Homework on Threads

- Create a Google Doc containing your answers and submit to Google Classroom for this assignment.
- Write only your answers in the Google Doc, not the questions.
- Include your name in the doc. Use good English with complete sentences.

1. Write a class named Accumulator that keeps a long value, with these public methods:.

   `void add(int amount)` - add the amount to the total value (a long) of accumulator

   `long get( )` - return the total value in the Accumulator

The initial value of the accumulator is 0.

Then, create an application class that launches 2 threads as shown here.

```
public class ThreadSum {

	public static void main( String[] args )
	{
		// upper limit of numbers to add/subtract to Accumulator
		int LIMIT = 10000;
		Accumulator accumulator = new Accumulator();
		runThreads( accumulator, LIMIT );
	}

	/** Run the AddTask and SubtractTask simultaneously using threads. */
	public static void runThreads(Accumulator accum, int limit )
	{
		// two tasks that add and subtract values using the same Accumulator
		AddTask addtask = new AddTask( accum, limit );
		SubtractTask subtask = new SubtractTask( accum, limit );

		// threads to run the tasks
		Thread thread1 = new Thread( addtask );
		Thread thread2 = new Thread( subtask );

		// start the tasks
		System.out.println("Starting threads");
		long startTime = System.nanoTime();
		thread1.start();
		thread2.start();
		System.out.println("Threads started");

		// wait for threads to finish
		try {
			thread1.join();
			thread2.join();
		} catch (InterruptedException e) {
			System.out.println("Threads interrupted");
		}
		double elapsed = 1.0E-9*( System.nanoTime() - startTime );

		// the sum should be 0.  Is it?
		System.out.printf("Accumulator total is %d\n", accum.get() );
		System.out.printf("Elapsed %.6f sec\n", elapsed);
	}

	/** AddTask adds number 1 ... limit to the accumulator total. */
	public static class AddTask implements Runnable {
		private Accumulator acc;
		private int limit;
		public AddTask(Accumulator acc, int limit)
```

```
            { this.acc = acc;  this.limit = limit; }  //TODO reformat code

            public void run() {
                for(int k=1; k<=limit; k++) {
                    acc.add(-k);
                    // show progress
                    if (k%1000 == 0) System.out.printf("AddTask: %d%n", k);
                }
            }
      }

      /** SubtractTask subtracts 1 ... limit from the accumulator total. */
      public static class SubtractTask implements Runnable {
            private Accumulator acc;
            private int limit;
            public SubtractTask(Accumulator acc, int limit)
            { this.acc = acc;  this.limit = limit; }  //TODO reformat code

            public void run() {
                for(int k=1; k<=limit; k++) {
                    acc.add(-k);
                    // show progress
                    if (k%1000 == 0) System.out.printf("SubTask: -%d%n", k);
                }
            }
      }
}
```

The AddTask adds 1, 2, .., LIMIT to the accumulator, and SubtractTask adds -1, -2, .. -LIMIT to the accumulator.

Obviously the total should be 1 + 2 + .. + LIMIT - 1 - 2 .. - LIMIT = 0.

Test your Accumulator. For example (in BlueJ or Jshell):

```
> Accumulator acc = new Accumulator();
> acc.add(50);
> acc.add(-15);
> acc.get()      // returns 35
```

1.1 Run the program several times and describe the results. Is the final value of accumulator always the same? Is it correct?

(If you *consistently* get a final value of 0 when you run this, then set LIMIT to a larger value.)

1.2 Explain the results. Why is the accumulator total sometimes not zero? Why is it not consistent?

1.3 Look at the progress messages printed by AddTask and SubtractTask. Are they always printed in nice alternatiing order (addtask subtask, addtask, subtask)? Is the order of progress messages the same each time you run the application?

Explain why, in terms of threads getting a turn to use the CPU.

(If the messages are always in same order try either a) increase LIMIT or b) print more frequently.)

2. The way in which the AddTask and SubtractTask are competing to change the accumulator is called what? (The name has 2 words.) The answer is in the *Multithreading* chapter of Big Java.

3. Explain how this behavior could affect a web application, where many users send requests to the web application. Each client request is handled in a separate thread on the web server.

Give a concrete example of some work of web application and how it could produce incorrect results if the programmer is not careful about handling thread competitition as in problem 1.

