Quiz 2

Questions on thread-safety and race conditions

Question # 1

What is a thread safe class?

A class is thread safe if it behaves correctly when accessed from multiple threads, irrespective of the scheduling or the interleaving of the execution of those threads by the runtime environment, and with no additional synchronization or other coordination on the part of the calling code.

Question # 2

Is the following class thread-safe?

```
public class Sum {

    int sum(int... vals) {

        int total = 0;
        for (int i = 0; i < vals.length; i++) {
            total += vals[i];
        }
        return total;
    }
}
```

Q

Check Answers

Show Explanation

The class `Sum` is stateless i.e. it doesn't have any member variables. All stateless objects and their corresponding classes are thread-safe. Since the actions of a thread accessing a stateless object can't affect the correctness of operations in other threads, stateless objects are thread-safe.

However, note that the method takes in variable arguments and the class wouldn't be thread safe anymore if the passed in argument was an array instead of individual integer variables and at the same the time, the `sum` method performed a write operation on the passed in array.

Question # 3

What is a race condition

A race condition occurs when the correctness of a computation depends on the relative timing or interleaving of multiple threads by the runtime; in other words, getting the right answer relies on lucky timing. Two scenarios which can lead to a race condition are:

- **check-then-act**: Usually the value of a variable is checked and then an action is taken. Without proper synchronization, the resulting

code can have a race condition. An example is below:

```
        Object myObject = null;
        if (myObject == null) {
            myObject = new Object();
        }
```

- **read-modify-write**: For instance, whenever a counter variable is increment, the old state of the counter undergoes a transformation to a new state. Without proper synchronization guards, the counter increment operation can become a race condition.

Question # 4

Given the following code snippet, can you work out a scenario that causes a race condition?

```
1.  class HitCounter {
2.
3.      long count = 0;
4.
5.      void hit() {
6.          count++;
7.      }
8.
9.     long getHits() {
10.         return this.count;
11.     }
}
```

The following sequence will result in a race condition.

1. Say `count = 7`
2. Thread A is about to execute line #6, which consists of fetching the `count` variable, incrementing it and then writing it back.

3. Thread A reads the count value equal to 7
4. Thread A gets context switched from the processor
5. Thread B executes line#6 atomically and increments `count = 8`
6. Thread A gets scheduled again
7. Thread A had previously read the `count = 7` and increment it to 8 and writes it back.
8. The net effect is `count` ends up with a value 8 when it should have been 9. This is an example of read-modify-write type of race condition.

Question # 5

Given the following code snippet, can you work out a scenario that causes a race condition?

```
1.  class MySingleton {
2.
3.      MySingleton singleton;
4.
5.      private MySingleton() {
6.      }
7.
8.      MySingleton getInstance() {
9.          if (singleton == null)
10.             singleton = new MySingleton();
11.
12.         return singleton;
13.     }
14. }
```

This is the classic problem in Java for creating a singleton object. The following sequence will result in a race condition:

1. Thread A reaches line#9, finds the `singleton` object null and proceeds to line#10
2. Before executing line#10, Thread A gets context switched out
3. Thread B comes along and executes lines#9 and 10 atomically and the reference `singleton` is no more null.
4. Thread A gets scheduled on the processor again and new's up the `singleton` reference once more.
5. This is an example of a check-then-act use case that causes a race condition.