

# Advanced Multithreading

Assignment 9 – Theoretical Questions

Advanced Programming (AP)

---

**Faraz Ardeh**

fa.ardeh@gmail.com

Shahid Beheshti University

June 2026

## Contents

|          |                                                                                     |          |
|----------|-------------------------------------------------------------------------------------|----------|
| <b>1</b> | <b>What Are Atomic Variables?</b>                                                   | <b>2</b> |
| 1.1      | How they differ from ordinary variables . . . . .                                   | 2        |
| 1.2      | Key difference summary . . . . .                                                    | 2        |
| <b>2</b> | <b>Classes from <code>java.util.concurrent.atomic</code></b>                        | <b>2</b> |
| 2.1      | Typical use case – <code>AtomicInteger</code> as a shared request counter . . . . . | 2        |
| <b>3</b> | <b>Locks vs. Atomic Variables</b>                                                   | <b>3</b> |
| 3.1      | When a lock is the better choice . . . . .                                          | 3        |
| 3.2      | When an atomic variable is the better choice . . . . .                              | 3        |
| <b>4</b> | <b>Correct but Slow – Scalability Limits Under High Contention</b>                  | <b>4</b> |
| 4.1      | Why this happens . . . . .                                                          | 4        |
| 4.2      | Three concurrency factors that limit scalability . . . . .                          | 4        |
| <b>5</b> | <b>Why More Threads Do Not Always Improve Performance</b>                           | <b>4</b> |
| 5.1      | Context switching . . . . .                                                         | 5        |
| 5.2      | Lock contention . . . . .                                                           | 5        |
| 5.3      | Cache coherence . . . . .                                                           | 5        |
| 5.4      | Synchronization overhead . . . . .                                                  | 5        |
| 5.5      | Summary . . . . .                                                                   | 5        |
| <b>6</b> | <b>Why Deadlocks Appear in Production but Not During Testing</b>                    | <b>5</b> |
| 6.1      | A thread-scheduling perspective . . . . .                                           | 5        |
| 6.2      | Two strategies to expose deadlocks during testing . . . . .                         | 6        |
|          | <b>Bonus: <code>AtomicInteger</code> vs. Plain <code>int</code> Race Condition</b>  | <b>6</b> |

## Question 1. What Are Atomic Variables?

An **atomic variable** is a variable whose read-modify-write operations are executed as a single, indivisible step – no other thread can observe an intermediate state during the operation.

### How they differ from ordinary variables

With a normal variable, the innocent-looking increment `i++` is actually *three* separate CPU instructions:

1. **Read** the current value of `i` into a register.
2. **Add 1** to that register value.
3. **Write** the result back to `i`.

If two threads execute `i++` concurrently, both may read the same value, both add 1, and both write back the same result – one increment is silently lost. This is a *race condition*.

An atomic variable uses a hardware-level instruction called **Compare-And-Swap (CAS)**, which reads, compares, and writes in a single CPU operation that cannot be interrupted. If another thread changed the value in between, the CAS fails and the operation retries – but no update is ever lost.

### Key difference summary

| Property                                     | Ordinary variable | Atomic variable |
|----------------------------------------------|-------------------|-----------------|
| Thread-safe reads/writes                     | No                | Yes             |
| Compound operations (e.g. <code>i++</code> ) | Not atomic        | Atomic (CAS)    |
| Requires explicit lock                       | Yes               | No              |
| Performance under low contention             | Faster            | Comparable      |

## Question 2. Classes from `java.util.concurrent.atomic`

The `java.util.concurrent.atomic` package provides lock-free, thread-safe operations for a range of data types. Four commonly used classes are listed below.

| Class                                 | What it wraps                                    |
|---------------------------------------|--------------------------------------------------|
| <code>AtomicInteger</code>            | A single <code>int</code> value                  |
| <code>AtomicLong</code>               | A single <code>long</code> value                 |
| <code>AtomicBoolean</code>            | A single <code>boolean</code> flag               |
| <code>AtomicReference&lt;V&gt;</code> | A reference to any object of type <code>V</code> |

### Typical use case – `AtomicInteger` as a shared request counter

Imagine a web server where many threads handle HTTP requests simultaneously. All of them must increment a shared request counter. Using `AtomicInteger` avoids a `synchronized` block while still guaranteeing that every request is counted exactly once:

```
1 import java.util.concurrent.atomic.AtomicInteger;
2
3 public class RequestCounter {
4     private final AtomicInteger count = new AtomicInteger(0);
5
6     public void handleRequest() {
7         int id = count.incrementAndGet(); // atomic: no lock needed
8         System.out.println("Handling request #" + id);
9     }
10
11     public int getTotal() {
12         return count.get();
13     }
14 }
```

Listing 1: Shared request counter with AtomicInteger

Because `incrementAndGet()` is a single atomic CAS instruction, hundreds of threads can call it simultaneously without any of them ever seeing a stale or lost count.

