ASSIGNMENT 2 SOLUTIONS

Exercise 2.1

1. The class written as a solution for this question can be seen in the github repository in a java file with the name “Person.java”

2. The class written as a solution for this question can be seen in the github repository in a java file with the name “PersonCreator.java”
   The id generator works by having 2 variables in the Person class. The first of these variables, which is the ‘currentId’ variable keeps track of the current id that has been generated by the Person class and the second of these variables which is the ‘id’ variable keeps track of the id of a person object that has just been created. When the createId() method is called, it increments the value of the currentId by 1 and then assigns the new value of currentId to the id of the new person object that has just been created. This createId() method does not suffer from race conditions because it is ‘synchronized’, thus ensuring that only one thread can execute the block of code within it at any given point in time.

3. Yes. My solution to the previous question proves that no race-condition exists.

Exercise 2.2

1. I observe that "main" thread's write to mi.value remains invisible to the t.thread and so it loops forever.

2. The thread t terminated as expected now.

3. The thread terminates as expected when the 'synchronized' keyword is taken from the get method. Even though the thread terminates as expected when the get method is not synchronized, it is not something that one should rely on. The synchronized is needed for both methods to ensure that there is visibility among all of the threads for any change in shared variables that occur.

4. The thread ‘t’ terminates as expected now. It should be sufficient to use 'volatile' and not 'synchronized' because 'volatile' ensures that there is visibility with regards to any changes in shared variables by both threads.

Exercises 2.3

1. In order to ensure that the class, DoubleArrayList, is thread safe, we need to ensure atomicity, visibility and reordering is done correctly,
   so different threads doesn't access a value that is in a inconsistent state.

   That is, making sure that threads are reading the field values from the main (shared) memory, and not from a thread's cache memory, which
   is not shared across cores. As well, we need to ensure that whenever a value is updated, it is flushed down to the main memory so
   other threads have access to the updated value.

   To do that, we need to synchronise all methods that perform reads of field properties or method that are manipulating the state of a field value.

   Using the keyword volatile would not have been sufficient to make the class thread safe, because volatile only ensures that the operation is
   done in the main memory. However, if a thread comes into a race condition with another thread, we are not ensured, because of unlucky-timing, that when
   a thread reads a value from the object that another thread is just about to do the same, and we will get an error.

2. Ensuring thread-safety for the class comes with a (small) price. When ensuring visibility and thread-safety, we are forcing the threads to read
   from the main memory, as well as to flush all variables from local cache to the main memory.
   This will affect the performance of the methods. However, the performance should not be affected to a noticeably amount, unless there is a very large amount of threads that call the methods many times. In this case, we would expect a downgrade on performance.

3. This will by no means ensure thread-safety nor visibility.

   Thread-safe
   The class is not thread-safe because every method gets a new and unique explicit lock. The lock will henceforth only lock within the one method call.
   If one thread calls the add method, it will acquire a new lock, which will never lock anything.

Let's say we have two threads, t1 and t2. T1 calls the add method, and acquires a new lock. At the same time, t2 does the same. It goes to the add method,
where it should be blocked by the lock acquired in t1, but it isn't. Instead, it just creates a new lock.

Visibility
The class does not ensure visibility because when retrieving the field values, we are not guaranteed that the value we read is updated with the latest change
another thread could have done. We might read the value from the thread's cache memory and not from the main memory. Same goes for when we are setting
a new value. We are not ensured that the new value is flushed to the main memory where other threads can read the updated value from.

Exercise 2.4

1. First step is to put a explicit lock on the reflective class in the block where we are incrementing the totalSize as follows:

```
synchronized (TestLocking2.class) {
   totalSize++;
}
```

An explicit lock on the class itself is needed to enforce thread-safety on the static field of totalSize across instances of the class.
Moreover, even if we introduced thread-safety in the class, we did not ensure visibility for the totalSize field.
A thread that calls the read method (totalSize()) may read an inconsistent state of the field value. Therefore,
we need to make sure that every read and write to the totalSize field is done in the main memory.
Using the keyword Volatile on the field value itself ensures exactly this.

```
private volatile static int totalSize = 0;
```

Now we have ensured visibility for the totalSize field, as well as made it thread-safe.

2. In order to make the field allLists thread-safe, we need to make sure that every time the class is instantiated, the class is added
to the HashMap correctly. In order to achieve this, we again need to put an explicit lock on the reflective class on the block
that adds it to the static field of allLists. The reason why an explicit lock is needed is because the we need to keep

track of the number of times the class is instantiated across instances, hence the allLists field needs to be available to the reflective class
and not the instance class. Therefore, we need to lock on the level of the class and not on the level of an instance.
Again, to ensure visibility, the get method (allLists()) needs to ensure that we are reading the value from the main memory.
Making the field of allLists volatile ensures just that.

Exercise 2.5

The performance measurement that was used to show the cost of updating the volatile fields is 'elapsed time'. In using the metric of elapsed time, the aim was to find out how long it took to update the volatile fields of the class. This was implemented by calling the System.nanotime() method **before** the volatile fields were updated and also calling the System.nanotime() method **after** the volatile fields had been updated and taking the difference between the two times.
After running this performance measurement 3 times, the results are shown below:
Estimated time cost of updating the volatile variable: 2762007
Estimated time cost of updating the volatile variable: 3205560
Estimated time cost of updating the volatile variable: 2566377

Exercise 2.6

1. According to the Java Language Specification for Java 11, when a field is declared static, "there exists exactly one incarnation of the field, no matter how many instances (possibly zero) of the class may eventually be created." As a result of this, the value of a field that has been declared as 'static' in a class is visible to all of the instances of that class. This ensures that the value of that static field is safely published.

2. The implicit locks involved in the lazy initialization holder class holder idiom can be found in the **static instance variable** 'resource' which has been declared in the **static class** 'ResourceHolder' , which can be found within the ResourceFactory class. Since 'ResourceHolder' is a static class, its state would be visible to all of the threads that are executing on different instances of the ResourceFactory class. Also, since the 'resource' variable is a static variable, any change in its value would be visible to all the threads that are executing on different instances of the ResourceHolder class.