# Design Pattern

## Singleton Design Pattern

We have various ways of creating singletons in Java. Now, first of all, what is Singleton and why is it required?

The singleton design pattern is used to restrict the instantiation of a class and ensures that only one instance of the class exists in the [JVM](https://dzone.com/articles/jvm-architecture-explained). In other words, a singleton class is a class that can have only one object (an instance of the class) at a time per JVM instance. There are various ways to design/code a singleton class.

1. ****Class-level Member (Eager Initialization Method):****
   1. Make constructor private.
   2. Make a private constant static instance (class-member) of this Singleton class.
   3. Write a static/factory method that returns the object of the singleton class that we have created as a class-member instance.
   4. We can also mark a static member as public to access constant static instance directly. But, I like to access class/instance members via methods only.
   5. So, the singleton class is different from a normal Java class in terms of instantiation. For a normal class, we use a constructor, whereas for singleton class we use the getInstance()method.

1

public class SingletonClass {

2

private static final SingletonClass SINGLE\_INSTANCE = new SingletonClass();

3

private SingletonClass() {}

4

public static SingletonClass getInstance() {

5

return SINGLE\_INSTANCE;

6

}

7

}

1. ****Class-level Member (Lazy Initialization Method):****
   1. Make constructor as private.
   2. Make a private static instance (class-member) of this singleton class. But, DO NOT instantiate it.
   3. Write a static/factory method that checks the static instance member for null and creates the instance. At last, it returns an object of the singleton class.

1

public class SingletonClass {

2

private static SingletonClass SINGLE\_INSTANCE = null;

3

private SingletonClass() {}

4

public static SingletonClass getInstance() {

5

if (SINGLE\_INSTANCE == null) {

6

synchronized(SingletonClass.class) {

7

SINGLE\_INSTANCE = new SingletonClass();

8

}

9

}

10

return SINGLE\_INSTANCE;

11

}

12

}

1. ****Class-level Member (Lazy Initialization with double lock Method):****
   1. Here, we run into a problem. Suppose that there are two threads running. Both can get inside of the if statement concurrently when the instance is null. Then, one thread enters the synchronized block to initialize the instance, while the other is blocked. When the first thread exits in the synchronized block, the waiting thread enters and creates another singleton object. Note that when the second thread enters the synchronized block, it does not check to see if the instance is non-null.

1

public class SingletonClass {

2

private static SingletonClass SINGLE\_INSTANCE = null;

3

private SingletonClass() {}

4

public static SingletonClass getInstance() {

5

if (SINGLE\_INSTANCE == null) {

6

synchronized (SingletonClass.class) {

7

if (SINGLE\_INSTANCE == null) {

8

SINGLE\_INSTANCE = new SingletonClass();

9

}

10

}

11

}

12

return SINGLE\_INSTANCE;

13

}

14

}

1. ****By using nested Inner class (Lazy Load method):****
   1. In this method is based on the Java Language Specifications (JLS). Java Virtual Machine loads static data-members only on-demand. So, here the class SingletonClass loads at first by the JVM. Since there is no static data memberin the class; SingletonClassHolder does not loads or creates SINGLE\_INSTANCE.
   2. This will happen only when we invoke getIntance method. JLS guaranteed the sequential execution of the class initialization; that means thread-safe. So, we actually do not need to provide explicit synchronization on static getInstance() method for loading and initialization. Here, since the initialization creates the static variable SINGLE\_INSTANCE in a sequential way, all concurrent invocations of the getInstance() will return the same correctly initialized SINGLE\_INSTANCE without synchronization overhead.

1

public class SingletonClass {

2

private SingletonClass() {}

3

private static class SingletonClassHolder {

4

static final Something SINGLE\_INSTANCE = new SingletonClass();

5

}

6

public static SingletonClass getInstance() {

7

return SingletonClassHolder.SINGLE\_INSTANCE;

8

}

9

}

1. ****By using Enums:****All of the above approaches are not full-proof in all the cases. We can still create multiple instances of the above implementations by using serialization or reflection. In both of the cases, we can bypass the private constructor and, hence, can easily create multiple instances. So, the new approach is to create singleton class by using enums since enums fields are compiled time constants, but they are instances of their enum type. And, they're constructed when the enum type is referenced for the first time.