(/)

Common Concurrency Pitfalls in Java

Last updated: December 22, 2021

Written by: Catalin Burcea (https://www.baeldung.com/author/catalin-burcea)

Java Concurrency (https://www.baeldung.com/category/java/java-concurrency)

1. Introduction

In this tutorial, we're going to see some of the most common concurrency problems (/cs/aba-concurrency) in Java. We'll also learn how to avoid them and their main causes.

2. Using Thread-Safe Objects

2.1. Sharing Objects

Threads communicate primarily by sharing access (/java-thread-safety) to the same objects. So, reading from an object while it changes can give unexpected results. Also, concurrently changing an object can leave it in a corrupted or inconsistent state.

The main way we can avoid such concurrency issues and build reliable code is to work with immutable objects (/java-immutable-object). This is because their state cannot be modified by the interference of multiple threads.

However, we can't always work with immutable objects. In these cases, we have to find ways to make our mutable objects thread-safe.

2.2. Making Collections Thread-Safe

Like any other object, collections maintain state internally. This could be altered by multiple threads changing the collection concurrently. So, **one way we can safely work with collections in a multithreaded environment is to synchronize them (/java-synchronized-collections)**:

```
Map<String, String> map = Collections.synchronizedMap(new HashMap<>());
List<Integer> list = Collections.synchronizedList(new ArrayList<>());
```

In general, synchronization helps us to achieve mutual exclusion. More specifically, **these collections can be accessed by only one thread at a time.** Thus, we can avoid leaving collections in an inconsistent state.

2.3. Specialist Multithreaded Collections

Now let's consider a scenario where we need more reads than writes. **By using a synchronized collection, our application can suffer major performance consequences.** If two threads want to read the collection at the same time, one has to wait until the other finishes.

For this reason, Java provides concurrent collections such as *CopyOnWriteArrayList* (/java-copy-on-write-arraylist) and *ConcurrentHashMap* (/java-concurrent-map) that can be accessed simultaneously by multiple threads:

```
CopyOnWriteArrayList<String> list = new CopyOnWriteArrayList<>();
Map<String, String> map = new ConcurrentHashMap<>();
```

The *CopyOnWriteArrayList* achieves thread-safety by creating a separate copy of the underlying array for mutative operations like add or remove. Although it has a poorer performance for write operations than a *Collections.synchronizedList,* it provides us with better performance when we need significantly more reads than writes.

ConcurrentHashMap is fundamentally thread-safe and is more performant than the *Collections.synchronizedMap* wrapper around a non-thread-safe *Map*. It's actually a thread-safe map of thread-safe maps, allowing different activities to happen simultaneously in its child maps.

2.4. Working with Non-Thread-Safe Types

We often use built-in objects like *SimpleDateFormat* (/java-simple-date-format) to parse and format date objects. The *SimpleDateFormat* class mutates its internal state while doing its operations.

We need to be very careful with them because they are not thread-safe. **Their state can become inconsistent in a multithreaded application due to things like race conditions.**

So, how can we use the *SimpleDateFormat* safely? We have several options:

- Create a new instance of *SimpleDateFormat* every time it's used
- Restrict the number of objects created by using a *ThreadLocal<SimpleDateFormat>* object. It guarantees that each thread will have its own instance of *SimpleDateFormat*
- Synchronize concurrent access by multiple threads with the *synchronized* keyword or a lock

SimpleDateFormat is just one example of this. We can use these techniques with any non-thread-safe type.

3. Race Conditions

A race condition (/cs/race-conditions) occurs when two or more threads access shared data and they try to change it at the same time. Thus, race conditions can cause runtime errors or unexpected outcomes.

3.1. Race Condition Example

Let's consider the following code:

```
class Counter {
    private int counter = 0;

    public void increment() {
        counter++;
    }

    public int getValue() {
        return counter;
    }
}
```

The *Counter* class is designed so that each invocation of the increment method will add 1 to the *counter*. However, if a *Counter* object is referenced from multiple threads, the interference between threads may prevent this from happening as expected.

We can decompose the *counter++* statement into 3 steps:

- Retrieve the current value of *counter*
- Increment the retrieved value by 1
- Store the incremented value back in *counter*

Now, let's suppose two threads, *thread1* and *thread2*, invoke the increment method at the same time. Their interleaved actions might follow this sequence:

- *thread1* reads the current value of *counter*; 0
- *thread2* reads the current value of *counter*; 0
- *thread1* increments the retrieved value; the result is 1
- *thread2* increments the retrieved value; the result is 1
- *thread1* stores the result in *counter*; the result is now 1
- *thread2* stores the result in *counter*; the result is now 1

We expected the value of the *counter* to be 2, but it was 1.

