Module 17 of 20

Concurrency

Threads and interruption, races and synchronized, the memory model and volatile, executors and Futures, locks, atomics and ConcurrentHashMap, CompletableFuture, latches, barriers and blocking queues.

Concurrency is where Java programs go wrong in ways that pass every test — a lost increment once in a million runs, a loop that never sees the flag flip, a pool that quietly stops accepting work. It is also the part of the language interviewers use to separate people who have shipped a server from people who have read about one.

This module builds from the bottom. Threads first: start versus run, join, the interrupt protocol and daemons. Then the race — why count++ is three operations — and synchronized as the fix, with lock choice, granularity and deadlock ordering. The Java Memory Model gets its own lesson, because visibility is the half of concurrency nobody is taught: happens-before, volatile, safe publication, double-checked locking done right. Executors and Futures replace hand-made threads with a lifecycle; ReentrantLock, the atomics and ConcurrentHashMap are the sharper tools; and CompletableFuture with the latch, barrier, semaphore and blocking queue is how modern code composes asynchronous work.

Every exercise runs real threads on the judge and must print the same answer every time — which is the discipline the module teaches: start everything, join everything, and only then read the results.

CONTENTS

1	Threads and Runnables — starting, joining, interrupting	14 min
2	Shared state, races and synchronized	15 min
3	Visibility, volatile and the Java Memory Model	15 min
4	Executors, thread pools and Futures	14 min
5	Locks, atomics and the concurrent collections	15 min
6	CompletableFuture and the coordination tools	15 min
✓	Checkpoint — Concurrency	30 min

Exercises in this module compile on OpenJDK 18. Reference solutions and the answer key follow each lesson.

LESSON 1 · 14 MIN

Threads and Runnables — starting, joining, interrupting

A thread is an independent path of execution with its own stack, sharing the heap with every other thread in the process. Java had threads in the language from day one — `Thread`, `Runnable`, `synchronized` — which is why so much of the platform, from servlet containers to the collections framework, assumes them. Everything higher-level (executors, futures, parallel streams) is built on what this lesson covers: creating a thread, giving it work, waiting for it, and stopping it politely.

Creating a thread: `Runnable`, not a subclass

```
Runnable work = () → System.out.println("hello from " + Thread.currentThread(
Thread t = new Thread(work, "worker-1");
t.start();          // schedules the thread; run() executes on it, some time l
t.join();           // waits for it to finish
```

`Runnable` is a functional interface with one method, `void run()`, so a lambda fits. Prefer passing a `Runnable` to a `Thread` over extending `Thread`: the task (what to do) and the thread (how to run it) are different concerns, a `Runnable` can be handed to an executor later, and your class keeps its inheritance slot. `Callable<V>` is the sibling that returns a value and may throw checked exceptions; plain threads cannot run one — that is what executors are for (lesson 4).

`start()` versus `run()`: `run()` is an ordinary method call on the *current* thread — nothing concurrent happens. Only `start()` creates the OS thread and calls `run()` on it. Calling `start()` twice throws `IllegalThreadStateException`. This is the first trick question of every concurrency interview.

Joining: waiting for a result

A started thread runs whenever the scheduler gives it CPU; `main` does not wait for it unless told to. `join()` blocks the caller until the target thread terminates; `join(millis)` gives up after a timeout. The standard pattern — start every worker, then join every worker, then read

the results — is what makes a multi-threaded program's output *deterministic*: nothing is printed until all the work is done, so scheduling order does not matter.

```
int[] partial = new int[k];
Thread[] workers = new Thread[k];
for (int i = 0; i < k; i++) {
    final int slot = i;
    workers[i] = new Thread(() → partial[slot] = sumOfSlice(data, slot, k));
    workers[i].start();
}
for (Thread w : workers) w.join();     // after this line every partial[i] is
int total = Arrays.stream(partial).sum();
```

`join()` also establishes *visibility*: every write the worker made before finishing is guaranteed visible to the joiner (lesson 3 explains why — `join` is a happens-before edge). Without a join or another synchronisation action, reading `partial` from `main` would be a data race.

Sleeping, yielding and the interrupt protocol

`Thread.sleep(ms)` parks the current thread for at least that long; `Thread.yield()` hints the scheduler to run someone else (rarely useful). Both are static and act on the *current* thread — `t.sleep()` does not put `t` to sleep, a second classic trick.

There is no safe way to kill a thread. `Thread.stop()` is deprecated-for-removal because it releases every lock the thread held mid-operation, leaving shared objects half-updated. Instead threads are asked to stop through **interruption**: `t.interrupt()` sets the target's interrupt flag; a thread blocked in `sleep`, `wait`, `join` or a blocking queue wakes up with `InterruptedException` (and the flag cleared); a running thread must check `Thread.currentThread().isInterrupted()` itself.

```
Thread worker = new Thread(() -> {
    try {
        while (!Thread.currentThread().isInterrupted()) {
            doOneUnitOfWork();
            Thread.sleep(100);
        }
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();     // restore the flag for whoeve
    }
    cleanUp();
});
```

Two rules that separate correct code from folklore: **never swallow** `InterruptedException` with an empty catch — either propagate it or re-set the flag so the request is not lost; and design loops around the flag so a stop request is honoured within one iteration. Interruption is cooperative; a thread that ignores it cannot be stopped.

Daemon threads and the end of the program

The JVM exits when the last **non-daemon** thread finishes. A daemon thread (`t.setDaemon(true)` before `start()`) does not keep the process alive — right for background housekeeping (cache eviction, metrics flushing), wrong for anything whose work must complete, because it is killed mid-run at exit with no `finally` blocks executed. The garbage collector's threads are daemons; your `main` thread is not.

Thread states

A thread is `NEW` before `start()`, `RUNNABLE` when it is running or eligible to run, `BLOCKED` when waiting to enter a `synchronized` block held by another thread, `WAITING` / `TIMED_WAITING` when parked in `wait`, `join`, `sleep` or a lock's condition, and `TERMINATED` after `run()` returns. `t.getState()` reports it; a thread dump (`jstack <pid>` or `Ctrl-Break`) lists every thread's state and stack, and is the first thing to take when a program hangs — two threads `BLOCKED` on each other's locks is a deadlock in one screen.

