Quiz 7

Threaded design and thread-safety questions.

Question # 1

Can you enumerate the implications of the poor design choice for the below class?

```
public class BadClassDesign {

    private File file;

    public BadClassDesign() throws InterruptedException {
        Thread t = new Thread(() -> {
            System.out.println(this.file);
        });
        t.start();
        t.join();
    }
}
```

The above class is a bad design choice for the following reasons:

- When creating the thread object in the constructor, the reference to the instance of the enclosing `BadClassDesign` class is also implicitly captured by the anonymous class that implements `Runnable`. The problem with this approach is that the anonymous class can attempt to use the enclosing object while it is still being constructed. This would not be an issue if we didn't start the thread in the constructor. Note that if we invoked an overrideable instance method in the constructor, we'll be giving a derived class a chance to access the half constructed object in an unsafe manner.

- The private fields of the `BadClassDesign` class also become accessible to the instance of the anonymous inner class that we pass in to the `Thread` class's constructor.

```
import java.io.File;

class Demonstration {
    public static void main( String args[] ) throws Exception {
      BadClassDesign bcd = (new BadClassDesign());
    }
}

class BadClassDesign {

    // Private field
    private File file;

    public BadClassDesign() throws InterruptedException {
        Thread t = new Thread(() -> {
            System.out.println(this.getClass().getSimpleName());

            // Private field of class is accessible in the anonymous class
            System.out.println(this.file);
        });
        t.start();
        t.join();
    }
}
```

Question # 2

What is stack-confinement in the context of threading?

All local variables live on the executing thread's stack and are confined to the executing thread. This intrinsically makes a snippet of code thread-safe. For instance consider the following instance method of a class:

```
    int getSum(int n) {
```

```
        int sum = 0;
        for (int i = 1; i <= n; i++)
            sum += i;
        return sum;

    }
```

If several threads were to simultaneously execute the above method, the execution by each thread would be thread-safe since all the threads will have their own copies of the variables in the method above.

Primitive local types are always stack confined but care has to be exercised when dealing with local reference types as returning them from methods or storing a reference to them in shared variables can allow simultaneous manipulation by multiple threads thus breaking stack confinement.