3.2. A Synchronized-Based Solution

We can fix the inconsistency by synchronizing the critical code:

```
class SynchronizedCounter {
    private int counter = 0;

    public synchronized void increment() {
        counter++;
    }

    public synchronized int getValue() {
        return counter;
    }
}
```

Only one thread is allowed to use the *synchronized* methods of an object at any one time, so this forces consistency in the reading and writing of the *counter*.

3.3. A Built-In Solution

We can replace the above code with a built-in *AtomicInteger* object. This class offers, among others, atomic methods for incrementing an integer and is a better solution than writing our own code. Therefore, we can call its methods directly without the need for synchronization:

```
AtomicInteger atomicInteger = new AtomicInteger(3);
atomicInteger.incrementAndGet();
```

In this case, the SDK solves the problem for us. Otherwise, we could've also written our own code, encapsulating the critical sections in a custom thread-safe class. This approach helps us to minimize the complexity and to maximize the reusability of our code.

4. Race Conditions Around Collections

4.1. The Problem

Another pitfall we can fall into is to think that synchronized collections offer us more protection than they actually do.

Let's examine the code below:

```
List<String> list = Collections.synchronizedList(new ArrayList<>());
if(!list.contains("foo")) {
    list.add("foo");
}
```

Every operation of our list is synchronized, but any combinations of multiple method invocations are not synchronized. More specifically, between the two operations, another thread can modify our collection leading to undesired results.

For example, two threads could enter the *if* block at the same time and then update the list, each thread adding the *foo* value to the list.

4.2. A Solution for Lists

We can protect the code from being accessed by more than one thread at a time using synchronization:

```
synchronized (list) {
    if (!list.contains("foo")) {
        list.add("foo");
    }
}
```

Rather than adding the *synchronized* keyword to the functions, we've created a critical section concerning *list,* which only allows one thread at a time to perform this operation.

We should note that we can use *synchronized(list)* on other operations on our list object, to provide a **guarantee that only one thread at a time can perform any of our operations** on this object.

4.3. A Built-In Solution for *ConcurrentHashMap*

Now, let's consider using a map for the same reason, namely adding an entry only if it's not present.

The *ConcurrentHashMap* offers a better solution for this type of problem. We can use its atomic *putIfAbsent* method:

```
Map<String, String> map = new ConcurrentHashMap<>();
map.putIfAbsent("foo", "bar");
```

Or, if we want to compute the value, its atomic *computeIfAbsent* method*:*

```
map.computeIfAbsent("foo", key -> key + "bar");
```

We should note that these methods are part of the interface to *Map* where they offer a convenient way to avoid writing conditional logic around insertion. They really help us out when trying to make multi-threaded calls atomic.

5. Memory Consistency Issues

Memory consistency issues occur when multiple threads have inconsistent views of what should be the same data.

In addition to the main memory, most modern computer architectures are using a hierarchy of caches (L1, L2, and L3 caches) to improve the overall performance. **Thus, any thread may cache variables because it provides faster access compared to the main memory.**

5.1. The Problem

Let's recall our *Counter* example:

```
class Counter {
    private int counter = 0;

    public void increment() {
        counter++;
    }

    public int getValue() {
        return counter;
    }
}
```

Let's consider the scenario where *thread1* increments the *counter* and then *thread2* reads its value. The following sequence of events might happen:

- *thread1* reads the counter value from its own cache; counter is 0
- t*hread1* increments the counter and writes it back to its own cache; counter is 1
- *thread2* reads the counter value from its own cache; counter is 0

Of course, the expected sequence of events could happen too and the *thread2* will read the correct value (1), but **there is no guarantee that changes made by one thread will be visible to other threads every time.**

5.2. The Solution

In order to avoid memory consistency errors, **we need to establish a happens-before relationship**. This relationship is simply a guarantee that memory updates by one specific statement are visible to another specific statement.

There are several strategies that create happens-before relationships. One of them is synchronization, which we've already looked at.

Synchronization ensures both mutual exclusion and memory consistency. However, this comes with a performance cost.

