Thread-safe programming in Java Best practice in multi-threading programming

Show Submission Password

Feedback to Cloud Engineers

(https://docs.google.com/forms/d/e/1FAIpQLSccxFOSrRMK0XDF-HX5HWfvxOf3pzVn8eRA-LIopAgUdwirTQ/viewform)

Best practice and common pitfalls in Java

Summary

Best practice and common pitfalls in Java

Synchronization

When you want to coordinate multiple threads in a Java program to ensure thread-safety,Java allows you to achieve this by using a **synchronized method** or **synchronized blocks**.

In Java, every object has a unique internal lock. When a method is declared as "synchronized", or a code snippet is enclosed by "synchronized(this)" block, that method or code snippet will be protected by an internal lock. When any thread wants to enter the method or the code snippet, it must first try to acquire the internal lock. This ensures that only one method can be executed at any given point in time. Other methods can invoke the method or enter the code snippet, however they have to wait until the running thread releases the lock by exiting from the protected area or call wait() on the lock.

You can also use the lock of other objects to protect the synchronized code block. The lock associated the object passed into synchronized block will protect the code block.

For example, in Figure 1, when you synchronize on “lock” object, you are using the internal lock of that object to protect the enclosed code snippet.

```java
// Good example! It's thread-safe
private Object lock = new Object();
public void synchronization() {
    synchronized (lock) {
        //Do something
    }
}
```

Figure 1: synchronized block

Thread-safety

Before we proceed, let's give a formal definition of thread-safety. Thread-safe code must ensure that **when multiple threads are accessing a shared object, no matter how these threads are scheduled or in what order they are executing, this object will always behave correctly without any external synchronization, and every thread will be able to see any changes happened on this object immediately.**

To summarize, in order to ensure your code is thread-safe, your code must ensure *Atomicity* and *Visibility*. Let's discuss each item:

1. **Atomicity.** Any operations that change the state of the shared object must be atomic, which means that the operation should either be executed completely or not executed at all. No intermediate state should be exposed. For example, when you call notSafeUpdate() method, you add 1 to the internal counter and change its internal state.

```java
// Bad example: Don't do this in your project
public class NonAtomicCounter {
    int count;

    public void notSafeUpdate() {
        count++;
    }
}
```

Figure 2: NonAtomicCounter

Is this counter thread-safe? **No, the reason is that "++" is not atomic.** Recall the bank example in the In fact, it is composed by three java byte code instructions: read [count] from memory, add 1 to it, and then write it back to memory. If thread B accesses this variable while thread A has

added 1 to it but did not write it back, then inconsistency has occurred.

Two modifications will be provided here. They are both correctly synchronized using the internal lock of any AtomicCounter instance.

```
// Good example!
public class AtomicCounter {
    int count;
    // These two methods are equivalent.
    public synchronized void safeUpdate1() {
        count++;
    }

    public void safeUpdate2() {
        synchronized (this) {
            count++;
        }
    }
}
```

Figure 3: AtomicCounter

2. **Visibility.** The changes made by one thread should be immediately visible by any other thread. However, modern computers are equipped with multi-level caches, which is the root cause of invisibility in most situations. Incomplete or even corrupted data may be observed if invisibility occurs.

```java
// Bad example: Don't do this in your project!
public class InvisibleCounter {
    int count;

    public int getCount() {
        return count;
    }

    public synchronized void safeUpdate() {
        count++;
    }
}
```

Figure 4: InvisibleCounter

Although the write operation is correctly synchronized, this object is still thread-unsafe, since the read operation is not synchronized. Other threads can still access this variable from cache when one thread is modifying it. The change made by the writer thread is invisible to readers. *Hence, you need to always synchronize both read and write operations when you want to ensure thread-safety.*

A correct modification is provided here: (**Unless you are really an expert in Java programming, we strongly recommend not using "volatile" in this project**, the usage of volatile is subtle and far beyond the scope of this project)

```java
// Good example: synchronize both read and write
public class VisibleCounter {
    int count;

    public synchronized int safeGetCount() {
        return count;
    }

    public synchronized void safeUpdate() {
        count++;
    }
}
```