Names, priorities, uncaught exceptions

Name threads (`new Thread(task, "payment-worker-3")`): the name appears in every stack trace and thread dump, and "Thread-17" tells you nothing at three in the morning. Priorities (`setPriority`, 1–10) are hints the OS may ignore — never rely on them for correctness. An exception that escapes `run()` kills that thread only, printing a stack trace via the default `UncaughtExceptionHandler`; set one (`t.setUncaughtExceptionHandler(...)`) to log properly, because a silently dying worker is the second-hardest bug to find after a race.

Cost, and why pools exist

A platform thread is an OS thread: about a megabyte of stack reserved, a system call to create, a context switch to run. A few hundred are fine; tens of thousands are not, and `new Thread` per request is how servers fall over under load (`OutOfMemoryError: unable to create native thread`). Reuse them through a pool — lesson 4 — and, since Java 21, consider **virtual threads** (module 19) for I/O-heavy work where a million blocked threads is the natural shape.

Interview angle

"`start()` versus `run()`?" `start()` creates a new thread that calls `run()`; calling `run()` directly runs it on the current thread.

"`Runnable` versus `Callable`?" `run()` returns nothing and throws nothing checked; `call()` returns a value and may throw — needs an executor.

"How do you stop a thread?" You cannot force it; interrupt it and have it check the flag / handle `InterruptedException`. `stop()` is unsafe.

"What does `join()` guarantee?" The target has terminated and all its writes are visible to the caller.

"Daemon thread?" Does not keep the JVM alive; killed at exit without cleanup.

Key takeaways

Task = `Runnable` / `Callable`; thread = `Thread`. Pass the task in; do not extend `Thread`.

`start()` once; `run()` is just a method. Start all, join all, then read results.

Stop cooperatively with `interrupt()`; never swallow `InterruptedException`; `Thread.stop` is gone for good reason.

Daemons die with the JVM; name your threads; set an uncaught-exception handler.

Platform threads are expensive — pool them, or go virtual for blocking I/O.

EXERCISE 1

Split the sum

Read an integer `n`, `n` integers, and a thread count `k`. Divide the array into `k` consecutive slices of `ceil(n / k)` elements (the last slices may be short or empty). Start `k` threads, thread `i` summing slice `i` into `partial[i]`. **Join every thread**, then print `partial=<Arrays.toString(partial)>` and `total=<sum of partials>`.

Example: `6` then `1 2 3 4 5 6` then `3` →

```
partial=[3, 7, 11]
total=21
```

INPUT 1

```
6
1 2 3 4 5 6
3
```

OUTPUT 1

```
partial=[3, 7, 11]
total=21
```

INPUT 2

```
5
10 20 30 40 50
2
```

OUTPUT 2

```
partial=[60, 90]
total=150
```

REFERENCE SOLUTION

```java
import java.util.*;

public class Main {
    public static void main(String[] args) throws InterruptedException {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        int[] a = new int[n];
        for (int i = 0; i < n; i++) a[i] = in.nextInt();
        int k = in.nextInt();
        int chunk = (n + k - 1) / k;
        int[] partial = new int[k];
        Thread[] workers = new Thread[k];
        for (int i = 0; i < k; i++) {
            final int slot = i;
            workers[i] = new Thread(() -> {
                int from = Math.min(n, slot * chunk), to = Math.min(n, (slot + 1) *
                int s = 0;
                for (int j = from; j < to; j++) s += a[j];
                partial[slot] = s;
            }, "worker-" + i);
            workers[i].start();
        }
        for (Thread w : workers) w.join();
        System.out.println("partial=" + Arrays.toString(partial));
        System.out.println("total=" + Arrays.stream(partial).sum());
    }
}
```

EXERCISE 2

Workers by name

Read an integer `n` and `n` integers, then an integer `k` and `k` target values. Create `k` threads named `worker-1` … `worker-k`; thread `i` counts how many array elements equal target `i` and records both the count and **its own thread name** (`Thread.currentThread().getName()`). Start all, then join them in order, printing `<name> found <count> of <target>` after each join. Finally print `state=<the state of worker-1 after its join>`.

Example: `5` then `1 2 2 3 2`, then `2` then `2 3` →

```
worker-1 found 3 of 2
worker-2 found 1 of 3
state=TERMINATED
```

INPUT 1

```
5
1 2 2 3 2
2
2 3
```

OUTPUT 1

```
worker-1 found 3 of 2
worker-2 found 1 of 3
state=TERMINATED
```

INPUT 2

```
3
7 7 7
1
8
```

OUTPUT 2

```
worker-1 found 0 of 8
state=TERMINATED
```

REFERENCE SOLUTION

```
import java.util.*;

public class Main {
    public static void main(String[] args) throws InterruptedException {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        int[] a = new int[n];
        for (int i = 0; i < n; i++) a[i] = in.nextInt();
        int k = in.nextInt();
        int[] targets = new int[k];
        for (int i = 0; i < k; i++) targets[i] = in.nextInt();
        int[] counts = new int[k];
        String[] names = new String[k];
        Thread[] workers = new Thread[k];
        for (int i = 0; i < k; i++) {
            final int slot = i;
            workers[i] = new Thread(() -> {
                int c = 0;
                for (int x : a) if (x == targets[slot]) c++;
                counts[slot] = c;
                names[slot] = Thread.currentThread().getName();
            }, "worker-" + (i + 1));
            workers[i].start();
        }
        for (int i = 0; i < k; i++) {
            workers[i].join();
            System.out.println(names[i] + " found " + counts[i] + " of " + targets[i
        }
        System.out.println("state=" + workers[0].getState());
    }
}
```

QUIZ

1. Predict the behaviour:

```
Thread t = new Thread(() → System.out.println(Thread.currentThread().getName()
t.run();
```

A. Prints `Thread-0` on a new thread

B. Prints `main` — `run()` is an ordinary call on the current thread; only `start()` makes a thread

C. Throws `IllegalThreadStateException`

D. Prints nothing

2. Calling `start()` twice on the same `Thread` …

A. Runs it twice

B. Throws `IllegalThreadStateException`

C. Is ignored the second time

D. Creates two threads

3. The correct way to stop a running thread is…

A. `t.stop()`

B. `t.interrupt()`, with the thread checking its flag or handling `InterruptedException`

C. `t.destroy()`

D. Setting a plain `boolean` field

4. After `t.join()` returns in `main`…

A. `t` may still be running

B. `t` has terminated and all its writes are visible to `main`

C. `t` is paused

D. Nothing is guaranteed about `t`'s writes

5. A daemon thread…

A. Has the highest priority

B. Does not keep the JVM alive and is killed at exit without running `finally` blocks

C. Cannot be interrupted

D. Runs only at start-up

ANSWER KEY · THREADS AND RUNNABLES — STARTING, JOINING, INTERRUPTING

1. B The most common concurrency trick question. `start()` creates the OS thread and calls `run()` on it.

2. B A thread runs once; create a new `Thread` (or use an executor) for another run.

3. B `stop()` is unsafe and removed; a plain boolean may never be seen (memory model). Interruption is the cooperative protocol.

