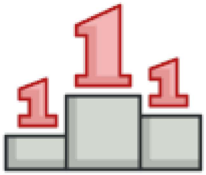
/ Design Patterns / Singleton / Java



Singleton in Java

Singleton is a creational design pattern, which ensures that only one object of its kind exists and provides a single point of access to it for any other code.

Singleton has almost the same pros and cons as global variables. Although they're super-handy, they break the modularity of your code.

You can't just use a class that depends on a Singleton in some other context, without carrying over the Singleton to the other context. Most of the time, this limitation comes up during the creation of unit tests.

Learn more about Singleton →

Navigation

Complexity: ★☆☆

Popularity: ★★☆

Usage examples: A lot of developers consider the Singleton pattern an antipattern. That’s why its usage is on the decline in Java code.

Despite this, there are quite a lot of Singleton examples in Java core libraries:

- `java.lang.Runtime#getRuntime()`
- `java.awt.Desktop#getDesktop()`
- `java.lang.System#getSecurityManager()`

Identification: Singleton can be recognized by a static creation method, which returns the same cached object.

Naïve Singleton (single-threaded)

It’s pretty easy to implement a sloppy Singleton. You just need to hide the constructor and implement a static creation method.

Singleton.java: Singleton

```java
package refactoring_guru.singleton.example.non_thread_safe;

public final class Singleton {
    private static Singleton instance;
    public String value;

    private Singleton(String value) {
        // The following code emulates slow initialization.
        try {
            ex.printStackTrace();
        }
        this.value = value;
    }

    public static Singleton getInstance(String value) {
        if (instance == null) {
            instance = new Singleton(value);
        }
        return instance;
    }
}
```

DemoSingleThread.java: Client code

```
package refactoring_guru.singleton.example.non_thread_safe;

public class DemoSingleThread {
    public static void main(String[] args) {
        System.out.println("If you see the same value, then singleton was reused (yay!)" + "
                "If you see different values, then 2 singletons were created (booo!!)" + "\n
                "RESULT:" + "\n");
        Singleton singleton = Singleton.getInstance("FOO");
        Singleton anotherSingleton = Singleton.getInstance("BAR");
        System.out.println(singleton.value);
        System.out.println(anotherSingleton.value);
    }
}
```

OutputDemoSingleThread.txt: Execution result

```
If you see the same value, then singleton was reused (yay!)
If you see different values, then 2 singletons were created (booo!!)

RESULT:

FOO
FOO
```

Naïve Singleton (multithreaded)

The same class behaves incorrectly in a multithreaded environment. Multiple threads can call the creation method simultaneously and get several instances of Singleton class.

Singleton.java: Singleton

```java
package refactoring_guru.singleton.example.non_thread_safe;

public final class Singleton {
    private static Singleton instance;
    public String value;

    private Singleton(String value) {
        // The following code emulates slow initialization.
        try {
            Thread.sleep(1000);
        } catch (InterruptedException ex) {
            ex.printStackTrace();
        }
        this.value = value;
    }

    public static Singleton getInstance(String value) {
        if (instance == null) {
            instance = new Singleton(value);
        }
        return instance;
    }
}
```

DemoMultiThread.java: Client code

```java
package refactoring_guru.singleton.example.non_thread_safe;

public class DemoMultiThread {
    public static void main(String[] args) {
        System.out.println("If you see the same value, then singleton was reused (yay!)" + "
                "If you see different values, then 2 singletons were created (booo!!)" + "\n
                "RESULT:" + "\n");
        threadFoo.start();
        threadBar.start();
    }

    static class ThreadFoo implements Runnable {
        @Override
        public void run() {
            Singleton singleton = Singleton.getInstance("FOO");
            System.out.println(singleton.value);
        }
    }

    static class ThreadBar implements Runnable {
        @Override
        public void run() {
            Singleton singleton = Singleton.getInstance("BAR");
            System.out.println(singleton.value);
        }
    }
}
```

OutputDemoMultiThread.txt: Execution result

```
If you see the same value, then singleton was reused (yay!)
If you see different values, then 2 singletons were created (booo!!)

RESULT:

FOO
BAR
```

Thread-safe Singleton with lazy loading

To fix the problem, you have to synchronize threads during first creation of the Singleton object.

Singleton.java: Singleton

```java
public final class Singleton {
    // The field must be declared volatile so that double check lock would work
    // correctly.
    private static volatile Singleton instance;

    public String value;

    private Singleton(String value) {
        this.value = value;
    }

    public static Singleton getInstance(String value) {
        // The approach taken here is called double-checked locking (DCL). It
        // exists to prevent race condition between multiple threads that may
        // attempt to get singleton instance at the same time, creating separate
        // instances as a result.
        //
        // It may seem that having the `result` variable here is completely
        // pointless. There is, however, a very important caveat when
        // implementing double-checked locking in Java, which is solved by
        // introducing this local variable.
        //
        // You can read more info DCL issues in Java here:
        // https://refactoring.guru/java-dcl-issue
        Singleton result = instance;
        if (result != null) {
            return result;
        }
        synchronized(Singleton.class) {
            if (instance == null) {
                instance = new Singleton(value);
            }
            return instance;
        }
    }
}
```

DemoMultiThread.java: Client code

```java
package refactoring_guru.singleton.example.thread_safe;

public class DemoMultiThread {
    public static void main(String[] args) {
            "RESULT:" + "\n");
        Thread threadFoo = new Thread(new ThreadFoo());
        Thread threadBar = new Thread(new ThreadBar());
        threadFoo.start();
        threadBar.start();
    }

    static class ThreadFoo implements Runnable {
        @Override
        public void run() {
            Singleton singleton = Singleton.getInstance("FOO");
            System.out.println(singleton.value);
        }
    }

    static class ThreadBar implements Runnable {
        @Override
        public void run() {
            Singleton singleton = Singleton.getInstance("BAR");
            System.out.println(singleton.value);
        }
    }
}
```

OutputDemoMultiThread.txt: Execution result

```
If you see the same value, then singleton was reused (yay!)
If you see different values, then 2 singletons were created (booo!!)

RESULT:

BAR
BAR
```

Want more?

There are even more special flavors of the Singleton pattern in Java. Take a look at this article to find out more:

★ Java Singleton Design Pattern Best Practices with Examples (single-threaded) (multithreaded) Singleton with lazy loading

RETURN

← Prototype in Java

READ NEXT

Adapter in Java →

Singleton in Other Languages

Home Refactoring Design Patterns Premium Content
Forum Contact us

Illustrations by Dmitry Zhart

Terms & Conditions Privacy Policy
Content Usage Policy About us

Ukrainian office:
FOP Olga Skobeleva
Abolmasova 7
Kyiv, Ukraine, 02002
Email:
support@refactoring.guru

Spanish office:
Oleksandr Shvets
Avda Pamplona 64
Pamplona, Spain, 31009
Email:
support@refactoring.guru