Singleton

3

If you've worked with patterns in the past, you're probably familiar with the Singleton Pattern; it's probably the most well-known pattern. Therefore, we will simply start with this pattern in order to find an "easy" introduction.

3.1 The Task of the Singleton Pattern

Imagine that you have a class that defines something that is so unique that there can only be one instance of it. This one instance should be accessible via a global access point. Take the metaclass of a class as an example. For each class that is loaded into the Java Virtual Machine (JVM), an instance of a class is created that stores descriptive meta-information. This object is so unique that it must exist only once in the JVM. You can query it with `String.class`, but also with `(new String()). getClass()`. You can prove that the return value is one and the same object by comparing the references. Enter the following line in a Java shell:

```
System.out.println(String.class ==
                              (new String(). getClass()));
```

On the console, `true` is output, so the references are the same and thus object equality is proven. This is in line with your expectations – it would be bad if there were two meta classes of one class.

The singleton pattern comes into play in another place: Your application's runtime environment is so unique that it must exist only once. Consider the API documentation of the `Runtime` class:

O. Musch, *Design Patterns with Java*, https://doi.org/10.1007/978-3-658-39829-3_3

> Every Java application has a single instance of class Runtime that allows the application to interface with the environment in which the application is running. The current runtime can be obtained from the getRuntime method. An application cannot create its own instance of this class.

You will resort to a singleton yourself if you manage your application's user rights, for example, or perhaps program a cache. A typical application can also be a thread pool. Also, resources like icons or images that you use in your program only need to be loaded once and can be stored in a singleton. Let's look at a few examples of how you can implement the Singleton Pattern.

3.2 A First Version of the Singleton

So how can the singleton pattern be implemented? You should first ensure that there is only one copy of the class. In Java, to prevent users from creating instances of a class, you must first set the constructor to private.

```
public class MySingleton {
    private MySingleton() {
    }
}
```

Now, access to the constructor is only possible from within the class or through other instances of this class. Since there is no (other) instance of the class, however, you must ensure that the one instance is created within the class. So define a method that creates an instance if there is no instance already; the instance of the class is returned to the caller of the method. This method must be public, of course. Keep a reference to this instance in a static variable:

```
public class MySingleton {
    private static MySingleton instance;
        private MySingleton() {
    }

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

That's it! This is a simple way to implement the singleton pattern! The instance of the class is created at the latest possible time, namely when it is needed for the first time. Therefore, this procedure is called *lazy instantiation*, i.e. *delayed loading*.

You can find the code for this in the sample project Singleton_1.

In the application, you then get the reference to the one instance with `MySingleton single = MySingleton.getInstance()` and then call the actual function of the singleton with `single.doSomething()`:

```
public static void main(String[] args) {
    var singleton1 = MySingleton.getInstance();
    singleton1.doSomething();
    var singleton2 = MySingleton.getInstance();
    singleton2.doSomething();
}
```

And here I use a feature added in Java 10 for local(!) variables: The so called "Local-Variable Type Inference".

Instead of.

```
MySingleton singleton1 = MySingleton.getInstance();
```

Since version 10 with the JDK Enhancement Proposal (JEP) 286, Java is able to determine the type for local variables independently and without further specification by the programmer out of the context. This happens at translation time, so it does not compromise Java's type safety. In the example here, it is already obvious to the reader from the context that we want to have an instance of a MySingleton. The additional prefix MySingleton has thus become superfluous. It is sufficient to use a.

```
var singleton1 = MySingleton.getInstance();
```

to recognize the variable singleton1 as MySingleton and generate corresponding code.

NetBeans also suggests the switch to var. declaration in a hint if you stand with the cursor in a declaration line where this is possible. This feature facilitates the readability of the code in many cases, if one thinks of the sometimes very long class names in Java:

```
BufferedInputStream inputStream =
                            new BufferedInputStream(...);
```

then becomes simply.