4. B `join` is a happens-before edge — that is what makes start-all/join-all/read deterministic.

5. B Right for housekeeping; wrong for work that must complete.

LESSON 2 · 15 MIN

Shared state, races and synchronized

Two threads and one variable is all it takes. `count++` looks atomic and is three operations — read, add, write — so two threads incrementing a shared counter a million times each end up with something less than two million, a different something each run. That is a **race condition**, and everything in this module is a way to avoid one. The oldest and still most common tool is `synchronized`: a block that only one thread at a time may execute, guarded by a lock that every object carries. This lesson is what a race is, what `synchronized` guarantees, how to use it without deadlocking, and the `wait` / `notify` protocol that lives on the same locks.

The anatomy of a race

```
class Counter { int count; void increment() { count++; } }
// two threads, 100 000 increments each → 200 000 expected; observed: 137 412,
```

Thread A reads `count` (5), thread B reads `count` (5), A writes 6, B writes 6: one increment is lost. The window is nanoseconds wide, which is why the bug appears only under load, only in production, only on Tuesdays. Any **check-then-act** (`if (!map.containsKey(k)) map.put(k, v)`) or **read-modify-write** (`count++`, `balance -= amount`, `list.add` on an `ArrayList`) on shared mutable state without synchronisation is a race. The fix is always one of: do not share, do not mutate, or coordinate access — and `synchronized` is coordination.

`synchronized`: mutual exclusion on an intrinsic lock

Every Java object has an **intrinsic lock** (monitor). A `synchronized` block acquires the lock of the object named, runs, and releases it — even if the body throws. While one thread holds it, any other thread reaching a block synchronised on the *same object* blocks.

```
class Counter {
    private int count;
    public synchronized void increment() { count++; }          // locks `this
    public synchronized int get() { return count; }             // also locks
}

class Registry {
    private final Object lock = new Object();                    // a private
    private final Map<String, Integer> map = new HashMap<>();
    public void put(String k, int v) {
        synchronized (lock) { map.put(k, v); }
    }
}
```

A `synchronized` instance method locks `this`; a `synchronized static` method locks the `Class` object; a `synchronized (obj)` block locks whatever you name. **Both writers and readers** must synchronise on the same lock: a `get()` without the lock may see a stale or half-written value (lesson 3), so "only the writes need synchronising" is a bug.

Two guarantees come with the lock: **atomicity** (the block runs as one indivisible step relative to other blocks on the same lock) and **visibility** (everything written before a release is visible to whoever acquires the lock next). The lock is **reentrant**: a thread that holds it may acquire it again (a synchronised method calling another synchronised method on the same object) without deadlocking itself.

Choosing the lock

Prefer a **private final lock object** over `this` for classes others can see: anyone with a reference to your object can `synchronized (yourObject)` and interfere with, or deadlock, your internal locking. Never lock on a `String` literal (interned, shared with the whole JVM), a boxed primitive (cached), or any value-based class — you would be sharing a lock with strangers. Lock on one object per independent piece of state; lock on the same object for state that must change together.

Granularity: how much to lock

```
// too coarse: the network call holds the lock for milliseconds
synchronized (lock) { data = fetchFromNetwork(); cache.put(k, data); }
// right: compute outside, publish inside
Data d = fetchFromNetwork();
synchronized (lock) { cache.put(k, d); }
```

Hold locks for as short a time as possible and never while doing I/O, sleeping, or calling out to code you do not control (a listener might try to acquire another lock). A lock held for a long time serialises the whole program onto one thread; a lock held for a few instructions costs almost nothing when uncontended — modern JVMs take an uncontended `synchronized` in a few nanoseconds.

Deadlock, and the one rule that prevents it

```
// thread 1: synchronized (a) { synchronized (b) { … } }
// thread 2: synchronized (b) { synchronized (a) { … } }   → each holds one, w
```

Deadlock needs two or more locks acquired in different orders. The rule: **acquire locks in a global, consistent order** — sort the objects by some key (account id, `System.identityHashCode` as a last resort) and always lock the smaller first. Or use one lock for both. Or use `tryLock` with a timeout (lesson 5). A thread dump shows a deadlock as two threads `BLOCKED`, each "waiting to lock" what the other "locked"; `jstack` even prints "Found one Java-level deadlock".

`wait` / `notify`: waiting for a condition

Mutual exclusion says "not at the same time"; sometimes a thread needs "not until something is true" — a consumer waiting for a non-empty queue. `Object.wait()` releases the lock and parks the thread; `notify()` / `notifyAll()` wakes waiters, who re-acquire the lock and continue. All three must be called while holding the monitor, or you get `IllegalMonitorStateException`.

```
synchronized (queue) {
    while (queue.isEmpty()) queue.wait();      // ALWAYS in a loop: spurious wa
    item = queue.remove();
}
synchronized (queue) { queue.add(x); queue.notifyAll(); }
```

Always `while`, never `if`: a thread can wake spuriously, or another consumer may have taken the item first. Prefer `notifyAll` unless you can prove one waiter suffices. And in practice prefer `java.util.concurrent` — a `BlockingQueue`, a `CountDownLatch`, a `Condition` — over hand-written `wait` / `notify`; the library versions are the same protocol without the mistakes.

What `synchronized` does not do

It does not make a *sequence* of synchronised calls atomic: `if (map.get(k) == null) map.put(k, v)` on a `Collections.synchronizedMap` is still a check-then-act race across two atomic calls — you need one block around both, or `putIfAbsent`. It does not help if some access bypasses it. It does not scale to many cores contending on one hot lock — for a counter, `AtomicInteger` or `LongAdder` (lesson 5) is faster. And it cannot be interrupted or time out — a thread waiting for a monitor waits until it gets it.

