Some questions about concurrent programming in Java

These questions assume that the student has taken some time to study these topics at least for a few hours before going through them. Theme-wise, the examples are extremely simple, to the point where we are not worried about Software Engineering when writing them. We want to avoid overencumbering the students with more code than would be the minimum necessary to illustrate the issue. Some of the classes defined here may be lacking imports, so that needs to be fixed before they compile.

1. What are threads?

In programming, specifically within the Java language, what can we say about threads? You can select multiple options.

a) Units of scheduling

b) They have a protected address space, which means that they do not share memory

c) They are more resource-intensive than processes

d) Their use creates new complications that are fundamentally different from those programmers face in their absence

e) Their execution is managed by the JVM

a, b, e (we also have all the regular problems)

For f, it actually depends. It is a qualified answer. Mostly, until very recently, their execution was only managed by the OS scheduler. Sun initially had the idea that "green threads", managed by the JVM, should also exist, and actually implemented them for old SUnOS versions (I don't know about Solaris). In practice, they are always OS threads. Recently, though, Java has incorporated virtual threads, which are JVM-managed.

2. Why threads?

Why would we use multiple threads in a program? You can select multiple options.

a) To simplify the design and implementation of that program

b) To make the program perform computations faster even when there is only one processor/core

c) To make the program perform computations faster, although only when there are two or more processors/cores

d) To avoid having the program stop execution when waiting for I/O operations

e) To enable the program to handle multiple requests in systems that may have multiple clients, e.g., Web servers

f) To modularize the system, enabling separate development of its parts

g) Threads guarantee that we can use the processing power of the underlying machine to its full extent.

c, d, e

About g, they don't guarantee anything. It depends on how parallelizable the program is (Amdahl's Law helps understand this notion).

3. Running with multiple threads

Consider the following Java program.

```
public class C {
    public static void main(String args[]) {
        Thread t1 = new Thread(new MyThread(1));
        Thread t2 = new Thread(new MyThread(2));
        t1.run();
        t2.run();
    }

}

class MyThread implements Runnable {
    private int id = 0;
    public MyThread(int id) {
        this.id = id;
    }

    public void run() {
        System.out.print(id + " ");
    }

}
```

Select the correct option about what it will output if executed.

a) It always prints 1 2, in this order.

b) It always prints 2 1, in this order.

c) It may print either 1 2 or 2 1. It is impossible to know the order.

d) It may print something altogether different, e.g., 1 1 or 2 2.

e) It does not compile.

a

This program is strictly sequential, as it invokes `run()` and not `start()`.

4. Running with multiple threads. Thread interleavings

Now consider the following slightly different Java program.

```
public class C {
    public static void main(String args[]) {
        Thread t1 = new Thread(new MyThread(1));
        Thread t2 = new Thread(new MyThread(2));
        t1.start();
        t2.start();
    }

}

class MyThread implements Runnable {
    private int id = 0;
    public MyThread(int id) {
        this.id = id;
    }

    public void run() {
```

```
        System.out.print(id + " ");
    }

}
```

Select the correct option about what it will output if executed.

a) It always prints 1 2, in this order.

b) It always prints 2 1, in this order.

c) It may print either 1 2 or 2 1. It is impossible to know which of the two.

d) It may print something altogether different, e.g., 1 1 or 2 2.

e) It does not compile.

c

Even for a parallel machine, we need to think in terms of instruction interleavings. The two threads may execute in any order and we have no control over that unless we explicitly enforce some kind of synchronization (more on that later), which this program does not do.

5. Thread ordering and data races.

Examine the code of the following program.

```
public class Counter {

    int ticks = 0;
    public static void main(String args[]) {
        Counter c = new Counter();
        Thread t1 = new Thread(new MyThread(c));
        Thread t2 = new Thread(new MyThread(c));
        t1.start();
        t2.start();
        System.out.println(c.ticks);
    }
}
class MyThread implements Runnable {

    private Counter counter;

    public MyThread(Counter count) {
        this.counter = count;
    }

    public void run() {
        counter.ticks += 1;
    }
}
```

Which values can it print to the console? You can select multiple options.

a) 0

b) 1

c) 2

d) A number greater than 2

e) Nothing

a, b, c

The point here is that this program has three threads: `t1`, `t2`, and the `main` thread. Since no synchronization is enforced among them, the `main` thread may be picked by the scheduler before either of the other threads, or even before both, which means that the `counter` may still be in its initial state or only updated by one of the threads.

The number 1 can be printed for more than one reason (ordering between the main thread and the other two, or because of a race condition).

6. Data races

Now consider the following program, which prints within the implementation of each thread:

```
public class Counter {

    int ticks = 0;
    public static void main(String args[]) {
        Counter c = new Counter();
        Thread t1 = new Thread(new MyThread(c));
        Thread t2 = new Thread(new MyThread(c));
        t1.start();
        t2.start();
    }
}
class MyThread implements Runnable {

    private Counter counter;

    public MyThread(Counter count) {
        this.counter = count;
    }

    public void run() {
        counter.ticks += 1;
        System.out.println(counter.ticks);
    }
}
```

Which values can it print to the console? You can select multiple options.

a) 0 0

b) 0 1

c) 1 0

d) 1 1

e) 1 2

f) 2 1

g) 2 2

d, e, f, g

Outputs with 0 are not possible because instruction reordering cannot happen within the threads in this example.

This program has **data races**. A data race is a situation where

- Two or more threads access the same memory position
- At least one of them performs a write
- There is no synchronization among them.

