Starter Code for lab: **https://goo.gl/PCfJ41**

1. Create a class named Counter that has a protected **long** value (the total), with these public methods:

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

   `long get( )` - return the total value in this Counter

Counter
total: long
+add(amount: int): void +get( ): long

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 Counter
        final int LIMIT = 10000;
        // The counter that accumulates a total.
        Counter counter = new Counter();

        runThreads( counter, LIMIT );
    }

    public static void runThreads( Counter counter, final int limit )
    {
        // two tasks that add and subtract values using same Counter
        AddTask addtask = new AddTask( counter, limit );
        SubtractTask subtask = new SubtractTask( counter, 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 );
        System.out.printf("Count 1 to %,d in %.6f sec\n", limit, elapsed);

        // the sum should be 0.  Is it?
        System.out.printf("Counter total is %d\n", counter.get() );
    }
```

```
}

/**
 * AddTask adds number 1 ... limit to the counter, then exits.
 */
public class AddTask implements Runnable {
    private Counter counter;
    private int limit;

    public AddTask(Counter counter, int limit) {
        this. counter = counter;
        this.limit = limit;
    }

    public void run() {
        for(int k=1; k<=limit; k++) counter.add( k );
         // If you want to see when a thread finishes, add this line:
         // System.out.println("Done "+Thread.currentThread().getName());
    }
}

//TODO Write a SubtractTask class. It subtracts numbers from the Counter.
```

Test your Counter class first. In BlueJ:

```
> Counter acc = new Counter();
> acc.add(50);
> acc.add(-15);
> acc.get()     // returns 35
> acc.add(-34);
> acc.get()     // returns 1
```

1.1 Run the ThreadSum program using limit = 1,000 (a small number). Run it several times (5 or more). The total should be zero. Is it? Is the total always the same?

Explain the result in the project `README.md` file.

1.2 Run the ThreadSum program 3 times using limit = 10,000,000. If your computer is very fast (run time < 0.01 second) then use a larger limit. If run time > 1.0 second, use a smaller limit (1,000,000).

Record the *average* runtime in the project README.md. You'll need it for problem 6.

1.3 Explain the results. Why is the counter total sometimes not zero? Why is it not the same each time?

2. Explain how this behavior could affect a Banking application, where customers can deposit, withdraw, or transfer money to/from a Bank Account. The customer might use ATM, e-banking, mobile banking, or bank teller. He might also receive a transfer to his account at any time (even while he is using mobile banking), or two people might initialize transactions simultaneously.

Business bank accounts can have several authorized users and may have *lots* of activity.

3. Create a subclass of **`Counter`** named **`CounterWithLock`**. 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 CounterWithLock extends Counter {
    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 `CounterWithLock` instead of `Counter`:

```
Counter counter = new CounterWithLock( );
```

Run the program a few times, then answer these questions. Write your answers in README.md.

3.1 Describe the results. Is the total always zero? Record the average runtime in README.md

3.2 Explain why the results are different from problem 1.

3.3 What does a ReentrantLock do? Why (and when) would you use it in a program?

3.4 Why do we write "**`finally { lock.unlock(); }`**" in the code?

4. Create another subclass of **`Counter`** named **`SynchronousCounter`**.

In **`SynchronousCounter`**, override the **`add( )`** method and declare it to be "synchronized". This is described in BIGJ, section 20.5 and the box "Special Topic 20.2".

Don't use ReentrantLock in this class!

```
public class SynchronousCounter extends Counter {

    @Override
    public synchronized void add(int amount) {
        // To avoid the overhead of calling the superclass add,
        // define the superclass attribute (total) as protected and
        // directly add to the total here. Don't define a new attribute
        // in this class!
    }
}
```

Modify the application class to create a `SynchronousCounter` instead of `Counter`.

```
Counter counter = new SynchronousCounter( );
```

Run the application a few times and then answer these questions:

4.1 Describe the results. Is the total 0? Return the average run time in README.md.

4.2 Explain why the results are different from problem 1.

4.3 What is the meaning of "synchronized"? Why (and when) would you use it in a program?

5. Finally, create another subclass of Counter named AtomicCounter. In this class, change total to be an **`AtomicLong`**. AtomicLong is a class in the Java API that performs "atomic" operations.

```
public class AtomicCounter extends Counter {
    private AtomicLong total;

    public AtomicCounter() {
        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() {
        //TODO How to you get the long value of an AtomicLong? (see API)
    }
}
```

Modify the application class to use an AtomicCounter:

