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.

```
## 10. Avoiding problems with synchronized.
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();
        }
    }
    class MyThread implements Runnable {

        private Counter counter;

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

        public void run() {
            synchronized(this.counter) {
                counter.ticks += 1;
                System.out.print(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

e

The `synchronized` modifier/block guarantees **mutual exclusion**, based on the use of monitors. Entering a synchronized scope (method or block) requires the thread to acquire the monitor for that scope and, for any monitor, either one thread holds it or none does. As a consequence, scopes protected by a monitor can only be entered by one thread at any given time, as long as the threads attempt to acquire the same monitor. Correct usage of monitors avoids data races and race conditions. Instructions within a `synchronized` scope are executed atomically, with respect to other threads attempt9ng to acquire the same monitor, i.e., within the synchronized scope, execution is sequential. This makes it easier to reason about correctness, but has a negative impact on performance.

Nice thing about synchronized: freeing the monitor is automatic and guaranteed. Any object can be a monitor. `synchronized` can be used with block or as a modifier to methods. This mechanism can be used to ensure **mutual exclusion**.

If we take the *sysout* line out of the `synchronized` block, then **f** is also correct, since the access to `ticks` will be unsynchronized and thus the to calls to *sysout* in the two threads can be reordered. Furthermore, IO operations are tricky because buffering can happen along the way and, as a consequence, the IO operations themselves can be reordered.

11. Incorrectly synchronized code

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();
    }
}
class MyThread implements Runnable {

    private Counter counter;

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

    public void run() {
        synchronized(this) { // CHANGED HERE
            counter.ticks += 1;
            System.out.print(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

From the perspective of multiple threads, **different monitors equals no monitors**. Thus, we may have interleavings that lead to weird results such as assignments being ignored. We cannot, however, for this example, have one of the threads printing 0, because there is a data dependency between lines incrementing the counter and printing out its value. Therefore, there can be no re-ordering within the threads, although there is still a data race in the access to `counter.ticks`. The JLS would not forbid strange reorderings to happen if the data dependency did not exist: an incorrectly synchronized program is similar to a program without synchronization.

During the lecture, a student asked whether option (d) was possible at all. I thought I might be mistaken but, on a more careful examination, yes, **option (d) is possible**. It would take an interleaving where counter.ticks never goes beyond the value of 1. In this interleaving, the following happens:

t1 reads counter.ticks == 0 t1 is interrupted t2 reads counter.ticks == 0 t2 is interrupted t1 updates counter.ticks to 1 (because it had previously read 0) t1 proceeds and prints 1 t2 updates counter.ticks to 1 (because it had previously read 0) t2 proceeds and prints 1

12. Exit synchronization

In the code snippet

```
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();
        try {
            t1.join();
            t2.join();
        } catch(InterruptedException ie) {}
        System.out.println(c.ticks);

    }
}
class MyThread implements Runnable {

    private Counter counter;

    public MyThread(Counter count) {
        this.counter = count;
    }
    public void run() {
        synchronized(this.counter) { // CHANGED HERE
            counter.ticks += 1;
        }
    }
}
```

If we remove the

