



CONTENTS

Singleton Pattern Principles

Java Singleton Pattern Implementation

1. Eager initialization
2. Static block initialization
3. Lazy Initialization
4. Thread Safe Singleton
5. Bill Pugh Singleton Implementation
6. Using Reflection to destroy Singleton Pattern
7. Enum Singleton
8. Serialization and Singleton

Conclusion

// Tutorial //

Java Singleton Design Pattern Best Practices with Examples

Updated on November 5, 2022





By Pankaj

Singleton Pattern

1. GoF Creational Pattern
2. Only one instance of class
3. Must have global access point to create the instance



Introduction

Java Singleton Pattern is one of the [Gangs of Four Design patterns](#) and comes in the *Creational Design Pattern* category. From the definition, it seems to be a straightforward design pattern, but when it comes to implementation, it comes with a lot of concerns.

In this article, we will learn about singleton design pattern principles, explore different ways to implement the singleton design pattern, and some of the best practices for its usage.

Singleton Pattern Principles



Singleton pattern restricts the instantiation of a class and ensures that only one instance of the class exists in the Java Virtual Machine.

Singleton class must provide a global access point to get the instance of the class.

- Singleton pattern is used for [logging](#), drivers objects, caching, and [thread pool](#).



- Singleton design pattern is also used in other design patterns like [Abstract Factory](#), [Builder](#), [Prototype](#), [Facade](#), etc.
- Singleton design pattern is used in core Java classes also (for example, `java.lang.Runtime`, `java.awt.Desktop`).

Java Singleton Pattern Implementation

To implement a singleton pattern, we have different approaches, but all of them have the following common concepts.

- Private constructor to restrict instantiation of the class from other classes.
- Private static variable of the same class that is the only instance of the class.
- Public static method that returns the instance of the class, this is the global access point for the outer world to get the instance of the singleton class.

In further sections, we will learn different approaches to singleton pattern implementation and design concerns with the implementation.

1. Eager initialization

In eager initialization, the instance of the singleton class is created at the time of class loading. The drawback to eager initialization is that the method is created even though the client application might not be using it. Here is the implementation of the static initialization singleton class:

```
package com.journaldev.singleton;

public class EagerInitializedSingleton {

    private static final EagerInitializedSingleton instance = new EagerInitializedSi

    // private constructor to avoid client applications using the constructor
    private EagerInitializedSingleton(){}

    public static EagerInitializedSingleton getInstance() {
        return instance;
    }
}
```

Copy



If your singleton class is not using a lot of resources, this is the approach to use. But in most of the scenarios, singleton classes are created for resources such as File System, Database connections, etc. We should avoid the instantiation unless the client calls the `getInstance` method. Also, this method doesn't provide any options for exception handling.

2. Static block initialization

Static block initialization implementation is similar to eager initialization, except that instance of the class is created in the static block that provides the option for **exception handling**.

```
package com.journaldev.singleton;

public class StaticBlockSingleton {

    private static StaticBlockSingleton instance;

    private StaticBlockSingleton(){}

    // static block initialization for exception handling
    static {
        try {
            instance = new StaticBlockSingleton();
        } catch (Exception e) {
            throw new RuntimeException("Exception occurred in creating singleton instance");
        }
    }

    public static StaticBlockSingleton getInstance() {
        return instance;
    }
}
```

Copy

Both eager initialization and static block initialization create the instance even before it's been requested and that is not the best practice to use.

3. Lazy Initialization



Lazy initialization method to implement the singleton pattern creates the instance in the global access method. Here is the sample code for creating the singleton class with this approach:

```
package com.journaldev.singleton;

public class LazyInitializedSingleton {

    private static LazyInitializedSingleton instance;

    private LazyInitializedSingleton(){}

    public static LazyInitializedSingleton getInstance() {
        if (instance == null) {
            instance = new LazyInitializedSingleton();
        }
        return instance;
    }
}
```

[Copy](#)

The preceding implementation works fine in the case of the single-threaded environment, but when it comes to multi-threaded systems, it can cause issues if multiple threads are inside the `if` condition at the same time. It will destroy the singleton pattern and both threads will get different instances of the singleton class. In the next section, we will see different ways to create a [thread-safe](#) singleton class.

