Homework 4

1. Create 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

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 accum = new Accumulator();
		// two tasks that send "add" messages to 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();
		// 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. */
	public static class AddTask implements Runnable {
		private Accumulator acc;
		private int limit;
		public AddTask(Accumulator acc, int limit)
		{ this.acc = acc;  this.limit = limit; }

		public void run() {
			for(int k=1; k<=limit; k++) {
				acc.add(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; }

		public void run() {
			for(int k=1; k<=limit; k++) {
```

```
                        acc.add(-k);
                }
            }
        }
}
```

The AddTask adds 1, 2, .., LIMIT to the accumulator, and SubtractTask adds -1, -2, .. -LIMIT to the accumulator. Obviously the total is 1 + 2 + .. + LIMIT - 1 - 2 .. - LIMIT = 0. (If you *consistently* get 0 when you run this, then set LIMIT to a larger value.)

Test your Accumulator. For example (in BlueJ):

```
> Accumulator acc = new Accumulator();
> acc.add(20);
> acc.add(15);
> acc.get()      // returns 35
```

1.1 Run the program several times and describe the results.

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

2. Explain how this behavior could affect a banking application, where customers can deposit, withdraw, or transfer money via ATM, e-banking, or bank teller. Many transaction involving the same account could occur at the same time.

3. 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 application class to create an `AccumulatorWithLock` instead of `Accumulator`:

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

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

3.2 Explain why the results are different from problem 1.

4. Create another subclass of **`Accumulator`** named **`SynchronousAccumulator`**.
In **`SynchronousAccumulator`**, override the **`add( )`** method and declare it to be "synchronous" (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) and declare it "synchronous"
}
```

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

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

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

4.2 Explain why the results are different from problem 1.

5. 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 an int value. */
	public long get() {
		//TODO return the value of total
	}
}
```

Modify the application class to use an AtomicAccumulator:

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

5.1 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.

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

6.2 Which of the above solutions can be applied to the broadest range of problems where you need to ensure that only thread modifies the resource at any one time? The "resource" could be a lot more complex than adding to a single variable (such as a List) . Give an explanation for your answer.