Java Concurrency

Q 1. Write a multithreaded version of the word occurrences counter program that looks up for the word Alice in 2 or more files. Each thread shall consider a file to lookup in it (Hint: the main passes the word to lookup for and one of the file names to each thread). The main shall wait till the last thread finishes to display the final total number of occurrences of the given word.

Q 2. Use the following unsynchronized counter class, which you can include as a nested class in your program:

```
static class Counter {
   int count;
     void inc() {
              count = count+1;
              }
     int getCount() {
              return count;
              }
     }
```

Write a thread class that will repeatedly call the `inc()` method in an object of type *Counter*. **The object should be a shared global variable**. Create several threads, start them all, and wait for all the threads to terminate. Print the final value of the counter, and see whether it is correct.
Let the user enter the number of threads and the number of times that each thread will increment the counter. **You might need a fairly large number of increments to see an error**. And of course there can never be any error if you use just one thread. Your program can use **join()** to wait for a thread to terminate.

Q 3. Write a program in which multiple threads add and remove elements from a java. util.LinkedList. Demonstrate that the list is being corrupted.

Q 4. Implement a stack as a linked list in which the **push**, **pop**, and **isEmpty** methods can be safely accessed from multiple threads.
Hint: You might use the methods in the classes ReentrantLock and Condition.

Q 5. **AllProducesAllConsum**ers Create a synchronized Queue class with methods add() and remove(). Implement multiple threads, each named "producer," to continuously add strings into the queue as long as it contains fewer than 10 elements (size() < 10). When the queue reaches its maximum capacity, all producers should wait until it becomes empty (size() == 0). For sample strings, utilize timestamps generated by new Date().toString(). Additionally, introduce other threads, named "consumer," responsible for removing and printing strings from the queue as long as it's not empty (size() > 0). When the queue is empty, consumers should wait until it's full again (size() == 10). Note that a producer can add only one element at a time, but it can add another element if other threads have produced an element. Similarly, a consumer can remove

only one element at a time, but it can consume another element if other threads have consumed an element.

Hint: You might use the methods in the classes ReentrantLock and Condition.

Q 6. Producer / Consumer:

You are implementing a Producer-Consumer scenario where multiple producer threads produce integer values and multiple consumer threads consume these values from a shared buffer. The buffer has a limited size and provides methods write (int value) to add an integer value to the buffer and read() to read and remove an integer value from the buffer.

Implement the Buffer class representing the shared buffer with limited size. The Buffer class should provide methods write(int value) to add an integer value to the buffer and read() to read and remove an integer value from the buffer. Ensure thread safety by synchronizing access to the shared buffer using appropriate synchronization mechanisms.

Implement producer threads that generate random integer values and add them to the buffer using the write() method.

Implement consumer threads that read integer values from the buffer using the read() method and process them.

Q7 **32.4** (*Synchronize threads*) Write a program that launches 1,000 threads. Each thread adds `1` to a variable `sum` that initially is `0`. You need to pass `sum` by reference to each thread. In order to pass it by reference, define an `Integer` wrapper object to hold `sum`. Run the program with and without synchronization to see its effect.

Q8 ***32.11** (*Demonstrate deadlock*) Write a program that demonstrates deadlock.