**DISADVANTAGES OF NORMAL SINGLETON DESIGN PATTERN**

**Concurrency**

A popular variant of singleton uses lazily initialized instance instead of creating it right away.

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

There are two main approaches how to solve that. The first one is to synchronize just the block, where the instance is initialized. 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. This approach eliminates the need of synchronization 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.