```
var inputStream = new BufferedInputStream(...);
```

I will use this feature frequently in the sample code below, without going into further detail. It is quite catchy and only allowed if the type of the local variable can really be determined unambiguously. Therefore it should not cause you any problems in the further course.

3.3 Synchronize Access

The class described above runs beautifully as long as you don't use concurrency. What can happen? Assume the following case: You have two threads, both accessing the `getInstance()` method. The first thread gets to line 9 and finds that there is no instance yet. Then the Virtual Machine takes away its compute time and lets Thread 2 access the method. Thread 2 is allowed to work until the end of the method and creates an instance of the class, which is returned. Then Thread 1 gets allocated compute time again. The last thing he remembers is that the reference is zero. It will enter the block and create an instance as well. And now comes exactly what you wanted to avoid – you have two instances of the class.

Whenever you want to prevent code from being executed by two threads at the same time, lock the affected code:

```
public class MySingleton {
    // … abridged
   public static synchronized MySingleton getInstance() {
        if (instance == null)
            instance = new MySingleton();
        return instance;
    }
}
```

This code can be found in the sample project Singleton_2. The method can only be entered by a thread when no other thread is working with it (anymore). This solves the problem. However, synchronization is costly, so this approach is an unnecessary brake. Imagine – every time you want to get the one instance of the class, a lock is set on the method. So there must be an alternative.

3.4 Double Checked Locking

Do not lock the entire method, but only the critical part! First query whether an instance has already been created. If the method is called for the second time, the result of the comparison is `false`. Therefore, a lock is only required on the first call, when the comparison `instance == null` returns a `true`. So inside the if statement, you create a block that

is synchronized. In this block, you query – this time in a thread-safe manner – whether there is an instance of the class. If not, create one. Finally, return the instance. Take a look at the example project Singleton_3.

```
public class MySingleton {
    private static volatile MySingleton instance;

    // … abridged

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

By the way, in NetBeans you get a hint for line 6 ("`synchronized (MySingleton.class)`"). NetBeans detects double checked locking and still suggests the use of a local variable to improve performance. If you're using NetBeans, take a look at that too. It makes the code a bit longer, so I'll leave it out of the example here. This approach to Double Checked Locking is correct in theory. However, in some circumstances you run into a problem. Imagine the following scenario: You have two threads; thread 1 calls the `getInstance()` method. At line 5, it detects that no instance of the class has been created yet, and gets the lock on the following block. In line 7, it checks again – in a thread-safe manner – whether an instance of the singleton class has really not yet been created. If not, it creates an instance with the `new operator` (line 8). Depending on the runtime environment, the following happens during instantiation: First, memory is requested, then the memory is passed to the variable `instance,` which is now nonzero. Only in the next step is the constructor of the class called. And now our problem begins to mature: thread 2 enters the `getInstance()` method before the constructor call by thread 1. It determines that `instance! = null`, gets the instance returned, and works with it. Only later does Thread 1 execute the constructor, which puts the instance into a valid state.

You can prevent this by declaring the variable `instance` as `volatile.`

```
public class MySingleton {
    private static volatile MySingleton instance;
    // … abridged
}
```

The problems around double-checked locking should almost justify a whole dissertation. I would just like to point out here that this solution can lead to runtime errors. There is no guarantee that the volatile keyword is implemented correctly in every virtual machine. For example, a JVM in a version below 1.5 may not implement this approach correctly at all. In such cases, you should strongly consider switching to more recent JVM versions. However, I hope that in 2021, when we are at Java 16, I won't have to convince anyone to migrate at least to Java 8 (2014), better 11 (2018). Java 11 is the most current "Long Term Support" version until Java 17 comes out (scheduled for fall 2021).

3.5 Early Instantiation – Early Loading

To avoid all problems in concurrent systems, you can resort to early loading. As usual, you create a static variable that holds the reference to the single instance. You initialize it as soon as you load it. A static method returns that instance. The code is really very simple. You can find it in the sample project Singleton_4:

```
public class MySingleton {
    private static final MySingleton INSTANCE =
                                    new MySingleton();

    private MySingleton() { }

    public static MySingleton getInstance() {
        return INSTANCE;
    }

    public void doSomething() {
        // Behavior of the object
        // … abridged
    }
}
```

Since no second instance of the class is to be created and cannot be created at all, the following solution (in the Singleton_5 example project) is also conceivable: Make the instance public, and you then additionally save the getInstance method:

```
public class MySingleton {
    public static final MySingleton INSTANCE =
                                    new MySingleton();

    private MySingleton() { }

    public void doSomething() {
```

```
            // Behavior of the object
            // … abridged
        }
    }
```

You avoid all problems resulting from concurrency with this approach. But you have to trust that the virtual machine will first create the instance before allowing other threads to access it – and that is exactly what you can rely on according to the specification.

