Exercises week 2

Last update 2020-08-29

Goals

The goals of this weeks exercises are to enable you:

- to make sure you initialize your objects safely from a concurrency perspective.
- to design immutable objects for concurrency
- to be able to explain the problem solved by *volatile*, and explain how *volatile* solves it

Exercise 2.1 *Green*

Consider the situation where we need a simple class for storing and retrieving person objects from a database.

The Person class should have id, name, zip and address. It should be possible to change zip and address together, it is not necessary to be able to change name. The id must not be able to be changed. It should be possible to get the values of all fields.

In addition, the Person class should be able to give each person a unique id (a long). Each new instance of Person gets an id one higher than the previous. It should be possible to set the id sequence to generate id from a specific value.

1. Write the class so that no race conditions can arise, neither with updating a Person object, nor with generating new Person objects.
2. Write a main method which starts a number of threads that each generate Person objects, and try to argue or show that the id generator works and do not suffer from race-conditions.
3. Did your solution to the previous question prove that no race-condition exist, or did it give evidence that no race condition exist.

Exercise 2.2 Consider the lecture's example in file TestMutableInteger.java, which contains this definition of class MutableInteger:

```
class MutableInteger {        // WARNING: USELESS IN THIS FORM
  private int value = 0;
  public void set(int value) {
    this.value = value;
  }
  public int get() {
    return value;
  }
}
```

As said in the Goetz book and the lecture, this cannot be used to reliably communicate an integer from one thread to another, as attempted here:

```
final MutableInteger mi = new MutableInteger();
Thread t = new Thread(new Runnable() { public void run() {
      while (mi.get() == 0) { }
      System.out.println("I completed, mi = " + mi.get());
  }});
```

```
t.start();
System.out.println("Press Enter to set mi to 42:");
System.in.read();                    // Wait for enter key
mi.set(42);
System.out.println("mi set to 42, waiting for thread ...");
try { t.join(); } catch (InterruptedException exn) { }
System.out.println("Thread t completed, and so does main");
```

Green

1. Compile and run the example as is. Do you observe the `"main"` thread's write to `mi.value` remains invisible to the `t` thread, so that it loops forever?
2. Now declare both the `get` and `set` methods `synchronized`, compile and run. Does thread `t` terminate as expected now?
3. Now remove the `synchronized` modifier from the `get` methods. Does thread `t` terminate as expected now? If it does, is that something one should rely on? Why is `synchronized` needed on **both** methods for the reliable communication between the threads?

Yellow

4. Remove both `synchronized` declarations and instead declare field `value` to be `volatile`. Does thread `t` terminate as expected now? Why should it be sufficient to use `volatile` and not `synchronized` in class MutableInteger?

Exercise 2.3 Consider class DoubleArrayList in TestLocking1.java. It implements an array list of numbers, and like Java's ArrayList it dynamically resizes the underlying array when it has become full.

Green

1. Explain the simplest natural way to make class DoubleArrayList thread-safe so it can be used from multiple concurrent threads.
2. Discuss how well the thread-safe version of the class is likely scale if a large number of threads call `get`, `add` and `set` concurrently.

Yellow

3. Now your notorious colleague Ulrik Funder suggests to improve the code by introducing a separate lock for each method, roughly as follows:

```
private final Object sizeLock = new Object(), getLock = new
    Object(),
  addLock = new Object(), setLock = new Object(), toStringLock =
      ...;
public boolean add(double x) {
  synchronized (addLock) {
    if (size == items.length) {
      ...
    }
    items[size] = x;
    size++;
    return true;
  }
}
public double set(int i, double x) {
  synchronized (setLock) {
```

```
      if (0 <= i && i < size) {
        double old = items[i];
        items[i] = x;
        return old;
      } else
        throw new IndexOutOfBoundsException(String.valueOf(i));
    }
  }
```

Would this achieve thread-safety? Explain why not. Would it achieve visibility? Explain why not.

Exercise 2.4 *Yellow*

Consider the extended class DoubleArrayList in TestLocking2.java. Like the class in the previous exercise it implements an array list of numbers, but now also has a static field `totalSize` that maintains a count of all the items ever added to any DoubleArrayList instance.

It also has a static field `allLists` that contains a hashset of all the DoubleArrayList instances created. There are corresponding changes in the `add` method and the constructor.

1. Explain how one can make the class thread-safe enough so that the `totalSize` field is maintained correctly even if multiple concurrent threads work on multiple DoubleArrayList instances at the same time. You may ignore the `allLists` field for now.
2. Explain how one can make the class thread-safe enough so that the `allLists` field is maintained correctly even if multiple concurrent threads create new DoubleArrayList instances at the same time.

Exercise 2.5 *Red*

Make a performance measurement which shows the cost (if any) to updating volatile fields in closed loops.

Exercise 2.6 *Red*

At the end of section 3.5.3 in *Goetz*, it says that *"Static initializers are executed by the JVM at class initialization time; because of internal synchronization in the JVM, this mechanism is guaranteed to safely publish any objects initialized in this way [JLS 12.4.2].* . JLS is short for Java Language Specification.

- Find the appropriate JLS section (make sure it fits your java version), and explain how this guarantee is achieved.
- Explain which (implicit) locks are involved in the *lazy initialization holder class* idiom illustrated in listing 16.6 in Goetz.