March 7, 2017 | Vojtech Ruzicka

Singleton Pattern Pitfalls

August 31, 2016 by

Singleton, one of the most popular design patterns has a lot of serious drawbacks and some even consider it an anti-pattern. What are the pitfalls of the famous pattern?

What is singleton?

Singleton, one of the original GoF patterns, is probably one of the most famous design patterns out there. Its intent is simple:

> Ensure a class only has one instance and provide a global access to it.

The reason of its popularity is probably its simplicity. Just one class, nothing complicated. It is probably the easiest of the original patterns to learn. If a person just discovered the whole new world of design patterns, you can be sure they know singleton. And they are eager to use it. The problem is – if you have only a hammer in your toolbox, everything suddenly looks like a nail. Because of that, singleton tends to be overused despite its many drawbacks, which are not even mentioned in the original book.

Implementation

The basic implementation of singleton in java is very simple:

1. Make the constructor private, so there is no way to instantiate the class form the outside.
2. Store the instance in a private static field.
3. Provide accessor method to the instance field.

```
public class Singleton {

private static Singleton instance = new Singleton();

private Singleton() {
}

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

Single Responsibility Principle

According to the Single Responsibility Principle, each class should have just one responsibility – just one reason to change. Clearly, Singleton pattern is violating this principle. Instead of clear separation of concerns, the class has two distinct responsibilities – the first is making sure that only one instance can be created. The second is core functionality of the class – accessing DB, managing some unique resource or whatever is it supposed to do.

The class having too many responsibilities and too many reasons to change is much harder to maintain as changes in one responsibility affect other responsibilities as well. Coupling of the core class functionality and management of number of its instances greatly reduces reusability of the class. The problem is that determining whether the class is singleton is usually not dependent on the class itself, but rather on the system and context in which it is used. That means – the same class, depending on system in which it is used, could be either singleton or have multiple instances. Ideally, the responsibility for managing the number of instances should be extracted from the class itself and managed externally. One example of this could be Dependency Injection container managing lifecycle of the components (and choosing which specific dependencies to instantiate). This way, the same class can be used in multiple contexts with different lifecycle management (single instance, unlimited instances, pool of instances, …).

Testing

On of the main disadvantages of singletons is that they make unit testing very hard. They introduce global state to the application. The problem is that you cannot completely isolate classes dependent on singletons. When you are trying to test such a class, you inevitably test the Singleton as well. When unit testing, you want the class to be as loosely coupled with other classes as possible and all the dependencies of the class should be ideally provided externally (either by constructor or setters), so they can be easily mocked. Unfortunately that is not possible with singletons as they introduce tight coupling and the class retrieves the instance on its own. But it gets even worse. The global state of stateful singletons is preserved between test cases. That has several serious implications:

- Order of the tests now matters
- Tests can have unwanted side effects caused by singleton
- You cannot run multiple tests in parallel
- Multiple invocations of the same test case can result in different results

I recommend watching the The Clean Code Talks – "Global State and Singletons" for further information about the topic.

Dependency hiding

Usually when a class requires external collaborators, it is immediately obvious from its constructor's or methods's signature. It is clear which dependencies class has and those can be easily provided. What's more – you can easily provide mocks instead when testing. If a class calls a singleton, it is not obvious from constructor or method. It is even worse when that singleton requires some kind of initialization (like by calling some kind of init(…) method with initial state). In that case there is no way for someone who is going to use that class to know, that they should initialize the singleton before. It gets even messier when there is a number of dependent singletons, which need to be initialized in a specific order.

Classloaders

Singleton is supposed to have no more than one instance. However the classical implementation does not ensure that there is just one instance per JVM, it only ensures that there is one instance per classloader! If you have a simple client app with just one classloader, you don't need to worry about that. The problem is when using multiple classloaders or deploying the app to an application server. App servers usually use a hierarchy of classloaders. Each deployed app then usually has its own isolated classloader. Different apps deployed on the app server or multiple instances of the same app then do not share one singleton, but each has its own.

Deserialization

Singleton prohibits creation of new instances by hiding its constructor(s), so nobody can call the constructor directly. There are, however additional means of creating new instances of a class. On of them is serialization/deserialization. If the singleton is supposed to be serializable, you need to make sure only one instance is created during deserialization.

Yo can achieve this by adding readResolve() method, which returns the current instance of the singleton.

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

You prevent multiple deserialization attempts of singleton, however it is still not completely safe. The problem is that the readResolve() method is called AFTER the object has been deseriaized and can alter object being returned or return different object altogether (like in our case). By returning existing instance from getInstance(), the freshly deserialized instance is not referenced by any other objects and thus is immediately eligible for garbage collection. It is therefore possible for an attacker to obtain that reference and keep the copy, preventing it from being garbage collected as described in Effective Java – Item 57: Provide a readResolve method when necessary. To prevent this, you need to declare all the fields of the singleton with object references as transient, which leaves you with ability to persist only primitives. Josh Bloch suggests using single-item enums as singleton to prevent this, but in my opinion it is abuse of enum for a purpose it is not intended to.

Additional cases, where multiple instances of singleton can be created are discussed in this post.

Concurrency

A popular variant of singleton uses lazily initialised instance instead of creating it right away. This approach is appropriate when creating the instance is very expensive operation and it is uncertain whether the instance would be used at all.

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

return instance;
}
```