Figure 5: VisibleCounter

Below is another example code. Can you spot the thread-safety issue? (the JVM ensures that reading or writing a primitive type variable is atomic except variables of type long or double)

```
// Bad example: Don't do this in your project!
public class InvisibleRunner{
    private boolean stopRunning = false;

    // Your Baymax will say hello forever when you run this program in server mode
    public void sayHello() {
        while (!this.stopRunning) {
            System.out.println("Hi, I am Baymax, your personal healthcare companion");
        }
    }

    public void invisibleStop() { this.stopRunning = true; }
}
```

Figure 6: InvisibleRunner

Thread-safety in Java Libraries

When programming in Java, you may want to know if one class is thread-safe or not. If it is thread-safe, then you do not need to synchronize yourself. If it is not thread-safe, you may need to explicitly perform synchronization externally, e.g. Use a lock to protect every method accessing the object. There are five levels thread-safety in Java library:

1. **Immutable.** Immutable objects cannot be changed or modified, so we can safely share it among threads and do not need to do any synchronization.

   Example: String, Integer, Long (they are different from int or long), BigInteger...

2. **Unconditionally thread-safe.** These objects are mutable, but have implemented sufficient "internal synchronization", which means that locking has already been designed and implemented by Java standard library developers, and we do not need to synchronize by ourselves manually. Thus, we can safely use them.

   Example: **Random, ConcurrentHashMap, ConcurrentHashSet, BlockingQueue PriorityBlockingQueue, AtomicInteger….** (Vector and HashTable are thread-safe, but they have been depreciated, so don't use them in your project)

```
// Good example! It's thread-safe
ConcurrentHashSet<String> set = new ConcurrentHashSet<>();
public void add(String str) {
    set.add(str);
}
```

Figure 7: How to use thread-safe object

3. **Conditionally thread-safe.** These objects are mutable, but have implemented sufficient internal synchronization on most of the methods. However, some methods still need to explicitly perform external synchronization. There are few examples in Java Library. In our project, you will not encounter this kind of object. **Read the documentation before using them if needed.**

   Example: Sets wrapped by Collections.synchronized(). Their iterators are not thread-safe.

4. **Thread-unsafe.** These objects are mutable and implemented with no internal synchronization. In order to use them safely, please explicitly synchronize ANY method call to these objects.

   Example: Most of the Java Collections, such as HashMap, HashSet, ArrayList, LinkedList. StringBuilder...

```
// Good example! It's thread-safe
HashSet<String> set = new HashSet<>();
public synchronized void add(String str) {
    set.add(str);
}
```

Figure 8: Explicit synchronization

5. **Thread-hostile.** These objects are not thread-safe even if you have already used external synchronization on ANY method call. Fortunately, the Java library has few examples, in other words, you should never encounter this situation in your project, or even your future career :)

   Example: System.runFinalizerOnExit(), deprecated since Java 1.6

Common Mistakes made by previous students

The following mistakes are made by previous students in the consistency project. Make sure you are not one of them in this semester :)

1. **No external lock on method call series.**Although containsKey() and put() method calls are respectively thread-safe, it's not thread-safe to call them in series without further synchronization.

   When thread A finds that key is not contained, it will enter the condition block and may be stopped by another thread B. Thread B can put a value inside the map, and when A is switched back, it will unconditionally override the value put by B. Boom! This causes inconsistency to occur.

```java
// Bad example: Don't do this in your project!
ConcurrentHashMap<String, Integer> map = new ConcurrentHashMap<>();
public void add(String key, Integer value) {
    if (!map.containsKey(key)) {
        map.put(key, value);
    }
}
```

Figure 9: Pitfall 1

Two modification will be provided here. Please read the document for the usage of putIfAbsent() (https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentHashMap.html#putIfAbsent(K,%20V))

```java
// Good example! It's thread-safe
ConcurrentHashMap<String, Integer> map = new ConcurrentHashMap<>();

public void add1(String key, Integer value) {
    synchronized (this) {
        if (!map.containsKey(key)) {
            map.put(key, value);
        }
    }
}

public void add2(String key, Integer value) {
    if (!map.containsKey(key)) {
        map.putIfAbsent(key, value);
    }
}
```

