to create a private default constructor as a private constructor does not allow new objects to be created.

The second step is to create a private static class type variable, this variable is to be static as it’s been used inside the constructor for object creation.

In the third and last step, we create a getInstance() method to return the object of the singleton class on the basis of a condition.

Note
If either we try to make more objects of Singleton class this is not possible now as an object is created at the time of getInstance() method is called and we can’t get more than one instance in the singleton design pattern.

Synchronized with Thread in Lazy Loading

To overcome the drawbacks with the above implementation we used. So when we have more than one instance then it will execute the things sequentially and for the first time when we call getInstance() the object is null thus a new object is created but when we call it for the second time we already have an object so it will not be executed and return an older object.

```
package SingletonDesign;

class Singleton {
    private static Singleton obj;

    private Singleton() {

    }

    public static synchronized Singleton getInstance() //check
    {
        if (obj == null) {
            obj = new Singleton();
        }
        return obj;
    }
}

public class CsharpCorner {
    public static void main(String[] args) {
        Thread t1 = new Thread(new Runnable() {
            public void run() {
                Singleton c1 = Singleton.getInstance();
            }
        });

        Thread t2 = new Thread(new Runnable() {
            public void run() {
                Singleton c1 = Singleton.getInstance();
            }
        });

        t1.start();
        t2.start();
    }
}
```

So in the above example, we have two threads, t1 and t2, and both are calling the same method and we start both at the same time.

Note
With thread, we can use synchronized with getInstance() method, making a method synchronized results in lots of work as here a huge amount of work is done by getInstance() method and it will decrease the performance by a factor of 100 and that's the issue with synchronized.

Using Thread with Lazy Loading (Double Check Locking Technique)

To overcome the drawbacks (time complexity) with the above implementation, we can use a concept of Double-Check Locking; it means we will check object value for two times i.e

1. SImply i.e (if(obj == null))
2. Inside Synchronized Block

To overcome the time complexity by synchronizing with the getInstance() method we can introduce synchronized while creating objects. So, instead of waiting for 100 milliseconds, we can wait for 5 milliseconds.

So here is the example,

```
package SingletonDesign;

class Singleton {
    private static Singleton obj;

    private Singleton() {

    }

    public static synchronized Singleton getInstance() //simple check
    {
        if (obj == null) {
            synchronized(Singleton.class) //double check loading
            {
                if (obj == null)
                    obj = new Singleton();
            }
        }
        return obj;
    }
}

public class CsharpCorner {
    public static void main(String[] args) {
        Thread t1 = new Thread(new Runnable() {
            public void run() {
                Singleton c1 = Singleton.getInstance();
            }
        });

        Thread t2 = new Thread(new Runnable() {
            public void run() {
                Singleton c1 = Singleton.getInstance();
            }
        });

        t1.start();
        t2.start();
    }
}
```

Enum with Singleton Pattern

To overcome the drawbacks of all four above implementations we have a new technique to create a singleton pattern.

From Java 1.5 we have one method to create a singleton design pattern and that method is thread-safe and utilizes fewer resources. This method will only work when we use the Java version above or 1.5.

Syntax

```
enum XYZ {
   INSTANCE; //inbuilt private constructor
}
```

Here's the example of using enum (a special type of a class).

```
package SingletonDesign;

enum Singleton {
    INSTANCE; //inbuilt private constructor
    int x;
    public void show() {
        System.out.println(x);
    }
}

public class CsharpCorner {
    public static void main(String[] args) {
        Singleton object1 = Singleton.INSTANCE;
        object1.x = 10;
        object1.show();

        Singleton object2 = Singleton.INSTANCE;
        object2.x = 20;
        object1.show();
    }
}
```

The above code will result in output as 10 and 20 which shows singleton object implementation.

```
10
20
```

Even though we are working with Double-Check Locking(DCL) we have a concept of deserialization. So, even if the class is a singleton, the readObject() method will give you a new object.

Note
So while working with enum we have a method called readResolve() it will not create the new object and will use the current object only.

Summary

Thus, coming towards the end of this article we had learned a singleton design pattern with eager and lazy loading techniques along with implementation with thread, double-check locking and enum.

What did we learn?

- What is the design pattern?
- Why design patterns?
- Types of design patterns.
- Singleton design pattern.
- Eager and lazy loading techniques.
- Synchronized with thread in the singleton design pattern.
- Double-check locking in the singleton design pattern.
- Singleton design pattern with ENUM.