5COSC001W - Tutorial 11 Exercises

1 Testing more your Knowledge on Threads

Here is some segment of code for a class that implements the Runnable interface:

```
public class Whiffler implements Runnable {
    Thread myT ;

    public void start(){
        myT = new Thread( this );
    }

    public void run(){
        while( true ){
            doStuff();
        }
        System.out.println("Exiting run");
    }
    // more class code follows....
}
```

Assume that the rest of the class defines method `doStuff()` , and so on, and that the class compiles without error. Also assume that a Java application creates a Whiffler object and calls the Whiffler start method and that no other direct calls to Whiffler methods are made and that the thread in this object is the only one the application creates. Which of the following are correct statements?

1. The doStuff method will be called repeatedly.
2. The doStuff method will never be executed.
3. The doStuff method will execute at least one time.
4. The statement in line 10 will never be reached.

2 Higher Level Locks

Convert the code given to you previously for the *producer-consumer* example shown again below to use the `ReentrantLock` class.

```
public class plum {
    public static void main(String args[]) {
        Producer p = new Producer();
        p.start();

        Consumer c = new Consumer(p);
        c.start();
    }
}

class Producer extends Thread {
    private String [] buffer = new String [8];
    private int pi = 0;  // produce index
    private int gi = 0;  // get index

    public void run() {
       // just keep producing - for ever
       for(;;) produce();
    }

    private final long start = System.currentTimeMillis();

    // method producing some random number strings
    private final String banana() {
        return "" + (int) (System.currentTimeMillis() - start);
    }

    synchronized void produce() {
        // while there isn't room in the buffer
        while ( pi-gi+1 > buffer.length ) {
            try {wait();} catch(Exception e) {}
        }

        buffer[pi%8] = banana();  // fill one datum to buffer
        System.out.println("produced["+(pi%8)+"] " + buffer[pi%8]);
        pi++;
        notifyAll();
     }

     synchronized String consume(){
         // while there's nothing left to take from the buffer
         while (pi==gi) {
             try {wait();} catch(Exception e) {}
         }
         notifyAll();

         return buffer[ gi++ % 8 ];
    }
}
```

```
class Consumer extends Thread {
    Producer whoIamTalkingTo;

    // constructor communicating with another object (producer)
    Consumer(Producer who) {
        whoIamTalkingTo = who;
    }

    public void run() {
        java.util.Random r = new java.util.Random();
        for(;;) {  // for ever consume
            String result = whoIamTalkingTo.consume();
            System.out.println("consumed: "+result);

            // just to make it run a bit slower.
            int randomtime = Math.abs(r.nextInt() % 250);
            try {
              sleep(randomtime);
            } catch(Exception e){}
        }
    }
}
```

3 Swing and Threads

Create a Swing application which contains one button and one label. Each time the button is pressed a new time consuming thread is created (create a long loop combined with a sleep to simulate this). As soon as the second thread finishes its task it displays its result inside the label of the main application.

You should use the `SwingWorker` class.

4 Interrupted Threads

Create a program with two threads. The first one is the main thread that every Java application runs. The main thread creates a second thread and waits for it to finish. if the second thread takes too long to finish (e.g. based on a number of seconds given as a command line argument when the application runs), the main thread interrupts it.