### Question 3. Locks vs. Atomic Variables

Both locks and atomic variables protect shared data from concurrent access, but they are suited to different scenarios.

#### When a lock is the better choice

- **Multiple variables must stay consistent together.** Atomic variables protect exactly one value at a time. If you need to update an account's `balance` and `lastModified` timestamp as one unit, only a lock can guard both fields atomically.
- **Complex conditions and waiting.** Locks support `Condition` objects (`await/signal`), letting a thread block until some business invariant is satisfied (e.g. "wait until the queue is non-empty"). Atomic variables provide no such mechanism.
- **Long critical sections.** If the guarded work is more than a single read-modify-write (e.g. several method calls or I/O), a lock is the natural fit. CAS-based retry loops become expensive when the critical section is long, because failed retries waste CPU cycles.

#### When an atomic variable is the better choice

- **Single-variable, simple operations.** Counters, flags, and version numbers that are updated with one CAS have lower overhead than acquiring and releasing a lock.
- **High-read, low-write scenarios.** `AtomicReference` allows a single writer to publish a new immutable snapshot while many readers access the current reference without any blocking.
- **Non-blocking algorithms.** Lock-free data structures (e.g. `ConcurrentLinkedQueue`) use CAS to guarantee progress even if one thread is delayed, whereas a lock-based

structure can stall all other threads if the lock holder is descheduled.

| Situation                   | Better choice   | Reason                                                 |
|-----------------------------|-----------------|--------------------------------------------------------|
| Single counter / flag       | Atomic variable | Low overhead, no blocking                              |
| Multiple fields as one unit | Lock            | Atomicity across several variables                     |
| Conditional waiting         | Lock            | <code>Condition.await()</code> / <code>signal()</code> |
| Long critical section       | Lock            | CAS retries too costly                                 |
| Non-blocking read-heavy     | Atomic variable | Readers never block                                    |

## Question 4. Correct but Slow – Scalability Limits Under High Contention

A program can be completely free of race conditions and still perform poorly when many threads compete for the same resources. Correctness and scalability are independent properties.

### Why this happens

When every thread must pass through the same lock, threads queue up waiting even if they could logically run in parallel. Adding more CPU cores does not help – the bottleneck is the shared resource, not raw compute power. This matches **Amdahl's Law**: total speedup is bounded by the fraction of work that *cannot* be parallelised.

### Three concurrency factors that limit scalability

- 1. Lock contention.** A coarse-grained lock (one lock protecting many accounts, for example) serialises all threads even when they are working on completely independent data. The more threads, the longer each one waits. The solution is to use finer-grained locking – one lock per account instead of one global lock – so threads that are not sharing data can proceed concurrently.
- 2. False sharing (cache-line contention).** Modern CPUs cache memory in 64-byte *cache lines*. If two threads update logically unrelated variables that happen to reside on the same cache line, each write by one thread invalidates the other core's cached copy. The hardware constantly transfers the cache line between cores – a hidden performance cost that grows with thread count even though there is no logical data sharing between the threads.
- 3. Memory bandwidth saturation.** With many threads all reading and writing shared state, the memory bus becomes a bottleneck. Once the bus is saturated, adding more threads only increases waiting time – each thread must wait longer for its memory access to be serviced. Throughput plateaus or even decreases despite the increase in CPU utilisation.

## Question 5. Why More Threads Do Not Always Improve Performance

Intuitively, more threads should mean more parallelism and faster execution. In practice, beyond a certain point additional threads hurt rather than help.

## Context switching

When the number of threads exceeds the number of CPU cores, the operating system must *time-slice*: it saves the full register state of one thread and loads the state of another many times per second. Each switch has a non-trivial cost (typically a few microseconds). With hundreds of threads competing for a small number of cores, the CPU can spend a significant fraction of its time doing bookkeeping instead of useful work.

## Lock contention

More threads means more threads competing for the same lock simultaneously. Each additional thread increases the average queue length at every lock, so threads spend proportionally more time *blocked* and less time running. In extreme cases, throughput actually decreases because the contention overhead outweighs any benefit from concurrency.

## Cache coherence

Every CPU core maintains its own L1/L2 cache. When one core writes to a shared variable, all other cores that cached that value must *invalidate* their copy (the MESI protocol). With many threads spread across many cores, this cache-invalidation “chatter” becomes expensive: cores stall waiting for the authoritative copy to arrive from another core’s cache or from main memory.

## Synchronization overhead

Every `lock()/unlock()` pair, every `synchronized` block, and every `volatile` write inserts a *memory fence* instruction. Memory fences force the CPU to flush its store buffers and ensure all prior writes are visible to other cores before proceeding. The more threads, the more synchronisation events per unit of time, and the more time spent on coordination rather than computation.

