Project 0: Parallel Sum

Goals
The point of this project is to get you familiarized with BitBucket, Eclipse, JUnit, and give you a taste of parallelism in Java.

You will implement a class which computes the sum of a list of numbers. Since the list is relatively large, and you want to make use of all the computing power available (multiple CPUs), you will perform the sum in a parallel manner, by using threads.

Enlist the BitBucket repository
The first thing you have to do is fork the repository containing project 0.

You first have to create a BitBucket account. Please make sure that your first and last names are set in the account settings page of BitBucket. For example:

Click on Account. Fill in your first and last name, and click Save settings.
This way it will be easier for us to associate your project solution with your name.

Secondly, you have to fork the project. In our case, Project0. Go to:

https://bitbucket.org/cmsc433_spring2012/project0/overview and click on fork:

In the fork settings screen, make sure that the name you use for the project is <Project Name>-<Directory ID>. Also, make sure that you check the Private checkbox, so that other students can't see your solution. For example:
Now click on Fork Repository.

Now, in the Git window, paste the clone address associated with the cloned repository. Something like:

```bash
git clone https://jondoe@bitbucket.org/jondoe/project0-jondoe.git
```

You're done. Now you can load the project in Eclipse.

**Project structure**

You are given three classes:

**ParallelAdder**

This is the frame class, which receives, at construction, the number of threads to use in computing the sum: `threadCount`.

It contains just one method: `int sum(LinkedList<Integer> numbers)` which is supposed to return the sum of all the numbers in the given list.

The way `sum()` works is: it creates exactly `threadCount` ParallelAdderWorker instances, and starts all of them. Then waits for each of them to finish, and returns the sum of their partial results.

You don't need to modify this class in any way.

**ParallelAdderWorker**

This is the class that does all the work. It is a subclass of Thread, which means that it can be run on a separate thread, and its `run()` method will be called when `Thread.start()` is called. You will need to make it work properly (a bit later).
The constructor receives a list of numbers that the worker will add together in its run() method. Since the list is shared across all workers, some synchronization will be needed at run time.

The int partialSum() method returns the partial result obtained so far.

Finally, the void run() method has a loop in which it extracts one number at a time from the list, adding it to the partial sum, that is stored internally.

You will enlist the skeleton of the project, and then you'll have to implement a set of simple modifications to it, to make it do what it's supposed to.

**ParallelAdderTest**
This is a JUnit test class. In order to be able to test your code, and make sure it's working properly, you'll first have to write a test for it, and that's what this class is for.

It contains just one method, corresponding to the test you'll have to write: void testTotalSum().

**Implementation**

The first step is to run the test method, which now contains only one fail statement. This means that the test will always fail, because its logic hasn't yet been implemented.

Run your test once. In order to do this, right-click on the class containing the test (ParallelAdderTest.java), and in the context menu, choose Run As > JUnit Test:
The test should fail with message “The test is not implemented.”:

So now you have to implement your test. Instantiate the ParallelAdder class, and initialize it with a thread count of 100:

```java
int threadCount = 100;
ParallelAdder adder = new ParallelAdder(threadCount);
```
Then you’ll create a list which will contain the numbers from 1 to 10,000 in order:
```java
int numbersCount = 10000;
LinkedList<Integer> numbers = new LinkedList<Integer>();
for (Integer i=1; i<=numbersCount; ++i)
  {  
    numbers.add(i);
  }
```

Now you’ll call the sum() method of the ParallelAdder instance you’ve just created. For each kind of exception caught, the test should fail with the corresponding message.
```java
int actualSum = 0;
try
{  
  actualSum = adder.sum(numbers);
}
catch (InterruptedException e)
{  
  e.printStackTrace();  
  fail("The test failed because the sum procedure was interrupted unexpectedly.");
}
catch (Exception e)
{  
  e.printStackTrace();  
  fail("The test failed because the sum procedure encountered a runtime error: ", e.getMessage());
}
```

Finally, if no error has occurred during the sum procedure, you’ll test that the expected sum is the same as the sum computed by the ParallelAdder instance:
```java
int expectedSum = (numbersCount * (numbersCount+1)) / 2;

assertEquals("The expected sum doesn't match the actual sum.", expectedSum, actualSum);
```

Here’s the complete code for the test method:
```java
@Test
public void testTotalSum()
{
  int threadCount = 100;

  ParallelAdder adder = new ParallelAdder(threadCount);

  int numbersCount = 10000;
  LinkedList<Integer> numbers = new LinkedList<Integer>();
  for (Integer i=1; i<=numbersCount; ++i)
    {
      numbers.add(i);
  }
```
int actualSum = 0;
try {
    actualSum = adder.sum(numbers);
} catch (InterruptedException e) {
    e.printStackTrace();
    fail("The test failed because the sum procedure was interrupted unexpectedly.");
} catch (Exception e) {
    e.printStackTrace();
    fail("The test failed because the sum procedure encountered a runtime error: " + e.getMessage());
}

int expectedSum = (numbersCount * (numbersCount+1)) / 2;
assertEquals("The expected sum doesn't match the actual sum.", expectedSum, actualSum);

Run the test again. It still fails, but now, for a different reason: some of the threads are running infinite loops! This is because LinkedList is not a thread safe data structure, so its internal data got corrupted, and although at some point it becomes empty, its isEmpty() method still returns false.

Let’s fix this. Replace the old code in ParallelAdderWorke.run() with the following:

```java
@Override
public void run() {
    while (true) {
        synchronized (this.numbers) {
            if (this.numbers.isEmpty()) { return; }

            Integer n = this.numbers.poll();
            this.partialSum += n;
        }
    }
}
```

Run the test again. Again, it fails! But this time, for a different reason: “The expected sum doesn’t match the actual sum. expected:<50005000> but was:<0>". This is because although the run() method computes the sum correctly, getPartialSum() always returns 0.
Change `getPartialSum()` so that it returns the computed partial sum:

```java
public Integer getPartialSum()
{
    return this.partialSum;
}
```

Now your code is complete. Run the test again. It should pass:

Submit your solution
Now you're ready to submit your solution. In your Git window, type:

```bash
git add .
```

This should update all the changes you've made in the directory.

Now commit your changes locally, with the message “This is the solution for project 0."

```bash
git commit -m "This is the solution for project 0."
```

Finally, you're ready to push your changes to the server:

```bash
git push
```

You're done!