Interview angle

"What is a race condition?" Unsynchronised access to shared mutable state where the result depends on timing — `count++` from two threads.

"What does `synchronized` guarantee?" Mutual exclusion on the monitor and visibility of writes at release/acquire; it is reentrant.

"Synchronised method versus block?" Method locks `this` (or the class for static); a block names the lock and can be narrower.

"How do you avoid deadlock?" Consistent lock ordering, one lock, or `tryLock` with timeout; never hold a lock while calling unknown code.

"Why `while` around `wait()`?" Spurious wake-ups and competing waiters; the condition must be rechecked.

Key takeaways

`count++` is three steps; every read-modify-write or check-then-act on shared state is a race.

`synchronized` = one thread at a time on a monitor + visibility on release/acquire; readers lock too; reentrant.

Lock a private object, briefly, never around I/O or foreign calls; one lock per unit of related state.

Deadlock = inconsistent lock order; fix with ordering, a single lock, or `tryLock`.

`wait` in a `while`, `notifyAll` unless proven otherwise — and reach for `java.util.concurrent` first.

EXERCISE 1

The counter, fixed

Read `k` (threads) and `m` (increments per thread). Share one `Counter` whose `increment()` and `get()` are both `synchronized`. Start `k` threads that each call `increment()` `m` times, join them, and print `expected=<k×m> actual=<counter> match=<expected == actual>`. Without the `synchronized`, `actual` would fall short on most runs; with it the program is correct every time.

Example: `4 1000` →

```
expected=4000 actual=4000 match=true
```

INPUT 1

```
4 1000
```

OUTPUT 1

```
expected=4000 actual=4000 match=true
```

INPUT 2

```
1 5
```

OUTPUT 2

```
expected=5 actual=5 match=true
```

REFERENCE SOLUTION

```java
import java.util.*;

class Counter {
    private int count;
    synchronized void increment() { count++; }
    synchronized int get() { return count; }
}

public class Main {
    public static void main(String[] args) throws InterruptedException {
        Scanner in = new Scanner(System.in);
        int k = in.nextInt(), m = in.nextInt();
        Counter counter = new Counter();
        Thread[] workers = new Thread[k];
        for (int i = 0; i < k; i++) {
            workers[i] = new Thread(() -> { for (int j = 0; j < m; j++) counter.incr
            workers[i].start();
        }
        for (Thread w : workers) w.join();
        int expected = k * m, actual = counter.get();
        System.out.println("expected=" + expected + " actual=" + actual + " match="
    }
}
```

EXERCISE 2

Transfers under one lock

Read an integer `n` and `n` opening balances, then an integer `t` and `t` transfers `from to amount` (0-based account indexes; balances may go negative). Run **each transfer on its own thread**, guarding the balances array with a single lock (`synchronized` on the bank object). Join all threads, then print `balances=<Arrays.toString>` , `total=<sum>` and `preserved=<total equals the opening total>` .

Example: `3` then `100 50 0` , `3` then `0 1 30` , `1 2 60` , `2 0 10` →

```
balances=[80, 20, 50]
total=150
preserved=true
```

INPUT 1

```
3
100 50 0
3
0 1 30
1 2 60
2 0 10
```

OUTPUT 1

```
balances=[80, 20, 50]
total=150
preserved=true
```

INPUT 2

```
2
5 5
1
0 1 10
```

OUTPUT 2

```
balances=[-5, 15]
total=10
preserved=true
```

REFERENCE SOLUTION

```java
import java.util.*;

class Bank {
    final int[] balances;
    Bank(int[] balances) { this.balances = balances; }
    synchronized void transfer(int from, int to, int amount) {
        balances[from] -= amount;
        balances[to] += amount;
    }
    synchronized int total() { return Arrays.stream(balances).sum(); }
}

public class Main {
    public static void main(String[] args) throws InterruptedException {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        int[] opening = new int[n];
        for (int i = 0; i < n; i++) opening[i] = in.nextInt();
        Bank bank = new Bank(opening.clone());
        int t = in.nextInt();
        List<Thread> threads = new ArrayList<>();
        for (int i = 0; i < t; i++) {
            int from = in.nextInt(), to = in.nextInt(), amount = in.nextInt();
            Thread th = new Thread(() -> bank.transfer(from, to, amount));
            threads.add(th);
            th.start();
        }
        for (Thread th : threads) th.join();
        System.out.println("balances=" + Arrays.toString(bank.balances));
        System.out.println("total=" + bank.total());
        System.out.println("preserved=" + (bank.total() == Arrays.stream(opening).su
    }
}
```

QUIZ

1. `count++` on a shared `int` from two threads is a race because…

A. `int` is 64-bit

B. It is read-modify-write: three steps that can interleave

C. The JIT forbids it

D. It is atomic but slow

2. A `synchronized` instance method locks…

A. The class

B. `this`

C. A global lock

D. Nothing — it only prevents reordering

3. Only the write methods of a class are `synchronized`; its `get()` is not. This is…

A. Fine — reads are always safe

B. A bug: the reader may see a stale or partially written value

C. Faster and correct

D. A compile error

4. The one rule that prevents deadlock with several locks is…

A. Use `Thread.sleep` between locks

B. Acquire them in a consistent global order

C. Use more locks

D. Lock on string literals

5. `wait()` must be called…

A. Anywhere

B. While holding the object's monitor, inside a `while` loop rechecking the condition

C. Only from `main`

D. After `notify()`

6. Predict: two threads each run `if (!map.containsKey(k)) map.put(k, 1)` on a `Collections.synchronizedMap`. The result…

A. Is always one insertion

B. May run `put` twice — each call is atomic, the pair is not

C. Throws

D. Deadlocks

ANSWER KEY · SHARED STATE, RACES AND SYNCHRONIZED

1. B Both threads can read 5 and both write 6, losing an increment.

2. B A `static synchronized` method locks the `Class` object; a block names its own lock.

3. B Visibility needs the acquire on the reading side too.

4. B Or use a single lock, or `tryLock` with back-off.

5. B Otherwise `IllegalMonitorStateException`; and spurious wake-ups make the loop mandatory.

6. B Check-then-act across two synchronised calls is still a race; use `putIfAbsent` or one block.

LESSON 3 · 15 MIN

Visibility, volatile and the Java Memory Model

Mutual exclusion is half of concurrency. The other half is **visibility**: when thread A writes a variable, when — if ever — does thread B see the new value? The intuitive answer, "immediately", is wrong. Compilers reorder statements, CPUs keep values in registers and per-core caches, and the JIT hoists a read out of a loop if nothing tells it the value can change. The **Java Memory Model** (JMM) is the contract that says exactly which writes a read is guaranteed to see, and `volatile`, `final`, locks and the other *synchronisation actions* are how you invoke it. This lesson makes that contract usable.

