Homework 3

Due: 7 April 2015

1. Create a class named Accumulator as shown in the UML below.

   `add(int amount)` - add the amount to `total`

   `getTotal( )` - return the value of total

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

```
public class ThreadSum {

	// upper limit of numbers to add/subtract to Accumulator
	static final int LIMIT = 1000;
	public static void main( String[] args )
	{
		Accumulator accum = new Accumulator();
		// two tasks that send "add" messages to same accumulator
		AddTask addtask = new AddTask( accum );
		SubtractTask subtask = new SubtractTask( accum );
		// threads to run the tasks
		Thread thread1 = new Thread( addtask );
		Thread thread2 = new Thread( subtask );
		// start the tasks
		System.out.println("Starting tasks");
		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.getTotal());
		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;
		public AddTask(Accumulator acc) { this.acc = acc; }
		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;
		public SubtractTask(Accumulator acc) { this.acc = acc; }
		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, then set LIMIT to a larger number.)

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

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

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 **`AtomicInteger`**.

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

	public AtomicAccumulator() {
		total = new AtomicInteger();
	}
```

```
        /** add amount to the total. */
        public void add(int amount) {
              total.getAndAdd(amount);
        }
        /** return the total as an int value. */
        public int getTotal() {
              //TODO
        }
}
```

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 restrict access to a resource (so that only one thread modifies the resource at any one time)? Give an explanation for your answer.