Exercises week 2

Last update: 2025/08/31

The goal of these exercises is to give you practical experience on:

- Solving concurrency problems using monitors.
- Using Java implicit locks (`synchronized`).
- Reasoning about and solving visibility issues in concurrent programs.

Exercise 2.1 Consider the Readers and Writers problem we saw in class. As a reminder, here is the specification of the problem:

- Several reader and writer threads want to access a shared resource.
- Many readers may access the resource at the same time as long as there are no writers.
- At most one writer may access the resource if and only if there are no readers.

Mandatory

1. Use Java Intrinsic Locks (i.e., `synchronized`) to implement a monitor ensuring that the access to the shared resource by reader and writer threads is according to the specification above. You may use the code in the lectures and Chapter 8 of Herlihy as inspiration, but do not feel obliged copy that structure.
2. Is your solution fair towards writer threads? In other words, does your solution ensure that if a writer thread wants to write, then it will eventually do so? If so, explain why. If not, modify part 1. so that your implementation satisfies this fairness requirement, and then explain why your new solution satisfies the requirement.
3. Does your monitor implementation contain any condition variables? If you answer yes, explain how many condition variables your monitor uses, how they are used in the monitor, and what method calls are used to access them. If you answer no, explain why condition variables are not used in the monitor.

Challenging

3. Is it possible to ensure absence of starvation using `ReentrantLock` or intrinsic java locks (`synchronized`)? Explain why.
4. Using `ReentrantLock`, implement a monitor for readers and writers as specified above, but with two condition variables: one for readers and one for writers. Make sure that you only signal conditional variables if necessary. Explain why your solution only signals condition variables if necessary and why it is free from deadlocks.

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

```
// WARNING: Not ready for usage by concurrent programs
class MutableInteger {
  private int value = 0;
  public void set(int value) {
        this.value = value;
  }
  public int get() {
        return value;
  }
}
```

For instance, as mentioned in Goetz, this class 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(() -> {
  while (mi.get() == 0)         // Loop while zero
  {/* Do nothing*/ }
  System.out.println("I completed, mi = " + mi.get());
});
t.start();
try { Thread.sleep(500); } catch (InterruptedException e) {
   e.printStackTrace(); }
mi.set(42);
System.out.println("mi set to 42, waiting for thread ...");
try { t.join(); } catch (InterruptedException e) { e.printStackTrace(); }
System.out.println("Thread t completed, and so does main");
```

Mandatory

1. Execute 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? Independently of your observation, is it possible that the program loops forever? Explain your answer.
2. Use Java Intrinsic Locks (`synchronized`) on the methods of the `MutableInteger` to ensure that thread `t` always terminates. Explain why your solution prevents thread `t` from running forever.
3. Would thread `t` always terminate if `get()` is not defined as `synchronized`? Explain your answer.
4. Remove all the locks in the program, and define `value` in `MutableInteger` as a `volatile` variable. Does thread `t` always terminate in this case? Explain your answer.

Exercise 2.3 Consider the small artificial program in file `TestLocking0.java`. In class `Mystery`, the single mutable field `sum` is `private`, and all methods are `synchronized`, so superficially the class seems that no concurrent sequence of method calls can lead to race conditions.

Mandatory

1. Execute the program several times. Show the results you get. Are there any race conditions?
2. Explain why race conditions appear when `t1` and `t2` use the `Mystery` object. Hint: Consider (a) what it means for an instance method to be synchronized, and (b) what it means for a static method to be synchronized.
3. Implement a new version of the class `Mystery` so that the execution of `t1` and `t2` does not produce race conditions, *without* changing the modifiers of the field and methods in the `Mystery` class. That is, you should not make any static field into an instance field (or vice versa), and you should not make any static method into an instance method (or vice versa).

   Explain why your new implementation does not have race conditions.
4. Note that the method `sum()` also uses an intrinsic lock. Is the use of this intrinsic lock on `sum()` necessary for this program? In other words, would there be race conditions if you remove the modifier `synchronized` from `sum()` (assuming that you have fixed the race conditions in 3.)?