The bug that motivates everything

```
class Worker implements Runnable {
    boolean running = true;                  // no volatile
    public void run() { while (running) { work(); } }
}
// main: worker.running = false;   → the worker may loop forever
```

The JIT sees that `running` is never written inside the loop, hoists the read, and compiles `while (true)`. Or the core running the worker keeps the old value in its cache. The write from `main` is real; it is simply not *required* to be seen. This is not a theoretical hazard — it reproduces reliably in a two-line program with `-server` mode.

Happens-before

The JMM does not talk about time or caches. It defines a partial order, **happens-before** (hb): if action X hb action Y, then Y sees X's effects. The sources of hb edges that you actually use:

Action A	happens-before
each statement in a thread	the next statement in that thread (program order)
unlock of a monitor	every later lock of the same monitor
write to a `volatile` field	every later read of that field
`Thread.start()`	every action in the started thread
every action in a thread	a `join()` on that thread returning
the end of a constructor writing `final` fields	any read of those fields through a properly published reference
`ExecutorService.submit`, `CountDownLatch.countDown`, `BlockingQueue.put`, `Future.get`, …	the corresponding release/acquire on the other side (all documented in `java.util.concurrent`)

hb is transitive. So: main writes `partial[i]`? No — worker writes `partial[i]`, then terminates; `join()` returns in main; therefore main's read sees the write. The pattern *write → release → acquire → read* is the whole theory: get an edge between the writer and the reader and the write is visible; have none and it is a **data race**, whose outcome is undefined (the JMM allows a reader to see a stale value or, for `long` / `double` before Java 17 on 32-bit systems, half of one).

`volatile`: visibility without a lock

A `volatile` field is never cached in a register or hoisted; every read goes to shared memory and sees the latest write, and a write to it hb any subsequent read of it. In the example, `volatile boolean running` fixes the loop.

```
private volatile boolean running = true;          // stop flag: perfect
private volatile Config current;                  // safely publishes a fully b
private volatile int count;  count++;             // WRONG: still read-modify-w
```

`volatile` gives **visibility and ordering**, not **atomicity**. `count++` on a volatile is still a race; use `AtomicInteger`. The rule: `volatile` for a single flag or a reference to an immutable object that one thread writes and others read; a lock or an atomic for anything compound. Also note

that a volatile write publishes everything written *before* it — build the object fully, then assign the volatile reference — which is what makes it a *publication* mechanism, not just a flag.

Safe publication

An object is **safely published** when other threads are guaranteed to see its fully constructed state. Handing a reference across threads through a plain field is not safe: the reader may see the reference before the fields it points to. The safe routes are: a `volatile` or `AtomicReference` field; a `synchronized` block on both sides; a static initialiser (class-init lock); a `final` field of a properly published object; or any `java.util.concurrent` hand-off (queue, latch, executor submission, future). Making the object **immutable** — every field `final`, no leaks of `this` from the constructor — lets it ride any of these safely and removes a whole class of bugs.

Double-checked locking, done right

The classic lazy singleton without `volatile` is broken: a second thread can see `instance ≠ null` before the object's fields are written. With `volatile` on the field it is correct since Java 5:

```
private static volatile Registry instance;
static Registry get() {
    Registry r = instance;                      // one volatile read on the fa
    if (r == null) {
        synchronized (Registry.class) {
            r = instance;
            if (r == null) instance = r = new Registry();
        }
    }
    return r;
}
```

And still prefer the holder idiom from the memory module — simpler, and the JVM does the locking.

`final` fields and immutability

The JMM gives `final` fields a special guarantee: once a constructor finishes, any thread that obtains a reference to the object *without a data race on the reference itself* sees the final fields correctly initialised — even with no synchronisation. This is why immutable objects (`String`, records, `Integer`) can be shared freely, and why "make it `final`" is concurrency advice, not just style. The guarantee does not extend to what the final field *points to* if that is mutable: a `final List` you keep adding to is not immutable.

Reordering: what the compiler may do

Within one thread, the JIT may reorder any two operations that do not depend on each other, as long as *that thread* cannot tell. Other threads can. The hb rules above are the only fences you have: a `volatile` write acts as a release (nothing before it moves after it), a `volatile` read as an acquire (nothing after it moves before it), and monitor exit/enter likewise. `Thread.sleep` and `System.out.println` are **not** synchronisation actions — "it works when I add a print" is the JMM telling you that you have a race.

Atomicity of primitives

Reads and writes of references and 32-bit primitives are atomic (you never see a torn value); `long` and `double` are guaranteed atomic only when `volatile` on older or 32-bit JVMs. Compound operations — `++`, `+=`, check-then-act — are never atomic without a lock or an atomic class.

Interview angle

"What does `volatile` do?" Guarantees visibility and ordering for that field's reads and writes; not atomicity.

"Can `volatile` replace `synchronized`?" For a single flag or a published immutable reference, yes; for compound updates, no.

"What is happens-before?" The JMM's order: if A hb B, B sees A's writes. Edges come from locks, volatiles, `start` / `join`, and the `java.util.concurrent` classes.

"Why do you need `volatile` in double-checked locking?" Without it a thread can see the reference before the object's fields are initialised.

"Why are immutable objects thread-safe?" Final fields are guaranteed visible after construction; nothing changes afterwards, so there is nothing to race on.

Key takeaways

Writes are not automatically visible to other threads; the JIT hoists and CPUs cache. A program that "works with a print" has a race.

Happens-before is the contract: unlock→lock, volatile write→read, start, join, and every `j.u.c` hand-off.

`volatile` = visibility + ordering, no atomicity; right for flags and publishing immutable objects.