4. Thread Safe Singleton

A simple way to create a thread-safe singleton class is to make the global access method [synchronized](#) so that only one thread can execute this method at a time. Here is a general implementation of this approach:

```
package com.journaldev.singleton;

class ThreadSafeSingleton {

    private static ThreadSafeSingleton instance;
```

[Copy](#)

```
private ThreadSafeSingleton(){}

public static synchronized ThreadSafeSingleton getInstance() {
    if (instance == null) {
        instance = new ThreadSafeSingleton();
    }
    return instance;
}

}
```



The preceding implementation works fine and provides thread-safety, but it reduces the performance because of the cost associated with the synchronized method, although we need it only for the first few threads that might create separate instances. To avoid this extra overhead every time, *double-checked locking* principle is used. In this approach, the synchronized block is used inside the `if` condition with an additional check to ensure that only one instance of a singleton class is created. The following code snippet provides the double-checked locking implementation:

```
public static ThreadSafeSingleton getInstanceUsingDoubleLocking() {
    if (instance == null) {
        synchronized (ThreadSafeSingleton.class) {
            if (instance == null) {
                instance = new ThreadSafeSingleton();
            }
        }
    }
    return instance;
}
```

[Copy](#)

Continue your learning with [Thread Safe Singleton Class](#).

5. Bill Pugh Singleton Implementation

Prior to Java 5, the Java memory model had a lot of issues, and the previous approaches us  in certain scenarios where too many threads tried to get the instance of the singleton class simultaneously. So [Bill Pugh](#) came up with a different approach to create the singleton class using an [inner static helper class](#). Here is an example of the Bill Pugh Singleton implementation: 

```
package com.journaldev.singleton;

public class BillPughSingleton {

    private BillPughSingleton(){}

    private static class SingletonHelper {
        private static final BillPughSingleton INSTANCE = new BillPughSingleton();
    }

    public static BillPughSingleton getInstance() {
        return SingletonHelper.INSTANCE;
    }
}
```

Copy

Notice the *private inner static class* that contains the instance of the singleton class. When the singleton class is loaded, `SingletonHelper` class is not loaded into memory and only when someone calls the `getInstance()` method, this class gets loaded and creates the singleton class instance. This is the most widely used approach for the singleton class as it doesn't require synchronization.

6. Using Reflection to destroy Singleton Pattern

Reflection can be used to destroy all the previous singleton implementation approaches. Here is an example class:

```
package com.journaldev.singleton;

import java.lang.reflect.Constructor;

public class ReflectionSingletonTest {

    public static void main(String[] args) {
        EagerInitializedSingleton instanceOne = EagerInitializedSingleton.getInstance();
        EagerInitializedSingleton instanceTwo = null;

        try {
            Constructor[] constructors = EagerInitializedSingleton.class.getDeclaredConstructors();
            for (Constructor constructor : constructors) {
```

Copy

```
        // This code will destroy the singleton pattern
        constructor.setAccessible(true);
        instanceTwo = (EagerInitializedSingleton) constructor.newInstance();
        break;
    }
} catch (Exception e) {
    e.printStackTrace();
}
System.out.println(instanceOne.hashCode());
System.out.println(instanceTwo.hashCode());
}
}
```

When you run the preceding test class, you will notice that `hashCode` of both instances is not the same which destroys the singleton pattern. Reflection is very powerful and used in a lot of frameworks like Spring and Hibernate. Continue your learning with [Java Reflection Tutorial](#).

7. Enum Singleton

To overcome this situation with Reflection, [Joshua Bloch](#) suggests the use of `enum` to implement the singleton design pattern as Java ensures that any `enum` value is instantiated only once in a Java program. Since [Java Enum](#) values are globally accessible, so is the singleton. The drawback is that the `enum` type is somewhat inflexible (for example, it does not allow lazy initialization).

```
package com.journaldev.singleton;
```

[Copy](#)

```
public enum EnumSingleton {

    INSTANCE;

    public static void doSomething() {
        // do something
    }
}
```



8. Serialization and Singleton

Sometimes in distributed systems, we need to implement `Serializable` interface in the singleton class so that we can store its state in the file system and retrieve it at a later point in time. Here is a small singleton class that implements `Serializable` interface also:

```
package com.journaldev.singleton;

import java.io.Serializable;

public class SerializedSingleton implements Serializable {

    private static final long serialVersionUID = -7604766932017737115L;

    private SerializedSingleton(){}

    private static class SingletonHelper {
        private static final SerializedSingleton instance = new SerializedSingleton(
        )

        public static SerializedSingleton getInstance() {
            return SingletonHelper.instance;
        }
    }
}
```