Figure 10: Thread-safe add()

2. **Only synchronize on write operations but not read operations** (an example is given in the "Visibility" part above)
3. **Be careful when using notify()**

Unless you know exactly what you are doing, We recommend that you use notifyAll() instead of notify(). notify() makes no guarantee on which thread it will wake. Most of the time, incorrect usage of notify() will lead to a deadlock, in which every thread is blocked and waiting to be notified, but no one can notify others. **Also, you should always call notify() or notifyall() inside a synchronized block or a synchronized method.**

4. **Wrong pattern when using wait().** Please always use this pattern when you use wait(). A thread can also wake up without being notified, interrupted, or timing out, a so-called spurious wakeup. While this will rarely occur in practice, applications must guard against it by testing for the condition that should have caused the thread to be awakened, and continuing to wait if the condition is not satisfied. In other words, waits should always occur in loops. **Never call wait() outside a loop.**

```
private Object lock = new Object();
public void conditionalRun() {
    synchronized (lock) {
        while (<condition does not hold>) {
            lock.wait();
        }
    }

    // Do something
}
```

Figure 11: Correct pattern to use wait()

See here for more example on usage of wait() (https://www.tutorialspoint.com/java/lang/object_wait.htm) and notifyAll() (https://www.tutorialspoint.com/java/lang/object_notifyall.htm).

5. **Synchronize on wrong lock.**

```
// Bad example: Don't do this in your project!
public void uselessSynchronization() {
    synchronized (new Object()) {
        //Do something
    }
}
```

Figure 12: Wrong synchronization

This synchronization is totally useless. In order to realize mutual exclusive access to any shared object, **the read and write operations should be protected under the same lock**. In the bad example above, every method invoker will synchronize on different objects, which, unfortunately, has no synchronization effect at all.

```
// Good example! It's thread-safe
private Object lock = new Object();
public void synchronization() {
    synchronized (lock) {
        //Do something
    }
}
```

Figure 13: Correct synchronization

A more common mistake is synchronizing on threads' own lock.Remember our previous discussion on synchronization, when you passed "this" inside the synchronized block, the block will be protected by the internal lock of that thread instance. Since each instance has its own internal lock, the code block is not protected by a global lock and every thread can access it at any time. Thus, this kind of synchronization will fail to work.

```
// Bad example: Don't do this in your project!
public void uselessSync() {
    new Thread(new Runnable() {
        @Override
        public void run() {
            synchronized (this) {
                //Do something
            }
        }
    }).start();
}
```

Figure 14: Useless synchronization

Alway synchronize on the same lock before accessing the shared object.

```java
// Good example! It's thread-safe
private Object lock = new Object();
public void sync() {
    new Thread(new Runnable() {
        @Override
        public void run() {
            synchronized (lock) {
                //Do something
            }
        }
    }).start();
}
```

Figure 15: Correct synchronization

6. **Over-synchronization.** Over-synchronization will not lead to concurrency failure, but is considered bad practice. There are two reasons why you should not over synchronize:

    1. You will pay a performance penalty.
    2. Sometimes over-synchronization can lead to a deadlock.

    Typical over-synchronization is when you synchronize on a stack variable. Stack variables are enclosed inside the current thread, so they cannot be shared among a thread. Thus, we should not synchronize on use of stack variables.

```java
// Bad example: Don't do this in your project!
private Object lock = new Object();
public void sync() {
    HashMap<String, Integer> map = new HashMap<>();
    synchronized (lock) {
        map.put("Over-Sync", 0);
    }
}
```

Figure 16: Example of over-synchronization

7. **If you don't know the thread-safety of any object or method call, please check the Java documentation. Don't make an assumption, it will lead to bad results.**

Summary

That’s all! After reading this primer, we're sure that you will enjoy your time playing with multithreading and the consistency project. If you want to know more about multithreading, we recommend two great books: *Effective Java* and *Java Concurrency in Practice*.

May the force be with you.