```
try {
    t1.join();
    t2.join();
} catch(InterruptedException ie) {}
```

part, what are the possible outputs of the program? You can select multiple options.

a) -1

b) 0

c) 1

d) 2

e) 3

f) Numbers larger than 3 or smaller than -1.

b, c, d

Why is that? Because without the calls to `join()`, the main thread may execute the line `System.out.println(c.ticks)` before the counter is incremented (or maybe it is incremented only once). The join guarantees that threads `t1` and `t2` have finished before that line is executed, which ensures that the counter is incremented exactly twice, going from 0 to 2. This is independent of the `synchronized` block in method `run()` using the right monitor or not. In this program, we are using the right monitor and the problem may still occur. The two calls two `join()` block the main thread until the two sub-threads, `t1` and `t2`, can finish. Only then can the main thread proceed.

13. Limitations of synchronized

Consider the following propositions about how threads can use`synchronized` blocks and methods and select the ones that are true. You can select multiple options.

a) Threads can use `synchronized` to acquire exclusive access to resource but with a timeout, after which we just give up and do something else

b) Threads can use `synchronized` to obtain exclusive access to a resource by leveraging monitors

c) We can directly use `synchronized` to implement acquire and release resource patterns that are not organized in a block scope.

d) It is possible to interrupt a thread that is attempting to acquire exclusive access to a resource using `synchronized`.

e) A `synchronized` method can recursively call itself without blocking unintentionally.

b, e

Letter (e) requires some clarification about what we mean. A recursive call means that the method `m()` acquires the monitor then calls itself, which would lead to it attempting to acquire the same monitor again. Would this be a problem? Not if `synchronized` is **reentrant**, which it is. In C's pthreads library, we can have locks that are non-reentrant, which means that a thread that acquires a lock and attempts to acquire the same lock again will be blocked in that attempt forever, i.e., it will **deadlock**.

About letter (c), it is actually possible to implement non-blocked-scoped resource acquisition with `synchronized`, but it requires non-trivial, although not enormous, extra work. Basically, we need to implement a small library just to do that.

14. ReentrantLock example

The following program is part of a library to ensure exclusive access to resources. A resource may be a network connection, a file, a database connection, a memory region, etc. Users of the library are expected to invoke `beginOperation()` when acquiring the resource and `endOperation()` when releasing.

```
import java.util.concurrent.locks.ReentrantLock;
import java.util.concurrent.locks.Lock;

public class ResourceManager {
    private final Lock lock = new ReentrantLock();

    public boolean beginOperation() {
        int attempts = 3; // Number of attempts
        while (attempts > 0) {
            try {
                if (lock.tryLock()) { // Try to acquire the lock
                    return true; // Successfully acquired the lock
                } else {
                    Thread.sleep(100); // Wait before retrying
                }
            } catch (InterruptedException e) {
                System.err.println(Thread.currentThread().getName() + " was interrupted during lock attempts.");
                return false; // Give up on interruption
            }
            attempts--;
        }
        return false; // Failed after max attempts
    }

    public void endOperation() {
        lock.unlock(); // Release the lock
    }
}
```

After having examined the program, select the propositions that are correct among the following. You can select multiple options.

a) The program has a bug because it should be invoking `lock.lock()`, instead of `lock.tryLock()`.

b) Method `beginOperation()` always returns `false`.

c) The program has a bug because it is not invoking `lock.unlock()` right after locking, within the `beginOperation()` method.

d) The program has a bug because locking operations do not return `boolean`.

e) Method `tryLock()` is only invoked multiple times in the same execution of `beginOperation()` if some other thread holds the same lock.

f) Method `endOperation()` has a bug. It should be invoking `lock.tryUnlock()`.

e

Letter (a) is wrong because that would result in a type error. Furthermore, even if that weren't the case, it would unconditionally acquire the lock three times, which is not the expected behavior. Letter (d) is incorrect because `tryLock()` returns a `boolean` value indicating whether it succeeded or not in acquiring the lock. Letter (b) is trivially false: if the lock is successfully acquired, it returns `true`. Letter (c) is borderline nonsensical. That would

make the lock acquisition useless from a resource management perspective: mutual exclusion would not be ensured. Letter (f) is incorrect, as method `tryUnlock()` does not exist.

Explicit locks are nice because it is also possible to interrupt them if lock acquisition does not succeed immediately when using method `lockInterruptibly()`. Furthermore, it is possible to check whether some thread holds a lock or not by invoking `isLocked()` on the lock object.

Differently from monitors, which are always released by the JVM when the a thread holding it fails, there's no such guarantee for locks. The lock is not released. This is why it is important to invoke `unlock()` from within a `finally` block. There are even documented bugs in the implementation of the JDK itself related to this behavior: https://bugs.openjdk.org/browse/JDK-8028686

15. ReentrantLocks are monitors too and this may be confusing.

If it were correctly synchronized, the following program should always print 0 when it ends execution. It is not correctly synchronized, however.