Copy

The problem with serialized singleton class is that whenever we deserialize it, it will create a new instance of the class. Here is an example:

```
package com.journaldev.singleton;

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInput;
import java.io.ObjectInputStream;
import java.io.ObjectOutput;
import java.io.ObjectOutputStream;
```

Copy

```
public class SingletonSerializedTest {  
  
    public static void main(String[] args) throws FileNotFoundException, IOException {  
        SerializedSingleton instanceOne = SerializedSingleton.getInstance();  
        ObjectOutput out = new ObjectOutputStream(new FileOutputStream(  
            "filename.ser"));  
        out.writeObject(instanceOne);  
        out.close();  
  
        // deserialize from file to object  
        ObjectInput in = new ObjectInputStream(new FileInputStream(  
            "filename.ser"));  
        SerializedSingleton instanceTwo = (SerializedSingleton) in.readObject();  
        in.close();  
  
        System.out.println("instanceOne hashCode="+instanceOne.hashCode());  
        System.out.println("instanceTwo hashCode="+instanceTwo.hashCode());  
  
    }  
  
}
```

That code produces this output:

Output

```
instanceOne hashCode=2011117821  
instanceTwo hashCode=109647522
```

So it destroys the singleton pattern. To overcome this scenario, all we need to do is provide the implementation of `readResolve()` method.

```
protected Object readResolve() {  
    return getInstance();  
}
```

Copy

After you will notice that `hashCode` of both instances is the same in the test program.



Read about [Java Serialization](#) and [Java Deserialization](#).

Conclusion

Thanks for learning with the DigitalOcean Community. Check out our offerings for compute, storage, networking, and managed databases. This article covered the singleton design pattern.

[Learn more about us](#) →
Continue your learning with more [Java tutorials](#).

About the authors



Pankaj Author



[Bradley Kouchi](#) Editor

Still looking for an answer?

Ask a question

Search for more help



Was this helpful?

Yes

No



Comments

JournalDev  • March 28, 2021 

I mean if you added volatile to the singleton it would not get destroyed by the reflection pattern Use double-check locking principle when there is not so many thread use bill pugh singleton implementation when you have many threads private static volatile DBSingleton instance = null; private DBSingleton()
{ if(instance != null){ throw new RuntimeException("Use getInstance() method to create"); } } public static DBSingleton getInstance() { if(instance == null){ synchronized (DBSingleton.class){ if (instance == null){ instance = new DBSingleton(); } } } return instance; } }

- Islam

JournalDev  • March 1, 2021 

Double checked locking strategy does not work in multi thread environment due to compiler optimization in java please see <http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html>

- Vishwam

JournalDev  • March 10, 2020 

Wonderful explanation, thank you so much!

- saravan

 JournalDev  • January 22, 2020 



Wrong way to implement double-lock. `Instance = new ThreadSafeSingleton();` It's possible to reorder. Returns without initializing the object.

- Thomas

JournalDev  • November 16, 2019 

> Prior to Java 5, java memory model had a lot of issues and the above approaches used to fail in certain scenarios where too many threads try to get the instance of the Singleton class simultaneously. Could you give an example? How about we make the `instance` volatile? Will this avoid above issues? ```
public class ThreadSafeSingleton { private static volatile ThreadSafeSingleton instance; private ThreadSafeSingleton(){} public static ThreadSafeSingleton getInstanceUsingDoubleLocking(){ if(instance == null){ synchronized (ThreadSafeSingleton.class) { if(instance == null){ instance = new ThreadSafeSingleton(); } } } return instance; } } ```

- Jiabin

JournalDev  • May 13, 2019 

Hi Pankaj, I follow your articles. I think the below code is thread safe as the object will be created only once. `package com.journaldev.singleton; public class EagerInitializedSingleton { private static final EagerInitializedSingleton instance = new EagerInitializedSingleton(); //private constructor to avoid client applications to use constructor private EagerInitializedSingleton(){} public static EagerInitializedSingleton getInstance(){ return instance; } }` Please clarify? Thanks Naidu

- Naidu

IDev  • April 12, 2019 

Which approach is preferred? DCC with volatile instance of singleton OR static inner helper class Noticed the volatile instance case isn't mentioned here



- Heena

JournalDev  • February 14, 2019 

I recommend capitalizing “instance” in the eager initialization example, as it follows convention, but also drives home the point of the instance being final and created at class load.

