1. Below is a Java class Mas written by a student Max. The class extends the Thread class and is to be run in a thread. Give your comment to the code from the perspective of multithreaded programming.

```
class Mas extends Thread {
       public void run(){
              System.out.println("Running in a thread...");
       }
      public static void main(String args[]){
           Mas m1 = new Mas();
            m1.run();
      }
}
```

   The code will not be able to run concurrently/as expected. This is because the m1 thread is not started as Max invokes the run() method directly instead of using the start() method to properly start a thread.

2. What happens when the start() method of a thread class is called?.

   The start method will start a new thread of execution by calling run() method of Thread/Runnable object. The thread then moves from New state to Runnable state.

3. What is the error(s) in the code below? Explain your answer.

```
class X extends Y
{
    public static void main(String [] args)
    {
        X x1 = new X();
        x1.start();
        System.out.println("first run…");
        x2.start();
        System.out.println("second run…");
    }

    public void run()
    {
        System.out.print("I am the ");
    }
}
```

   The x2 thread has not been previously declared. Therefore, the program will not be able to run properly and produce the expected print output.

4. What is the concurrency issue faced by the code below? Explain your answer with an example.

```
class Stack<V> {
     …
     synchronized boolean isEmpty() { … }

      synchronized void push(V val) { … }
```

```
      synchronized V pop() {
            if(isEmpty())
            throw new StackEmptyException();
            …
       }

      V peek() {
           V ans = pop();
           push(ans);
           return ans;
      }
}
```

The main concurrency issue faced by the code below is a race condition (but not a data race). The issue lies with the peek generic method. The peek generic method in of itself has no overall effect on the shared data as it is a 'reader' not a 'writer' and therefore the state should be maintained. However, the way it is being implemented creates an inconsistent intermediate state that is exposed which causes wrong/bad interleavings. This is because the peek method is not synchronized, which allows multiple threads to access it at the same time.

The ideal property we want is if there has been a push and no pop, then isEmpty returns false. The program will fail with two threads if one is doing a peek, and other is checking ifEmpty. Consider 2 threads, T1 and T2. The race condition can occur when T2 invokes push() method and T1 invokes pop() method. If T2 calls isEmpty(), the ideal property is violated.

5. Write a multithreaded program to find which integer between 1 and 100000 has the largest number of divisors, how many divisors does it have, and prints out the results. It is possible that several integers in this range have the same maximum number of divisors. Your program only has to print out one of those integers and how many divisors it has.

```
public class Q5 {

    private final static int MAX_INT = 100000;
    private volatile static int maxDivisorCount = 0;
    private volatile static int intWithMaxDivisorCount;

    public static void main(String[] args) {
        int numOfThreads = 2;

        System.out.println("\nCounting divisors using " +
numOfThreads + " threads...");
        long startTime = System.currentTimeMillis();

        CountDivisorsRunnable[] threadArr = new
CountDivisorsRunnable[numOfThreads];
        Thread t[] = new Thread[numOfThreads];
        int intsInThread = MAX_INT / numOfThreads;

        int start = 1;
        int end = start + intsInThread - 1;

        for (int i = 0; i < numOfThreads; i++) {
            if (i == numOfThreads - 1) {
                end = MAX_INT;
            }

            threadArr[i] = new CountDivisorsRunnable(start, end);
            t[i] = new Thread(threadArr[i]);
            start = end + 1;
            end = start + intsInThread - 1;
        }

        maxDivisorCount = 0;

        // Start threads
        for (int i = 0; i < numOfThreads; i++)
            t[i].start();

        // Join threads to ensure all threads have completed before
termination
        for (int i = 0; i < numOfThreads; i++) {
            while (t[i].isAlive()) {
                try {
                    t[i].join();
                } catch (InterruptedException e) {
                }
            }
        }

        long elapsedTime = System.currentTimeMillis() - startTime;
        System.out.println("\nMaximum divisor " + "for numbers
between 1 and " + MAX_INT + " is: " + maxDivisorCount);
```

```
        System.out.println("An integer with max divisor " +
maxDivisorCount + " is: " + intWithMaxDivisorCount);
        System.out.println("Total elapsed time: " + (elapsedTime /
1000.0) + " seconds.\n");
    }

    synchronized private static void printResult(int
maxCountFromThread, int intWithMaxFromThread) {
        if (maxCountFromThread > maxDivisorCount) {
            maxDivisorCount = maxCountFromThread;
            intWithMaxDivisorCount = intWithMaxFromThread;
        }
    }

    private static class CountDivisorsRunnable implements Runnable {
        int min, max;

        public CountDivisorsRunnable(int min, int max) {
            this.min = min;
            this.max = max;
        }

        public void run() {
            int maxDivisors = 0;
            int whichInt = 0;
            for (int i = min; i < max; i++) {
                int divisors = countDivisors(i);
                if (divisors > maxDivisors) {
                    maxDivisors = divisors;
                    whichInt = i;
                }
            }
            printResult(maxDivisors, whichInt);
        }
    }

    public static int countDivisors(int N) {
        int count = 0;
        for (int i = 1; i <= N; i++) {
            if (N % i == 0)
                count++;
        }
        return count;
    }

}
```