```
import java.util.concurrent.locks.ReentrantLock;

public class E {
    static int count = 0;
    static ReentrantLock lock = new ReentrantLock();
    public static void main(String args[]) {
        while (true) {
            Thread t1 = new Thread(new MyThread1());
            Thread t2 = new Thread(new MyThread2());
            t1.start();
            t2.start();
            try {
                t1.join();
                t2.join();
            } catch (InterruptedException e) {}
            System.out.println(E.count); // should be 0
        }
    }
}
class MyThread1 implements Runnable {
    public void run() {
        synchronized (E.lock) {
            E.count += 1;
        }
    }
}
class MyThread2 implements Runnable {
    public void run() {
        E.lock.lock();
        try {
            E.count -= 1;
        } finally {
            E.lock.unlock();
        }
    }
}
```

After having examined the program, select the propositions that are correct among the following. You can select multiple options.

a) The program is incorrectly synchronized because a `ReentrantLock` object cannot be used as a monitor.

b) The program would be correctly synchronized if we made the body of the `run()` method of `MyThread1` equal to the `run()` method of `MyThread2`, except for incrementing the counter, instead of decrementing it, i.e., if we used the `lock` there.

c) The program would be correctly synchronized if we made the body of the `run()` method of `MyThread2` equal to the `run()` method of `MyThread1`, except for decrementing the counter, instead of incrementing it, i.e., if we used `synchronized` there (with `lock` as the monitor).

d) The program is incorrectly synchronized because the two threads are using different monitors.

e) The program is incorrectly synchronized because the two threads are using different locks.

b, c

Option (a) is incorrect because **any** object in Java can act as a monitor, including `Lock` objects. The thing is: a `ReentrantLock` object can be used to ensure mutual exclusion either as a monitor or as a lock. However, it must be the same for all threads trying to access the resource. In other words, using it as a monitor in one thread and as a lock in the other does not guarantee mutual exclusion.

Answers (d) and (e) are nonsensical. There's only one monitor and there's only one lock and each one is only being used in one thread.

16. Condition-based synchronization

Consider the `Producer` and `Consumer` classes. They implement threads with the homonym roles. A `Producer` could be, e.g., getting images from a satellite and providing them, byte by byte, to a `Consumer` responsible for processing them. The `Producer` cannot send more data than the `CAPACITY` of the `Consumer`'s buffer. Multiple `Producer` threads can send data to the same `Consumer` thread.

```
class Producer implements Runnable {
    private final Queue<Integer> buffer;
    private final Object monitor;
    private int value = 0;

    Producer(Queue<Integer> buffer, Object monitor) {
        this.buffer = buffer;
        this.monitor = monitor;
    }
    public void run() {
        while (true) {
            synchronized (monitor) {
                if (buffer.size() >= CAPACITY) {
                    try { monitor.wait(); } catch (InterruptedException e) {}
                }
                buffer.offer(++value);
                System.out.println("Produced: " + value);
                monitor.notifyAll();
            }
        }
    }
}
class Consumer implements Runnable {
    private final Queue<Integer> buffer;
    private final Object monitor;

    Consumer(Queue<Integer> buffer, Object monitor) {
        this.buffer = buffer;
        this.monitor = monitor;
    }
```