Safe publication through volatile, locks, static init, `final`, or `j.u.c`; immutability makes publication trivially safe.

DCL needs `volatile`; the holder idiom needs nothing.

EXERCISE 1

Publish through a latch

Read an integer `k` and `k` words. Start `k` threads; thread `i` writes the upper-cased word into `results[i]` and then calls `countDown()` on a `CountDownLatch(k)`. The main thread calls `await()` — **no joins** — and then prints `results=<Arrays.toString(results)>` and `remaining=<latch.getCount()>`. The latch's happens-before edge is what makes every write visible.

Example: `3` then `ab cd ef` →

```
results=[AB, CD, EF]
remaining=0
```

INPUT 1

```
3
ab cd ef
```

OUTPUT 1

```
results=[AB, CD, EF]
remaining=0
```

INPUT 2

```
1
solo
```

OUTPUT 2

```
results=[SOLO]
remaining=0
```

REFERENCE SOLUTION

```java
import java.util.*;
import java.util.concurrent.*;

public class Main {
    public static void main(String[] args) throws InterruptedException {
        Scanner in = new Scanner(System.in);
        int k = in.nextInt();
        String[] words = new String[k];
        for (int i = 0; i < k; i++) words[i] = in.next();
        String[] results = new String[k];
        CountDownLatch done = new CountDownLatch(k);
        for (int i = 0; i < k; i++) {
            final int slot = i;
            new Thread(() -> {
                results[slot] = words[slot].toUpperCase();
                done.countDown();
            }).start();
        }
        done.await();
        System.out.println("results=" + Arrays.toString(results));
        System.out.println("remaining=" + done.getCount());
    }
}
```

EXERCISE 2

An immutable snapshot

Read an integer `n` and `n` integers into a mutable `ArrayList`. Write a class `Snapshot` with a single `final List<Integer> values` field that **defensively copies** its constructor argument with `List.copyOf`. Build a snapshot, then mutate the source list: append `99` and set index 0 to `-1`. Print `snapshot=<snapshot.values()>` and `source=<the list>`. Then try `snapshot.values().add(1)`; catch the `UnsupportedOperationException` and print `mutable=false` (or `mutable=true` if no exception).

Example: `3` then `5 6 7` →

```
snapshot=[5, 6, 7]
source=[-1, 6, 7, 99]
mutable=false
```

INPUT 1

```
3
5 6 7
```

OUTPUT 1

```
snapshot=[5, 6, 7]
source=[-1, 6, 7, 99]
mutable=false
```

INPUT 2

```
1
0
```

OUTPUT 2

```
snapshot=[0]
source=[-1, 99]
mutable=false
```

REFERENCE SOLUTION

```
import java.util.*;

final class Snapshot {
    private final List<Integer> values;
    Snapshot(List<Integer> values) { this.values = List.copyOf(values); }
    List<Integer> values() { return values; }
}

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        List<Integer> source = new ArrayList<>();
        for (int i = 0; i < n; i++) source.add(in.nextInt());
        Snapshot snap = new Snapshot(source);
        source.add(99);
        source.set(0, -1);
        System.out.println("snapshot=" + snap.values());
        System.out.println("source=" + source);
        try {
            snap.values().add(1);
            System.out.println("mutable=true");
        } catch (UnsupportedOperationException e) {
            System.out.println("mutable=false");
        }
    }
}
```

QUIZ

1. A worker loops on `while (running) {}` where `running` is a plain `boolean`; `main` sets it to `false`. The worker…

A. Stops immediately

B. May never stop — the read can be hoisted or cached; the field needs `volatile`

C. Throws

D. Stops after one second

2. `volatile int count; count++;` from several threads is…

A. Atomic and visible

B. Visible but still a race — `++` is read-modify-write

C. A compile error

D. Slower but correct

3. Which is **not** a happens-before source?

A. Unlocking a monitor another thread later locks

B. A `volatile` write followed by a read of that field

C. `Thread.sleep(100)` between the write and the read

D. `Thread.join()` returning

4. Double-checked locking without `volatile` on the instance field can…

A. Deadlock

B. Return a reference to an object whose fields are not yet visible

C. Create two instances always

D. Not compile

5. Immutable objects are thread-safe because…

A. They live on the stack

B. `final` fields are guaranteed visible after construction and nothing changes afterwards

C. The JVM locks them

D. They are always cached

ANSWER KEY · VISIBILITY, VOLATILE AND THE JAVA MEMORY MODEL

1. B Without a happens-before edge the write need not become visible.

2. B `volatile` gives visibility and ordering, not atomicity; use `AtomicInteger`.

3. C Sleep and print statements are not synchronisation actions; "it works with a sleep" means you have a race.

4. B Reordering may publish the reference before the constructor's writes; `volatile` (or the holder idiom) fixes it.

5. B No mutable state, no race. Make fields `final`, copy inputs defensively, and never leak `this` from the constructor.

LESSON 4 · 14 MIN

Executors, thread pools and Futures

Creating a thread per task does not scale, and juggling `Thread` objects by hand — start, join, collect results, handle the one that threw — is repetitive and easy to get wrong. `java.util.concurrent` separates the *task* from the *thread that runs it*: you submit `Runnable`s and `Callable`s to an **`ExecutorService`**, a pool of reusable threads runs them, and each submission hands back a **`Future`** through which you collect the result or the exception. This lesson covers the pools, the two submission styles, the shutdown protocol everybody forgets, and what a `Future` actually promises.

The executor model

```
ExecutorService pool = Executors.newFixedThreadPool(4);
Future<Integer> f = pool.submit(() → expensiveComputation(42));   // a Callal
// … do other things …
int result = f.get();            // blocks until the task completes; rethrows
pool.shutdown();
```

`execute(Runnable)` fires and forgets; `submit(Callable)` returns a `Future<V>`; `submit(Runnable)` returns a `Future<?>` that completes with `null`. Tasks queue up when every pool thread is busy and run in submission order per queue — but on *different threads*, so nothing about their relative timing is guaranteed.