```
Counter counter = new AtomicCounter( );
```

5.1 Run the program a few times. AtomicCounter 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.

5.2 Describe why and when you would use AtomicLong (or AtomicDouble, AtomicInteger) in a program.

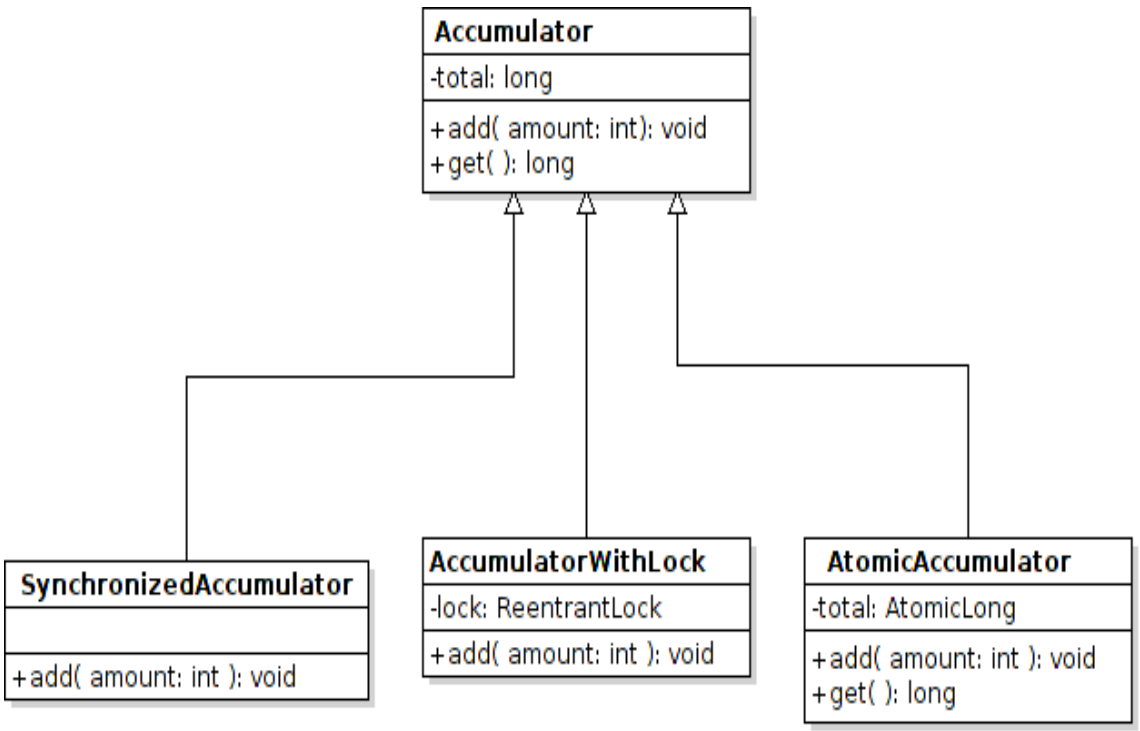
6. Now you have 3 "thread safe" solutions to the Counter in problem 1.

6.1 Compare the average run-times of all the solutions. 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 one 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 and example to support your answer.

Note: In this diagram the "Counter" is named "Accumulator". *Accumulator* is a widely used term for an object that accumulates something. For example, accumulator for a total (as in this case), or accumulator for "accumulating" other objects.



7. (Optional) Modify the Main class to create **10 threads** (5 adders and 5 subtracters). Record the run times again. Which is the fastest? Which is slowest?

To see when each thread finishes, print a message at the end of run(). Using an *Anonymous Class*:

```
Runnable addtask = new Runnable() {
    public void run() {
        for(int k=1; k<=limit; k++) counter.add( k );
        System.out.println("Done "+Thread.currentThread().getName());
    }
};
```

You need to create many threads and then wait for them all to finish (thread.join()).

Instead of writing redundant code like this:

```
thread1.join();
thread2.join();
thread3.join();
thread4.join();
...
```

Let's use an *ExecutorService* to manage our threads. An *ExecutorService* manages running tasks (both *Runnable* and *Callable* objects) using a *pool* of threads. There are many kinds of ExecutorService. For this program a fixed size thread pool is a good choice. Create an Executor Service like this:

```
int poolsize = 10;
ExecutorService executor = Executors.newFixedThreadPool( poolsize );
```

To submit a *Runnable* or *Callable* task to the executor for running in a Thread, use:

```
executor.submit( task );  // task implements Runnable or Callable
```

If you submit more tasks than the number of threads in the executor, it will wait for some tasks to finish before starting others. We want all threads to run at the same time, so make the pool big enough for all.

To wait for all the tasks to finish, you need two commands:

```
// shutdown tells Executor not to accept any more jobs
executor.shutdown();
executor.awaitTermination( 1, TimeUnit.MINUTES ); // wait at most 1 minute
System.out.println("All down");
```

Executor has other methods for running threads and getting the result (in case of *Callable* tasks).

Using an Executor, your code will be more compact:

```
/** Run nthread adders and nthread subtracters. */
public static void runThreads( int nthread, Counter counter, final int limit)
{
    ExecutorService executor = Executors.newFixedThreadPool(2*nthread);
    // start the tasks
    System.out.println("Starting tasks");
    long startTime = System.nanoTime();
    for(int k=1; k<=nthread; k++) {
        Runnable addtask = // create new add task
        Runnable subtask = // create new subtract task
        executor.submit(addtask);
        executor.submit(subtask);
    }
    //TODO wait for threads to finish
    //TODO print elapsed time and counter value
```