```
    public void run() {
        while (true) {
            synchronized (monitor) {
                while (buffer.isEmpty()) {
                    try { monitor.wait(); } catch (InterruptedException e) {}
                }
                int data = buffer.poll();
                System.out.println("Consumed: " + data);
                monitor.notifyAll();
            }
        }
    }
}
```

This program has a problem that makes deviate from its expected behavior. What is it?

a) The `Consumer` stays forever in the inner consumer loop without actually ever consuming anything

b) The `Producer` can produce more than the capacity of the buffer allows

c) The `Consumer` attempts to consume items even when the buffer is empty

d) The `Producer` uses the `notifyAll` when it should actually use `notify`

b

Letter (b) is the one describing the actual problem. The check for the buffer size is in an `if` statement, instead of a `while` loop. This means that multiple threads may be blocked waiting for the buffer to stop being full. When the `Consumer` takes an item out of the buffer, it notifies all these waiting `Producer`s, which will all try to add items to the buffer without first checking whether they are allowed, which may cause the buffer to overflow. If it were a `while` loop instead of an `if`statement, they would perform the check again before proceeding. Changing the notification in the `Consumer` to `notify()` (instead of `notifyAll()`) does not solve the problem because we are using only one condition variable (implicit to the monitor) to notify. Thus, when a `Producer` invokes `notifyAll()`, it will also notify waiting producers, which falls in the case of letter (b). One should always keep this kind of check within a loop and use `notifyAll()`. One approach to improve this implementation would be to have two conditions, one to keep track of a full buffer and one to keep track of an empty buffer. This is not possible with monitors, but it is possible with explicit locks.

When working with `wait()` and `notifyAll()`, one should think carefully about how monitor access is managed. A thread must have exclusive access to a monitor before invoking either method on it. Performing these invocations without first acquiring the corresponding monitor triggers an `IllegalMonitorStateException`. When a thread invokes `wait()` on a monitor it already holds, it also relinquishes the monitor, enabling other threads to acquire it and invoke `notifyAll()` on it. This kind of concurrent behavior is known as **cooperative concurrency**, as two or more threads need to work together to achieve a common goal.

17. Condition objects and their advantages

The `wait–notify` pattern can also be implemented using explicit `Lock` and `Condition` objects. This approach is even more flexible as one can have multiple `Condition`s associated to a single `Lock`. This code snippet shows how this can be implemented for the `Producer` and also for a `main()` method and a `Consumer` (not shown).

```
/* IMPORTS OMITTED */
public class ProducerConsumerTwoConditions {
    static final int CAPACITY = 5;

    public static void main(String[] args) {
        Queue<Integer> buffer = new LinkedList<>();
        Lock lockFull = new ReentrantLock();
        Lock lockEmpty = new ReentrantLock();
```

```
        Condition notFull = lockFull.newCondition();
        Condition notEmpty = lockEmpty.newCondition();

        Thread producer = new Thread(new Producer(buffer, lockFull, notFull, notEmpty));
        Thread consumer = new Thread(new Consumer(buffer, lockEmpty, notFull, notEmpty));

        producer.start();
        consumer.start();
    }
}
class Producer implements Runnable {
    private final Queue<Integer> buffer;
    private final Lock lock;
    private final Condition notFull;
    private final Condition notEmpty;
    private int value = 0;