The factory methods, and when each is wrong

Factory	Threads	Queue	Use	Danger
`newFixedThreadPool(n)`	exactly `n`	unbounded	CPU-bound work, `n ≈ cores`	unbounded queue → memory grows under overload
`newCachedThreadPool()`	as many as needed, reused for 60 s	none (direct hand-off)	many short I/O tasks, bursty	unbounded threads → `unable to create native thread`
`newSingleThreadExecutor()`	1	unbounded	serialise access to a resource; ordered event processing	one slow task blocks everything
`newScheduledThreadPool(n)`	`n`	delayed	periodic jobs, timeouts	an exception kills the periodic task silently
`newWorkStealingPool()`	parallelism = cores	per-thread deques	a `ForkJoinPool` for many small tasks	no ordering; daemon threads
`newVirtualThreadPerTaskExecutor()` (21)	a virtual thread per task	none	blocking I/O at scale	not for CPU-bound work

For production services the honest answer is the constructor underneath: `new ThreadPoolExecutor(core, max, keepAlive, unit, new ArrayBlockingQueue<>(capacity), factory, rejectionHandler)` — bounded queue, named threads, and an explicit policy (`CallerRunsPolicy`, `AbortPolicy`) for what happens when both are full. "Both the queue and the thread count are bounded" is the sentence that shows you have run one.

Sizing: CPU-bound → about the number of cores (`Runtime.getRuntime().availableProcessors()`); I/O-bound → more, roughly `cores × (1 + wait/compute)`, or virtual threads.

Submitting many tasks

```
List<Callable<Integer>> tasks = List.of(() → work(1), () → work(2), () → wo
List<Future<Integer>> futures = pool.invokeAll(tasks);      // blocks until Al
for (Future<Integer> f : futures) System.out.println(f.get());
Integer first = pool.invokeAny(tasks);                      // the first to su
```

`invokeAll` returns the futures **in the order of the task list**, regardless of which finished first — which is how a multi-threaded program prints deterministic output. `invokeAny` is for redundant requests (three replicas, take the fastest).

What a `Future` gives you

`get()` — block until done; returns the value, or throws `ExecutionException` **wrapping** the task's exception (`e.getCause()` is the real one), or `InterruptedException` if *you* were interrupted while waiting, or `CancellationException`.

`get(timeout, unit)` — the same with a deadline; `TimeoutException` if it passes. Prefer this in anything user-facing.

`isDone()`, `isCancelled()`; `cancel(mayInterruptIfRunning)` — removes a queued task, or interrupts a running one if you say so (and only helps if the task honours interruption).

A `Future` is *pull-based*: you ask and wait. It cannot be composed ("when this finishes, do that") — that is `CompletableFuture` (lesson 6). A task that throws does not print anything: the exception sits inside the `Future` until someone calls `get()`. **Uncollected futures hide failures**; either call `get()` or use an executor whose `afterExecute` logs them.

Shutting down, properly

An `ExecutorService` keeps its threads alive; a pool you never shut down keeps the JVM alive (non-daemon threads) and is a resource leak in anything long-running.

```
pool.shutdown();                                        // stop accepting tasks
if (!pool.awaitTermination(30, TimeUnit.SECONDS)) {     // wait for them
    pool.shutdownNow();                                 // interrupt what is st
}
```

`shutdown()` is graceful; `shutdownNow()` interrupts. Since Java 19 `ExecutorService` is `AutoCloseable`, so `try (var pool = Executors.newFixedThreadPool(4)) { … }` does

shutdown-and-await for you. In a scheduled pool, remember that a periodic task that throws is **cancelled silently** — wrap the body in a try/catch that logs.

Exceptions and thread death

In a pool, an exception escaping a `Runnable` given to `execute()` kills the worker thread (the pool replaces it) and reaches the uncaught-exception handler; the same exception from `submit()` is captured in the `Future` and *nothing is printed*. Choose deliberately: `execute` when you want loud failures, `submit` when you will look at the `Future`.

`ThreadLocal` in pools — the classic leak

`ThreadLocal` gives each thread its own value (a per-request context, a non-thread-safe formatter). In a pool the thread outlives the task, so the value survives into the next unrelated task unless you `remove()` it in a `finally`. Always pair `set` with `remove`.

Interview angle

"Why use an executor instead of `new Thread`?" Reuse (thread creation is expensive), bounded concurrency, queuing, results via `Future`, a clean lifecycle.

"Fixed versus cached pool?" Fixed: bounded threads, unbounded queue — for CPU work. Cached: unbounded threads — for short bursty I/O. Production: `ThreadPoolExecutor` with both bounded.

"What does `Future.get()` throw?" `ExecutionException` wrapping the task's failure, `InterruptedException`, `CancellationException`, `TimeoutException` with a deadline.

"`shutdown` versus `shutdownNow`?" Graceful (finish queued tasks) versus interrupt-and-return-the-rest.

"Why is `Callable` needed?" `Runnable` cannot return or throw checked exceptions; `Callable<V>` does both.

Key takeaways

Task in, `Future` out; the pool owns the threads. `submit` for results, `execute` for fire-and-forget.

Know the factories and their unbounded side; in production build a `ThreadPoolExecutor` with a bounded queue and a rejection policy.

`invokeAll` returns futures in task order — deterministic output from parallel work.

`get()` rethrows as `ExecutionException`; never leave futures uncollected; `get` with a timeout in user-facing code.

`shutdown()` + `awaitTermination()` (+ `shutdownNow()`), or try-with-resources on 19+; `ThreadLocal.remove()` in a `finally`.

EXERCISE 1

invokeAll keeps the order

Read a pool size `p`, an integer `n` and `n` integers. Build a `Callable<Integer>` per integer `x` that counts the primes `≤ x`, submit them all with `invokeAll` on a fixed pool of `p` threads, and print `<x> → <count>` in **input order** — `invokeAll` returns the futures in task order however the threads finish. Then `shutdown()`, `awaitTermination`, and print `terminated=<isTerminated()>`.

