APIT, Lab Book 2

Dr. Simon Rogers

Jan 2019

Introduction and aims

This lab book covers the material in the concurrency section of the course. As before, do them at your own pace.

Exercises 1: creating threads

Various exercises to practise making Threads and Runnable objects.

Tasks

1. Create a class that implements the `Runnable` interface. Within the `run()` method sleep for a random number of seconds and then print a message to the Console. In a `main` method, create a `Thread` object (passing an instance of your `Runnable` class) and start the `Thread`. You will need to catch the `InterruptedException` when you call `Thread.sleep()`.
2. Repeat the previous exercise, but this time extend `Thread` instead of implementing `Runnable`
3. Using your `Runnable` class, create a new `main` method that creates multiple instances of the `Runnable` class in different threads and starts them all.
4. Repeat exercise 3 but this time give each `Thread` a name and have the `Thread` display its name as part of the message when it finished.
5. Have your `main` method print a message after it has started all of the `Thread`s. This message should appear before the `Thread`s have finished. If it doesn't, you're not starting the `Thread`s properly.
6. Use `Thread.join()` to get the thread running `main` to wait for all other `Thread`s to finish before it prints its message.

Exercise 2: max finder

This exercise is designed to help you create a *race condition*. The following code will create a 2-dimensional array of random `Doubles` between 0 and 1:

```
int nRows = 100;
int nCols = 50;
Double[][] randArray = new Double[nRows][nCols];
for(int r=0;r<nRows;r++) {
    for(int c=0;c<nCols;c++) {
        randArray[r][c] = Math.random();
    }
}
```

Tasks

1. Within a `main` method, write some simple code (a couple of `for` loops) to find the maximum value in `randArray`. You will use this to test your threaded solution.
2. Create a class that implements `Runnable` that finds the maximum of a 1D array. The constructor should take three arguments: the 1D array, an *array* in which to store the result and the position in the result array for it to use. Within your `main` create a single instance of this object for each row in `randArray` (remember that `randArray[r]` is a one-dimensional array). Pass each thread the same 1D array in which to store the results (`Double`, length `nRows`) and give each one a different position in which to store their maximum value. Start and join all threads. Once they have all finished, you will be left with a single 1D array holding the maximum values of each row in the original array.
3. Pass this new array to a final instance of your `Runnable` object and an associated `Thread`. If you pass it a length 1 Double array for the result, and position 0 it should end up with the maximum value from the whole array. Start and join this `Thread` and print the final value that it finds. It should agree with the value you obtained with your initial (non-threaded) loops.
4. So far, no race condition. To induce a race condition create a new version of your `Runnable` object that instead of being passed an array for the results is passed an instance of the following object (all Threads should be passed the same instance):

```
public class SharedDouble {
    private Double d;
    public Double getD() {
        return d;
    }
    public Double setD(Double d) {
        this.d = d;
    }
}
```

Within your `Runnable` class, at each iteration through its row, your `Runnable` object should `getD()` to get the current *global* maximum and then, if the value in its array is bigger, `setD` with the new value. Start and join all of the threads and once they have all finished, print `SharedDouble.getD()`. Do you see the same as in the loop solution?

5. You may well not see a race condition - if you think about the circumstances required for one to be visible then you should see that it is very unlikely. However, it is possible to make it more likely. Place a `Thread.sleep(10)` within the `run` method of your `Runnable` object in a position that makes the race condition more likely to take place. You should be able to make it such that the Threaded version disagrees with the non-threaded version more often than not.

6. Create a copy of your code and refactor it such that all comparisons are done in the `SharedDouble` object. I.e. `SharedDouble` becomes (including the `sleep` to help with the races):

```
public class SharedDouble {
    private Double d;
    public Double getD() {
        return d;
    }
    public void compare(Double a) {
        if(a > d) {
            try {
                Thread.sleep(1);
            }catch(InterruptedException e) {
                e.printStackTrace();
            }
            d = a;
```

```
        }
    }
}
```

7. You now have two versions of the code. Fix one with a `synchronized` block or method and the other with a `lock` (this will only work one way around!).

Exercise 3 - locks and conditions

In this exercise you will use conditions to help avoid deadlocks. Download `SimpleStack.zip`. This consists of three classes that simulate a system in which computational jobs are added to a stack (by a thread) and then removed by other threads.

Tasks

1. Compile and run the code. After a while a deadlock will occur.
2. Inspect the code and convince yourself that you understand what's going on. It's not totally straightforward. And you might find it helpful to comment out some things from main.
3. Fix the deadlock through the use of `Condition`, a single `await()` line and a single `signal()` or `signalAll()` line.

Exercise 4 - Swingworkers

A few things to help familiarise you with `SwingWorker` objects

Tasks

1. Create a Swing application that has a `JTextField` and two buttons: `Start` and `Stop`. When the user clicks `Start` a `SwingWorker` object should be created that takes the `int` in the `JTextField` and counts down, one step per second until either it reaches zero or the user clicks `Stop`.
2. Extend your solution to part 1 to make the `Start` button unclickable when the system is counting and the `Stop` button unclickable when the system is stopped.
3. **Primes**: Last semester you wrote a program that could determine if a number was a prime. In this exercise, use your prime computation method to create a Swing application that, when you press start, keeps finding primes until you press stop. It should have display all the primes in a `JtextArea` and display the value of the prime and which number it is. I.e. 2 is prime number 1, 3 is prime number 2, 5 is prime number 3, etc. So the display should look like:

1: 2

2: 3

3: 5

4: 7

etc

Note: you should **not** allow your SwingWorker to directly add lines to the JTextArea. Instead you shoul use `publish` and `process`. Note that when you call `publish` (every time you find a prime) you need to pass an object. You should create an object that can hold the two integers (prime, and which number it is). Also remember that Java calls `process` (although you have to implement it). It is passed a `List` of all of the objects published since `process` was last called. Loop over these objects, adding each to the `JTextArea`. You may need to pass a reference to the `TextArea` to the worker, or pass a reference to the `JFrame` object and

provide a public method that can append a `String` to the `JTextArea`. It's interesting to print out how long the `List` is that is passed to `process`.