    Producer(Queue<Integer> buffer, Lock lock, Condition notFull, Condition notEmpty) {
        this.buffer = buffer;
        this.lock = lock;
        this.notFull = notFull;
        this.notEmpty = notEmpty;
    }
    public void run() {
        while (true) {
            lock.lock();
            try {
                while (buffer.size() >= ProducerConsumerTwoConditions.CAPACITY) {
                    notFull.await();  // Wait for space in the buffer
                }
                buffer.offer(++value);
                System.out.println("Produced: " + value);
                notEmpty.signalAll();  // Signal consumers that buffer is not empty
            } catch (InterruptedException e) {
            } finally { lock.unlock(); }
        }
    }
}
```

This snippet has a small problem, though, and would not work as expected, in terms of constraints to add elements to or remove from the buffer. What is that problem?

a) It is using methods `signalAll()` and `await()`, instead of the correct ones, `wait()` and `notifyAll()`.

b) The `main()` method is not passing the two `Lock` objects as arguments to the `Producer` constructor.

c) The two `Condition` objects are associated to different `Lock` objects.

d) It is invoking `await()` and `signalAll()` on different `Condition` objects. Instead, we should have one for the `Producer` and the other for the `Consumer`.

c

Why use two `Condition` variables? First, because using two condition variables makes it clearer which condition a thread is waiting for (e.g., buffer full vs. buffer empty). Furthermore, with two condition variables, only the threads waiting on the relevant condition are notified, reducing unnecessary wake-ups.

Letter (c) is the problem because the two conditions must be associated to the same lock object, because we still need to have a single mutual exclusion region, so as to stop any two threads from accessing the buffer at the same time. Letter (b) is nonsensical. Using two locks is wrong, but the problem is not that the two are not being passed. It is not letter (a) because the methods here are `signalAll()` and `await()`, not `wait()` and `notifyAll()`. Why? Because every

object in Java can be a monitor and therefore every method implements `wait()` and `notifyAll()`, thus we would need to use different method names to work with explicit `Condition` objects. Letter (d) is nonsensical. Both conditions are necessary to both threads. `Producer` threads stop when the buffer is full and notify `Consumer` threads and when it is not anymore empty. Conversely for `Consumer`s when the buffer is empty or stops being full.

18. Utility classes for concurrent and parallel programming: AtomicInteger

This program presents an implementation of a **thread-safe** counter that is started with an initial value and decreases that value when the `decrement()` method is called. The value of the counter can never be less than 0 and the program should have no data races or race conditions.

```
public class NonNegativeCounter {
    private final AtomicInteger value;

    public NonNegativeCounter(int initialValue) {
        if (initialValue < 0) {
            throw new IllegalArgumentException("Initial value cannot be negative");
        }
        this.value = new AtomicInteger(initialValue);
    }
    public boolean decrement() {
        while (true) {
            int current = value.get();
            if (current == 0) {
                return false;
            }
            if (value.compareAndSet(current, current - 1)) {
                return true;
            } else {
                return false;
            }
        }
    }
    public int getValue() {
        return value.get();
    }
}
```

Which of the following propositions are true? You can select multiple options.

a) This implementation does not work as specified without making the `decrement()` and `getValue()` methods `synchronized` (or adding `synchronized` blocks around their bodies, using the same monitor).

b) Removing the `while(true)` loop from the body of `decrement()` (and fixing the resulting compilation error) would result in invocations to `decrement()` potentially not changing the counter even when it is greater than 0.

c) This works because the internal implementation of `AtomicInteger` uses `Lock`s internally.

d) There is a data race in the invocation to method `compareAndSet()`.

e) The value of the counter may change between the invocation of `value.get()` and the invocation of `compareAndSet()`.

b, e

Option (b) is correct. Removing the loop would lead to situations where the current value of the counter changes between the invocations to `value.get()` and `compareAndSet()`. In this scenario, the *compare* part of the the latter would fail and the counter would not be updated, even if its value was still above 0. Precisely because of this

scenario, option (e) is also correct and this does not affect the correctness of the implementation; the loop ensures that `decrement` will try again until either it succeeds or the value of the counter reaches 0.

This implementation is correct. `AtomicInteger` does not use locks or monitors internally, so option (c) is wrong. It used to in the past, but does not anymore and the class still behaves correctly. Option (d) is also wrong because this method ensures (with assistance from the hardware's underlying instructions) that the value of variable `value` is read and updated **atomically**. In other words, the situation where one thread reads from the variable while the other updates it is not possible and therefore, by definition, data races cannot occur. Option (a) is wrong because no additional synchronization is required to use `AtomicInteger`. It is thread-safe but, since it is a non-blocking data structure, it behaves differently from a blocking ones data structure.

There are many other utility classes in the `java.util.concurrent` package.