**Quiz 5**

Exercise on how to make classes thread-safe

**Question # 1**

***Is the following class thread-safe?***

public class Sum {  
   
    int count = 0;  
   
    int sum(int... vals) {  
   
        count++;  
   
        int total = 0;  
        for (int i = 0; i < vals.length; i++) {  
            total += vals[i];  
        }  
        return total;  
    }  
   
    void printInvocations() {  
        System.out.println(count);  
    }  
}

**Q**

**Your Answer**

**A)**

Yes

**Correct Answer**

**B)**

No

?????

**Oops, you got that wrong. Want to try again?**

Retake Quiz

**Hide Explanation**

The class isn't thread-safe because it has state that can be mutated by different threads without synchronization amongst them. The state consists of the variable **count**

**Question # 2**

***What are the different ways in which we can make the Sum class thread-safe?***

We can use an instance of the **AtomicInteger** for keeping the count of invocations. The thread-safe code will be as follows:

**Using Atomic Integer**

public class SumFixed {  
   
    AtomicInteger count = new AtomicInteger(0);  
   
    int sum(int... vals) {  
   
        count.getAndIncrement();  
   
        int total = 0;  
        for (int i = 0; i < vals.length; i++) {  
            total += vals[i];  
        }  
        return total;  
    }  
   
    void printInvocations() {  
        System.out.println(count.get());  
    }  
}

We can also fix the sum class by using synchronizing on the object instance.

**Using Synchronization on this**

public class SumFixed {  
   
    int count = 0;  
   
    synchronized int sum(int... vals) {  
   
        count++;  
   
        int total = 0;  
        for (int i = 0; i < vals.length; i++) {  
            total += vals[i];  
        }  
        return total;  
    }  
   
    synchronized void printInvocations() {  
        System.out.println(count);  
    }  
}

We could also use another object other than **this** for synchronization. The code would then be as follows:

public class SumFixed {  
   
    int count = 0;  
    Object lock = new Object();  
   
    int sum(int... vals) {  
   
        synchronized (lock) {  
            count++;  
        }  
   
        int total = 0;  
        for (int i = 0; i < vals.length; i++) {  
            total += vals[i];  
        }  
        return total;  
    }  
   
    void printInvocations() {  
        synchronized (lock) {  
            System.out.println(count);  
        }  
    }  
}

**Question # 3**

***In the above question, when we fixed the Sum class for thread safety we synchronized the printInvocations() method. What will happen if we didn't synchronize the printInvocations() method?***

The **printInvocations()** method performs a read-only operation of the shared variable **count**. If we skipped synchronizing the method, then the method call can potentially return/print stale value for the **count** variable including zero.

One may be tempted to skip synchronizing the read-only access of variables if the application logic can tolerate stale values for a variable but that is a dangerous proposition. Writes to the **count** variable may not be visible to other threads because of how the Java's memory model works. We'll need to declare the **count** variable ***volatile*** to ensure threads reading it see the most recent value. However, marking a variable ***volatile*** will not eliminate race conditions.

**Question # 4**

***If we synchronize the sum() method as follows, will it be thread-safe?***

    int sum(int... vals) {  
   
        Object myLock = new Object();  
        synchronized (myLock) {  
            count++;  
        }  
   
        int total = 0;  
        for (int i = 0; i < vals.length; i++) {  
            total += vals[i];  
        }  
        return total;  
    }

**Q**

**A)**

Yes

**Your Answer**

**B)**

No

**Great, you got it right!**

Retake Quiz

**Hide Explanation**

If multiple threads call into the **sum()** method then each thread will create a **myLock** object on it's thread stack on which it will synchronize. In order to ensure thread-safety, threads need to synchronize on the same object.