- Jeremiah

JournalDev  • December 24, 2018 

if is condition not loop :)

- Sajal Gupta

[Show replies](#) 

JournalDev  • September 20, 2018 

why we declare it as static “private static StaticBlockSingleton instance” what would go wrong if we declare it as non static

- Sunny

[Show replies](#) 

Load More Comments



This work is licensed under a Creative Commons Attribution-NonCommercial- ShareAlike 4.0 International License.

Try DigitalOcean for free

Click below to sign up and get **\$200 of credit** to try our products over 60 days!

[Sign up](#)

Popular Topics

[Ubuntu](#)

[Linux Basics](#)

[JavaScript](#)

[Python](#)

[MySQL](#)

[Docker](#)

[Kubernetes](#)

[All tutorials →](#)

[Talk to an expert →](#)



Congratulations on unlocking the whale ambience easter egg! Click the whale button in the bottom left of your screen to toggle some ambient whale noises while you



you to the [Glacier Bay National Park & Preserve](#) and [Merrick079](#) for the sounds of this easter egg.



- Interested in whales, protecting them, and their connection to helping prevent climate change? We recommend checking out the [Whale and Dolphin Conservation](#).

Reset easter egg to be discovered again / Permanently dismiss and hide easter egg

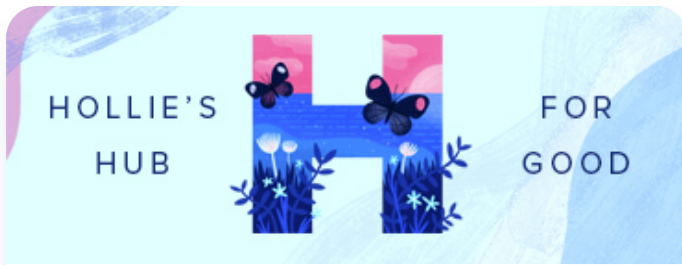


Get our biweekly newsletter

Sign up for Infrastructure as a Newsletter.

[Sign up →](#)

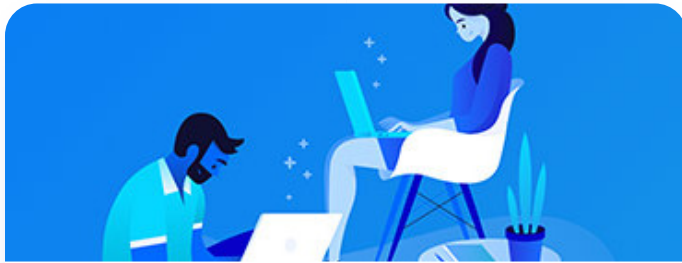




Hollie's Hub for Good

Working on improving health and education, reducing inequality, and spurring economic growth? We'd like to help.

[Learn more →](#)



Become a contributor

You get paid; we donate to tech nonprofits.

[Learn more →](#)



Featured on Community



[Kubernetes Course](#)

[Learn Python 3](#)

[Machine Learning in Python](#)

[Getting started with Go](#)

[Intro to Kubernetes](#)

DigitalOcean Products

[Cloudways](#)

[Virtual Machines](#)

[Managed Databases](#)

[Managed Kubernetes](#)

[Block Storage](#)

[Object Storage](#)

[Marketplace](#)

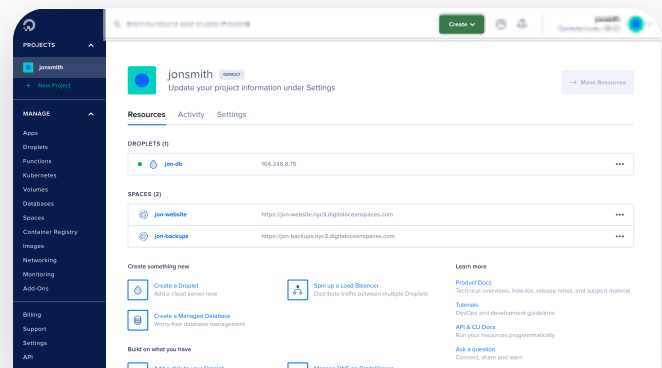
[VPC](#)

[Load Balancers](#)

Welcome to the developer cloud

DigitalOcean makes it simple to launch in the cloud and scale up as you grow – whether you're running one virtual machine or ten thousand.

[Learn more →](#)



Get started for free

Sign up and get \$200 in credit for your first 60 days with DigitalOcean.

Get started



Optional offer applies to new accounts only.



Company	▼
Products	▼
Community	▼
Solutions	▼
Contact	▼

 © 2023 DigitalOcean, LLC. [Sitemap](#).