## Summary

| Factor                   | Effect                                                |
|--------------------------|-------------------------------------------------------|
| <b>Context switching</b> | CPU wastes time saving/restoring thread state         |
| <b>Lock contention</b>   | Threads queue up; throughput flattens or drops        |
| <b>Cache coherence</b>   | Cores stall waiting for invalidated cache lines       |
| <b>Sync overhead</b>     | Memory fences cost cycles per lock/volatile operation |

## Question 6. Why Deadlocks Appear in Production but Not During Testing

### A thread-scheduling perspective

A deadlock requires a very specific simultaneous state: Thread A holds lock 1 and waits for lock 2, while Thread B holds lock 2 and waits for lock 1 – at the *same instant*. This is called a *circular wait*.

In a typical test environment:

- The workload is small (few threads, few iterations).
- The JVM is freshly started with warm-up behaviour.
- The OS scheduler tends to run the same thread continuously for short tasks, so the critical interleaving rarely occurs.

In production:

- There are many more threads running concurrently.
- Higher load means more context switches per second.
- Different hardware (more cores, different cache sizes) produces different scheduling patterns.
- The combination of load, hardware, and OS scheduler eventually *will* hit the exact timing that triggers the circular wait, making it a statistical certainty over hours of operation.

## Two strategies to expose deadlocks during testing

1. **Stress testing with a start-gun latch.** Create many more threads than CPU cores (e.g. 50–100 threads for a 4-core machine), use a `CountDownLatch` to hold them all back, then release them simultaneously with a single `countDown()`. Every thread starts at the same instant, maximising contention and forcing the scheduler to interleave lock acquisitions in ways that almost never happen under light load. A deadlock that requires simultaneous lock attempts becomes statistically likely within seconds. This is exactly the technique used in the provided `BankAccountTransferDeadlockTest`:

```
1 CountDownLatch startGun = new CountDownLatch(1);
2 // ... submit 50 000 tasks that each await(startGun) before
   transferring
3 startGun.countDown(); // all tasks released at once
```

Listing 2: Stress test with a start-gun latch

2. **Inject random delays between lock acquisitions.** Insert a short `Thread.sleep(randomMillis)` between acquiring the first lock and trying to acquire the second. This *widens* the window during which another thread can step in, acquire its own first lock (which is the second lock of the first thread), and then attempt the lock that the first thread holds – the exact condition for a circular wait. Even a 1–5 ms random delay dramatically increases the probability of hitting the deadlock without requiring a huge thread count. This technique is sometimes called *sleep injection* or *timing perturbation*.

## Bonus: `AtomicInteger` vs. Plain `int` Race Condition

The program below creates 100 threads, each incrementing both a plain `int` and an `AtomicInteger` 1 000 times. The expected final value is  $100 \times 1000 = 100\,000$ .

```
1 import java.util.concurrent.atomic.AtomicInteger;
2
3 public class RaceConditionDemo {
4
```

```

5      static int unsafeCounter = 0;
6      static AtomicInteger safeCounter = new AtomicInteger(0);
7
8      public static void main(String[] args) throws
        InterruptedException {
9
10         int threadCount      = 100;
11         int incrementsPerThread = 1000;
12
13         Thread[] threads = new Thread[threadCount];
14
15         for (int i = 0; i < threadCount; i++) {
16             threads[i] = new Thread(() -> {
17                 for (int j = 0; j < incrementsPerThread; j++) {
18                     unsafeCounter++;           // NOT atomic --
19                     // race condition!
20                     safeCounter.incrementAndGet(); // atomic --
21                     // always correct
22                 }
23             });
24         }
25
26         for (Thread t : threads) t.start();
27         for (Thread t : threads) t.join();
28
29         int expected = threadCount * incrementsPerThread;
30
31         System.out.println("Expected : " + expected);
32         System.out.println("Unsafe int: " + unsafeCounter
33             + (unsafeCounter == expected ? " (correct)" : " <--
34             LOST UPDATES!"));
35         System.out.println("AtomicInt : " + safeCounter.get()
36             + (safeCounter.get() == expected ? " (correct)" : "
37             <-- wrong!"));
38     }
39 }

```

Listing 3: RaceConditionDemo.java

### Sample output

```

Expected : 100000
Unsafe int: 94371 <-- LOST UPDATES!
AtomicInt : 100000 (correct)

```

### Why the unsafe counter is wrong

`unsafeCounter++` compiles to three bytecode instructions: `getfield`, `iadd`, `putfield`. Between any two of these instructions the thread scheduler can switch to another thread. If two threads both read the same value (say 5 000), both add 1, and both write back 5 001,

one increment vanishes entirely. Under 100 threads this happens thousands of times, so the final value is typically several thousand less than 100 000.

`AtomicInteger.incrementAndGet()` maps to a single `lock xadd` CPU instruction (on x86) that is guaranteed to be indivisible. No thread can read an intermediate state; every increment is counted exactly once, and the result is always 100 000.