We can also avoid memory consistency problems by using the *volatile* keyword. Simply put, **every change to a volatile variable is always visible to other threads.**

Let's rewrite our *Counter* example using *volatile*:

```
class SyncronizedCounter {
    private volatile int counter = 0;

    public synchronized void increment() {
        counter++;
    }

    public int getValue() {
        return counter;
    }
}
```

We should note that **we still need to synchronize the increment operation because *volatile* doesn't ensure us mutual exclusion.** Using simple atomic variable access is more efficient than accessing these variables through synchronized code.

5.3. Non-Atomic *long* and *double* Values

So, if we read a variable without proper synchronization, we may see a stale value. **For *long* and *double* values, quite surprisingly, it's even possible to see completely random values in addition to stale ones.**

According to JLS-17 (https://docs.oracle.com/javase/specs/jls/se14/html/jls-17.html#jls-17.7), JVM may treat 64-bit operations as two separate 32-bit operations. Therefore, when reading a *long* or *double* value, it's possible to read an updated 32-bit along with a stale 32-bit. Consequently, we may observe random-looking *long* or *double* values in concurrent contexts.

On the other hand, writes and reads of volatile *long* and *double* values are always atomic.

6. Misusing Synchronize

The synchronization mechanism (/java-thread-safety) is a powerful tool to achieve thread-safety. It relies on the use of intrinsic and extrinsic locks. Let's also remember the fact that every object has a different lock and only one thread can acquire a lock at a time.

However, if we don't pay attention and carefully choose the right locks for our critical code, unexpected behavior can occur.

6.1. Synchronizing on *this* Reference

The method-level synchronization comes as a solution to many concurrency issues. However, it can also lead to other concurrency issues if it's overused. This synchronization approach relies on the *this* reference as a lock, which is also called an intrinsic lock.

We can see in the following examples how a method-level synchronization can be translated into a block-level synchronization with the *this* reference as a lock.

These methods are equivalent:

```
public synchronized void foo() {
    //...
}
```

```
public void foo() {
    synchronized(this) {
      //...
    }
}
```

When such a method is called by a thread, other threads cannot concurrently access the object. This can reduce concurrency performance as everything ends up running single-threaded. This approach is especially bad when an object is read more often than it is updated.

Moreover, a client of our code might also acquire the *this* lock. In the worst-case scenario, this operation can lead to a deadlock.

6.2. Deadlock

Deadlock (/java-dining-philoshophers) describes a situation where two or more threads block each other, each waiting to acquire a resource held by some other thread.

Let's consider the example:

```java
public class DeadlockExample {

    public static Object lock1 = new Object();
    public static Object lock2 = new Object();

    public static void main(String args[]) {
        Thread threadA = new Thread(() -> {
            synchronized (lock1) {
                System.out.println("ThreadA: Holding lock 1...");
                sleep();
                System.out.println("ThreadA: Waiting for lock 2...");

                synchronized (lock2) {
                    System.out.println("ThreadA: Holding lock 1 & 2...");
                }
            }
        });
        Thread threadB = new Thread(() -> {
            synchronized (lock2) {
                System.out.println("ThreadB: Holding lock 2...");
                sleep();
                System.out.println("ThreadB: Waiting for lock 1...");

                synchronized (lock1) {
                    System.out.println("ThreadB: Holding lock 1 & 2...");
                }
            }
        });
        threadA.start();
        threadB.start();
    }
}
```

In the above code we can clearly see that first *threadA* acquires *lock1* and *threadB* acquires *lock2*. Then, *threadA* tries to get the *lock2* which is already acquired by *threadB* and *threadB* tries to get the *lock1* which is already acquired by *threadA*. So, neither of them will proceed meaning they are in a deadlock.

We can easily fix this issue by changing the order of locks in one of the threads.

We should note that this is just one example, and there are many others that can lead to a deadlock.

7. Conclusion

In this article, we explored several examples of concurrency issues that we're likely to encounter in our multithreaded applications.

First, we learned that we should opt for objects or operations that are either immutable or thread-safe.

Then, we saw several examples of race conditions and how we can avoid them using the synchronization mechanism. Furthermore, we learned about memory-related race conditions and how to avoid them.

Although the synchronization mechanism helps us to avoid many concurrency issues, we can easily misuse it and create other issues. For this reason, we examined several problems we might face when this mechanism is badly used.

As usual, all the examples used in this article are available over on GitHub. (https://github.com/eugenp/tutorials/tree/master/core-java-modules/core-java-concurrency-advanced-3)