This can often lead to problems. It does here. There are many **interleavings** where the two threads, working together, do not work like a sequential program would.

Reorderings within the `run()` method of each thread in this case are not possible. In other words, we are sure that the increment will happen before printing to standard output. The Java compiler is now allowed to reorder within-thread instructions when there are dependencies between them. It **is** allowed to re-order instructions within a thread, when there are multiple threads, if there are no dependencies among instructions *in the same thread*. Example 17.4-1 of the Java Language Specification, re-enacted below, is very enlightening:
https://docs.oracle.com/javase/specs/jls/se8/html/jls-17.html

7. Data races and instruction re-ordering

Now look at this program which defines two types of threads.

```
public class D {
    static int varA = 0;
    static int varB = 0;
    public static void main(String args[]) {
        Thread t1 = new Thread(new MyThread1());
        Thread t2 = new Thread(new MyThread2());
        t1.start();
        t2.start();
    }
}

class MyThread1 implements Runnable {
    public void run() {
        int r1 = D.varB;
        D.varA = 2;
    }
}

class MyThread2 implements Runnable {
    public void run() {
        int r2 = D.varA;
        D.varB = 1;
    }
}
```

Immediately before the end of the execution of the two `run()` methods, i.e., while r1 and r2 are still in scope, which of the following are possible values of variables `r1` and `r2`, respectively? You can select multiple options.

a) 0 0

b) 1 0

c) 0 1

d) 1 1

e) 0 2

f) 2 0

g) 1 2

h) 2 1

i) 2 2

a, b, e, g

Result (g) looks impossible because it would require both D.varA = 2 and D.varB = 1 to come before the declarations of r1 and r2. However, since there are no data dependencies between subsequent instructions within the same thread, Java does not preclude them from being re-ordered. Not only is this valid. This example is provided in the Java Language Specification: https://docs.oracle.com/javase/specs/jls/se8/html/jls-17.html#jls-17.4

It is really difficult to think about this even for very simple programs. Data races make reasoning about program execution a nightmare. Furthermore, Java allows instruction reorderings to happen (at the compiler or architecture level) when there are no data dependencies between instructions. This has no effect on sequential programas, but may produce weird results in multi-threaded ones.

Also, some of the things we are discussing here are **allowed** by the JLS. They are possible, not guaranteed. Some may simply never happen in a practical scenario. the point is: the correctness of one's programs cannot depend on that.

8. Race conditions

In the example program below, what is the smallest possible value for attribute `ticks` of the `DecCounter` object created in method `main()` by the end of its execution?

```
public class DecCounter {

    int ticks = 100;
    public static void main(String args[]) {
        DecCounter c = new DecCounter();
        Thread t1 = new Thread(new MyThread(c));
        Thread t2 = new Thread(new MyThread(c));
        t1.start();
        t2.start();
    }
}
class MyThread implements Runnable {

    private DecCounter counter;

    public MyThread(DecCounter count) {
        this.counter = count;
    }
    public void run() {
        while (this.counter.ticks > 0) {
            counter.ticks -= 1;
        }
    }
}
```

a) 1

b) 0

c) -1

d) -2

e) Potentially lower than -2.

c

Threads `t1` and `t2` pass the condition of the loop with `this.counter.ticks == 1`. If `t1` is interrupted before executing the line that decrements the counter, `t2` decrements it (`this.counter.ticks == 0`). When `t1` resumes execution, it will read a value of 0 but not perform any checks. It will then decrease the value, making `this.counter.ticks == -1`. This situation where a specific thread interleaving can lead to programs behaving differently from any potential sequential execution is a **race condition**.

Bear in mind: this also includes a data race on the access to attribute `ticks`. Other problems can stem from that, e.g., ignored updates to the attribute.

9. Non-atomicity of `long` and `double`.

Take a look at the following program. It is worth pointing out that the type of variable `value` is `long`.

```
public class LongAssignment {
    static long value = 0;
    public static void main(String args[]) {
        Thread t1 = new Thread(new MyThread1());
        Thread t2 = new Thread(new MyThread2());
        t1.start();
        t2.start();
    }
}
class MyThread1 implements Runnable {
    public void run() {
        LongAssignment.value = 1;
    }
}
class MyThread2 implements Runnable {
    public void run() {
        LongAssignment.value = 17179869183L; // The 'L' here is necessary.
    }
}
```

Which of the following are valid final values of `value`, after threads `t1` and `t2` have finished (ignoring 2's complement representation and considering that every number in Java is signed)? You can select multiple options. Tip: think of the binary representations of the numbers.

a) 0

b) 1

c) 17179869183L

d) 17179869184L

e) 12884901889

f) 12884901888

g) 4294967295

b, c, e, g

Number 17179869183 is 34 bits with 1's. Cases b and c are trivial. d is not possible (only if we added up the values, instead of just assigning them to the shared variable)

`t2`: 11 11111111111111111111111111111111 `t1`: 00 00000000000000000000000000000001

Case (e) happens when we have the most significant two bits from `t2` and the 32 least significant bits from `t1`. Case (f) happens when we have the inverse: the two most significant bits from `t1` and the 32 least significant bits from `t2`.

To stop this from happening, we can make the variable `value` become `volatile` (this is a modifier to the attribute). This ensures that assignments and reads to/from this variable are **atomic**. This has a *small* performance cost and does not solve the problems of data races and race conditions/atomicity violations, but does ensure atomicity in variable updates.

Bear in mind: this is not guaranteed to happen. In some machines/compilers, it **never** will. The problem is that the JLS allows it to, which means that it can.