4. Create a subclass of **`Accumulator`** named **`AccumulatorWithLock`**. Override the `add()` method to use a **`ReentrantLock`** (see the BIGJ chapter 20, section 20.4 for ReentrantLock). The code is like this:

```
public class AccumulatorWithLock extends Accumulator {
	private Lock lock = new ReentrantLock();

	public void add(int amount) {
		try {
			lock.lock();
			super.add(amount);
		} finally {
			lock.unlock();
		}
	}
}
```

Modify the ThreadSum class to create an `AccumulatorWithLock` instead of `Accumulator`:

```
Accumulator accumulator = new AccumulatorWithLock( );
```

4.1 Run the program a few times and describe the results.

4.2 Explain why the results are different from problem 1.

5. Create another subclass of **`Accumulator`** named **`SynchronousAccumulator`**.
In **`SynchronousAccumulator`**, override the **`add( )`** method and declare it to be "synchronized" (see BIGJ, section 20.5 and the box "Special Topic 20.2"). **Don't use** a ReentrantLock in this class!

```
public class SynchronousAccumulator extends Accumulator {

	//TODO override add(int amount) method and declare it "synchronized"
}
```

Modify the ThreadSum class to create a `SynchronousAccumulator` instead of `Accumulator`.

```
Accumulator accumulator = new SynchronousAccumulator( );
```

5.1 Run the program a few times and describe the results.

5.2 Explain why the results are different from problem 1.

6. Finally, create another subclass of Accumulator named AtomicAccumulator. In this class, change total to be an **`AtomicLong`**.

```
public class AtomicAccumulator extends Accumulator {
	private AtomicLong total;

	public AtomicAccumulator() {
		total = new AtomicLong();
	}
	/** add amount to the total. */
	public void add(int amount) {
		total.getAndAdd(amount);
	}
	/** return the total as a long value. */
	public long get() {
		return total.get();
	}
}
```

Modify the ThreadSum class to use an AtomicAccumulator:

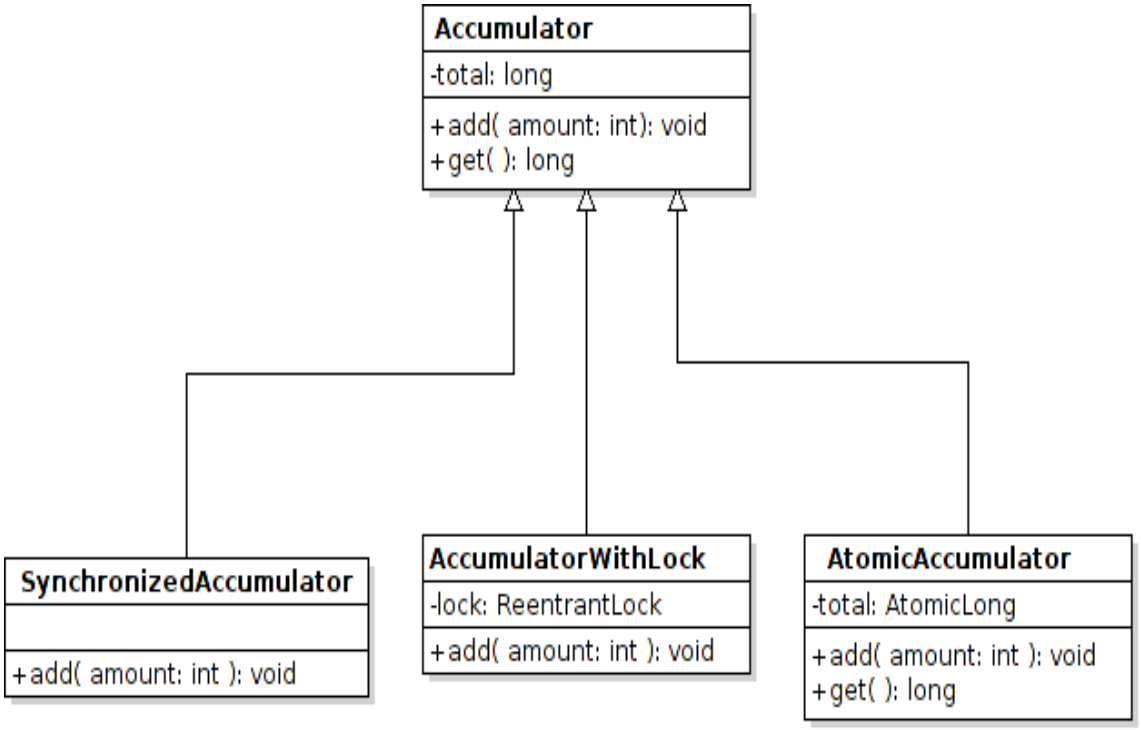
```
Accumulator accumulator = new AtomicAccumulator( );
```

6.1 Run the program a few times. AtomicAccumulator does not use a lock (like problem 3) and the add method isn't synchronized, but it still fixes the error in problem 1. Explain why.

7.1 Now you have 3 "thread safe" solutions to the Accumulator in problem 1. Which one is fastest? Which is slowest?

7.2 Which of the above solutions can be applied to the broadest range of problems where you need to ensure that only thread modifies some resource at any one time? The "resource" could be a more complex than adding to a single variable -- it could be modifying a List (as in the Coin Purse) or writing to a file..

Give an explanation for your answer.



8. **Optional**: Try each of the above cases using a different number of threads. For example, 10 adder threads and 10 substracter threads. Does using many threads affect which implementation is fastest?