If the implementation above is used, it is possible that multiple instances are created when multiple threads are accessing the singleton's getInstance() method.

1. The first thread enters the getInstance() method, while instance is still not initialised
2. The second thread takes over before the first thread could create the singleton instance and creates it
3. The first thread continues and creates its own instance

Common fix to that is to declare getInstance() method as synchronised, which prevents this issue. The problem is that then lazy initialisation saves unnecessary instance creation at startup, but instead every access to the instance is more expensive due to the synchronisation cost. It can be a problem if the instance is frequently accessed. But the only case when the method needs to be synchronised is when the getInstance() is called for the first time.

There are two main approaches how to solve that. The first one is to synchronise just the block, where the instance is initialised. Keep in mind, that you cannot use this approach pre-java 1.5 as it used different memory management model. Also be sure to declare instance field as volatile.

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

return instance;
}
```

The second approach is to use the Lazy Initialization Holder class pattern, as described in Java Concurrency in Practice – 16.2.3. Safe Initialization Idioms. This approach eliminates the need of synchronisation altogether. It is based on JVM's behavior, which defers initialization of the nested classes until they are actually needed.

```
public class Singleton {

private Singleton() {
}

public static Singleton getInstance() {
return SingletonHolder.instance;
}

private static class SingletonHolder {
private static final Singleton instance = new Singleton();
}
}
```

Reflection attack

The maximum of one instance is enforced by making the constructor private. However there is a way to bypass this limitation using reflection. At runtime, access modifier can be changed from private to public and suddenly constructor can be called from outside, resulting in multiple instances.

```
Class clazz = Singleton.class;
Constructor constructor = clazz.getDeclaredConstructor();
constructor.setAccessible(true);
```

You do not need to worry about this if you use your singleton just in your application. However, when you distribute a module, which will be used by third parties, this can be an issue. Especially if having multiple instances can result in security risks or other unexpected behaviour of your module.

You can read more about preventing this in Reflection Proofing the Java Singleton Pattern when using Lazy Loading.

Conclusion

Although singleton design pattern is very famous and popular, it has many serious drawbacks. Fortunately, with time those are becoming more and more widely recognised and original pattern is now often considered more of an anti-pattern. Having a single instance is still very useful, however, there are better ways to achieve that. The best solution is to use a container such as Spring to manage lifecycle independently from the class itself. This way, the separation of concerns is maintained, class is reusable in different contexts and well testable.

What do you think about using the original Singleton? Is it still a viable option, should it be used with caution or avoided altogether? You can share your opinions in the comments bellow.

Filed Under: Design Patterns
