5COSC019W - Solutions to Tutorial 10 Exercises

1 Running a Thread

The last option is the only correct one. Inside `main`, the `start` method should be called and the `run` method should be defined as opposed to `start`. Simply calling method `run` will never execute method `start`.

Calling method `run()` directly on an object of a class which extends `Thread` but does not override `run()` will simply call the default implementation of the `run()` method of class `Thread` (check the Java API documentation of class Thread in the Oracle website). The default implementation (if a `Runnable` object is not attached to the `Thread object`) simply "does nothing and returns".

The correct code which fixes that is:

```
public class Background extends Thread {
    public static void main(String argv[]) {
        Background b = new Background();
        b.start();   // <--- MODIFIED LINE
    }

    public void run() {   // <--- MODIFIED LINE
        for (int i = 0; i < 10; i++) {
            System.out.println("Value of i = " + i);
        }
    }
}
```

2 More on Problematic Threads

Threads should call method `start()` to execute their `run()` method as some behind the scenes work needs to be done before starting a thread.

In fact, directly calling method `run()` will not create a new thread and simply the method will be executed sequentially like any other method. In the code given, the first call `t1.run()` will block the last statement (`t2.start()`) which will never be executed!

Run the code and make sure that you understand what is happening, or otherwise ask your tutor!

3 Testing your knowledge on Threads

The first answer `a)` is the only correct one. It correctly creates a new thread attached to the runnable object and starts it.

Answer `b)` is incorrect because the new thread is not attached to the runnable object, so it cannot find the run method. Instead, the default run in the Thread class is executed. Answer `c)` is incorrect because the thread that is executing the start method calls run in the Thread class. The `myT` Thread is not started.

4 Unsynchronised Threads

```
class Increaser extends Thread {
    Object o1;

    Increaser(Object o) {
        o1 = o;
    }

    public void run() {
        synchronized(o1) {
            if (UnsynchronisedThreads2.x + 20 <= 100) {
                try {
                    sleep(100);
                }
                catch (InterruptedException e) {
                    System.out.println("thread increaser was interrupted");
                }

                System.out.println(getName() + " Increaser is executing " +
                                                UnsynchronisedThreads2.x);

                UnsynchronisedThreads2.x = UnsynchronisedThreads2.x + 20;
            }
        }
    }
}


class Decreaser extends Thread {
    Object o1;

    Decreaser(Object o) {
        o1 = o;
    }

    public void run() {
        synchronized(o1) {
            if (UnsynchronisedThreads2.x - 20 >= 0) {
                try {
                    sleep(100);
                }
                catch (InterruptedException e) {
```

```
                    System.out.println("thread decreaser was interrupted");
                }

                System.out.println(getName() + " Decreaser is executing " +
                                                UnsynchronisedThreads2.x);

                UnsynchronisedThreads2.x = UnsynchronisedThreads2.x - 20;
            }
        }
    }
}

class UnsynchronisedThreads2 {
    public static int x = 0;   // range of x should remain within [0, 100]
    public static Object o1 = new Object();  // object to synchronise on it

    public static void main(String[] args) {
        Thread myThreads[] = new Thread[20];

        for (int i=0; i <= 9; i++) {
            myThreads[i] = new Increaser(o1);
            myThreads[i+10] = new Decreaser(o1);

            myThreads[i].start();
            myThreads[i+10].start();
        }


        /* Wait for all threads to finish */
        try {
            for (int i=0; i < 20; i++)
                myThreads[i].join();
        }
        catch (InterruptedException e){
            System.out.println("thread was interrupted");
            e.printStackTrace();
        }

        System.out.println("\n\n-> after all threads executed, x is: " +
                                    UnsynchronisedThreads2.x);
    }
}
```

5 Challenge: Race Conditions (Data Races)

The problem is that the methods of `ArrayList` are not defined as `synchronized`, therefore reading and writing concurrently into such a list results in a corruption as the following sample code illustrates:

```java
import java.util.ArrayList;
import java.util.Random;

/**
   This demo for demonstrating how threads can corrupt an ArrayList.
*/
public class ArrayListThreadCorruption implements Runnable {
    private static ArrayList<String> list;
    private static Random randomizer;

    public void run() {
        int choice = list.size();
        if (list.size() > 0) {
            choice = randomizer.nextInt(list.size() + 1);
        }

        if (choice == list.size()) {
            String token = randomizer.nextInt(1000) + "";
            list.add(token);
            System.out.println(Thread.currentThread().getName() + "- Added: " + token);
        }
        else { // remove a random element
            System.out.println(Thread.currentThread().getName() +
                               "- removing index " + choice);
            String token = list.remove(choice);
            System.out.println(Thread.currentThread().getName() +
                               "- Removed: " + token);
        }

        if (list.size() > 0) {
            run();
        }
    }

    public static void main(String[] args) {
        randomizer = new Random();
        list = new ArrayList<>();

        Thread thread1 = new Thread(new ArrayListThreadCorruption());
        thread1.start();
        Thread thread2 = new Thread(new ArrayListThreadCorruption());
        thread2.start();
    }
}
```