Nevertheless, there are three things to keep in mind here as well:

1. The instance is created when the class is loaded. Assuming you have an extensive initialization routine, this will be run through in any case, even if you do not need the singleton afterwards.
2. You have no chance to pass arguments to the constructor at runtime.
3. The singleton pattern ensures that the class is instantiated only once per class loader. In the case of a web server, it may be that the class is loaded in more than one class loader within a virtual machine; then you have more than one instance of the singleton class again. Without going into detail, such situations are nowadays dealt with using Dependency Injection.

3.6 Singleton – The UML Diagram

In Fig. 3.1 you can see the (quite simple) UML diagram for the Singleton from the sample project Singleton_4, i.e. with the `getInstance` method.

3.7 Antipattern

You now know the Singleton Pattern and how to implement it. You know the advantages and disadvantages of the different implementations. But there are fundamental objections against the Singleton Pattern. I mean objections to the pattern as a definition and not to the realization shown. Search for the term “evil singleton” in your trusted search engine. I get over 2,000,000 hits.

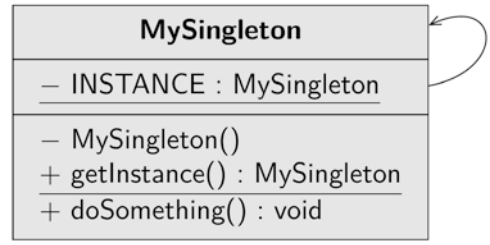

Fig. 3.1 UML diagram of the Singleton Pattern (example project Singleton_4)

3.7.1 Criticism of the Singleton

I would like to present a few of the criticisms.

- Each class should focus on exactly one task: Single Responsibility Principle. Singleton violates this principle in that the class must take care of both the business logic and its own object creation.
- Just as with global variables, the dependency is not immediately obvious. Whether a class makes use of a singleton class cannot be seen from the interface, but only from the code.
- The coupling is increased.
- Too extensive use of singleton tempts the programmer to program procedurally.
- If the instance has its own attributes, these are available in the complete application. It is questionable whether there is data that is actually needed "everywhere": in the view, in the controller, in the model and in the persistence layer.
- You restrict yourself when using singletons: You cannot create subclasses of singleton classes. If 1 day you do need a second instance, you have to rewrite the code again.
- Singletons create something like a global state. This makes the test procedure more difficult for you.

Considering these points, many programmers consider the singleton pattern to be an antipattern.

3.7.2 Is Singleton an Antipattern?

Antipatterns describe unsuitable procedures, negative examples, so to speak. Each antipattern can be described just as formally as a design pattern. They identify why it is not good to proceed in a certain way. Antipatterns help to learn from the mistakes of others.

The arguments against the use of the Singleton are certainly understandable. However, there are tasks that would be difficult to solve without a Singleton. I do not want to agree to a blanket evaluation – you will have to decide on a case-by-case basis and consider for each Singleton class whether it is really necessary.

3.8 Summary

Go through the chapter again in key words:

- A singleton is used when you have a class of which there may be only one instance.
- There must be a global access point for this one instance.
- Lazy instantiation creates the instance when it is needed.

- Simultaneous access – concurrency – to the singleton leads to runtime errors, especially during generation, and must be protected accordingly.
- Early instantiation avoids the problems of concurrency, but has other disadvantages.
- Antipatterns describe negative examples, i.e. unsuitable procedures.

3.9 Description of Purpose

The Gang of Four describes the purpose of the pattern “Singleton” as follows:

> Ensure that a class has exactly one copy, and provide a global access point to it.