Example: `2` then `3` then `10 20 1` →

```
10 → 4
20 → 8
1 → 0
terminated=true
```

INPUT 1

```
2
3
10 20 1
```

OUTPUT 1

```
10 → 4
20 → 8
1 → 0
terminated=true
```

INPUT 2

```
1
1
2
```

OUTPUT 2

```
2 → 1
terminated=true
```

REFERENCE SOLUTION

```java
import java.util.*;
import java.util.concurrent.*;

public class Main {
    static int primesUpTo(int x) {
        int c = 0;
        for (int i = 2; i ⩽ x; i++) {
            boolean prime = true;
            for (int d = 2; d * d ⩽ i; d++) if (i % d == 0) { prime = false; break;
            if (prime) c++;
        }
        return c;
    }

    public static void main(String[] args) throws Exception {
        Scanner in = new Scanner(System.in);
        int p = in.nextInt(), n = in.nextInt();
        int[] xs = new int[n];
        for (int i = 0; i < n; i++) xs[i] = in.nextInt();
        ExecutorService pool = Executors.newFixedThreadPool(p);
        List<Callable<Integer>> tasks = new ArrayList<>();
        for (int x : xs) tasks.add(() -> primesUpTo(x));
        List<Future<Integer>> futures = pool.invokeAll(tasks);
        for (int i = 0; i < n; i++) System.out.println(xs[i] + " -> " + futures.get(
        pool.shutdown();
        pool.awaitTermination(10, TimeUnit.SECONDS);
        System.out.println("terminated=" + pool.isTerminated());
    }
}
```

EXERCISE 2

Failures stay in the Future

Read an integer `n` and `n` integers. On a fixed pool of 2 threads, `submit` a `Callable` per value that returns `100 / x`. Collect each `Future` in order: print `task <i>: <result>` on success, or `task <i>: failed <simple class name of the cause>` when `get()` throws `ExecutionException` (use `getCause()`). Then print `completed=<n ok> failed=<n failed>` and shut the pool down.

Example: `3` then `4 0 25` →

```
task 0: 25
task 1: failed ArithmeticException
task 2: 4
completed=2 failed=1
```

INPUT 1

```
3
4 0 25
```

OUTPUT 1

```
task 0: 25
task 1: failed ArithmeticException
task 2: 4
completed=2 failed=1
```

INPUT 2

```
2
0 0
```

OUTPUT 2

```
task 0: failed ArithmeticException
task 1: failed ArithmeticException
completed=0 failed=2
```

REFERENCE SOLUTION

```
import java.util.*;
import java.util.concurrent.*;

public class Main {
    public static void main(String[] args) throws InterruptedException {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        int[] xs = new int[n];
        for (int i = 0; i < n; i++) xs[i] = in.nextInt();
        ExecutorService pool = Executors.newFixedThreadPool(2);
        List<Future<Integer>> futures = new ArrayList<>();
        for (int x : xs) futures.add(pool.submit(() -> 100 / x));
        int ok = 0, failed = 0;
        for (int i = 0; i < n; i++) {
            try {
                System.out.println("task " + i + ": " + futures.get(i).get());
                ok++;
            } catch (ExecutionException e) {
                System.out.println("task " + i + ": failed " + e.getCause().getClass
                failed++;
            }
        }
        System.out.println("completed=" + ok + " failed=" + failed);
        pool.shutdown();
    }
}
```

QUIZ

1. `Executors.newCachedThreadPool()` is dangerous under load because…

A. It has too few threads

B. It creates an unbounded number of threads

C. It queues forever

D. It cannot run `Callable`s

2. `Future.get()` on a task that threw `IllegalStateException`…

A. Throws `IllegalStateException` directly

B. Throws `ExecutionException` whose cause is the `IllegalStateException`

C. Returns `null`

D. Blocks forever

3. `invokeAll(tasks)` returns…

A. Futures in completion order

B. Futures in the order of the task list, after all have completed

C. The first result

D. A single combined future

4. `shutdown()` versus `shutdownNow()`:

A. Identical

B. `shutdown` stops accepting and lets queued tasks finish; `shutdownNow` interrupts running tasks and returns the ones never started

C. `shutdownNow` waits longer

D. `shutdown` kills threads

5. A `ThreadLocal` set in a pooled thread and never removed…

A. Is cleared automatically after the task

B. Leaks into the next task that runs on that thread

C. Throws

D. Is copied to all threads

6. A periodic task in a `ScheduledExecutorService` throws once. Afterwards…

A. It keeps running

B. It is cancelled silently and never runs again

C. The pool shuts down

D. The exception is printed each period

ANSWER KEY · EXECUTORS, THREAD POOLS AND FUTURES

1. B A burst of tasks becomes a burst of OS threads; `newFixedThreadPool` has the opposite problem — an unbounded queue.

2. B Task exceptions are captured and rewrapped; an uncollected future hides the failure entirely.

3. B Which is what makes multi-threaded output deterministic.

4. B Follow `shutdown()` with `awaitTermination()`; escalate to `shutdownNow()` on timeout.

5. B Pool threads outlive tasks; pair `set` with `remove` in a `finally`.

6. B Wrap the body in try/catch and log; otherwise the job disappears without a trace.

LESSON 5 · 15 MIN

Locks, atomics and the concurrent collections

`synchronized` is a blunt instrument: one lock, no timeout, no fairness, no way to try and back off. `java.util.concurrent` provides sharper ones. `ReentrantLock` is `synchronized` with options; `ReadWriteLock` lets many readers share; the **atomic** classes update a single variable without any lock at all, using the CPU's compare-and-swap; and the **concurrent collections** — `ConcurrentHashMap` above all — bake thread-safety into the data structure so most code never needs an explicit lock. This lesson is the toolbox, and the rule for reaching into it: the highest-level thing that fits.

`ReentrantLock`: `synchronized` with a manual

```
private final ReentrantLock lock = new ReentrantLock();      // or new Reentrar

void transfer(Account from, Account to, int amount) {
    lock.lock();
    try {
        from.debit(amount); to.credit(amount);
    } finally {
        lock.unlock();                                        // ALWAYS in finc
    }
}
```

What it adds over `synchronized`: `tryLock()` (return `false` instead of waiting — the way out of deadlock), `tryLock(timeout, unit)`, `lockInterruptibly()` (a waiting thread can be interrupted), *fairness* (longest-waiting thread gets it next; slower, but no starvation), and **multiple `Condition`s** per lock (`lock.newCondition()`: separate "not empty" and "not full" queues instead of one `notifyAll` waking everyone). What it costs: you must unlock by hand, so the `try/finally` is not optional. When you need none of the extras, `synchronized` is shorter and the JIT optimises it well.

Deadlock avoidance with `tryLock`:

```
while (true) {
    if (a.lock.tryLock()) {
        try { if (b.lock.tryLock()) { try { doBoth(); return; } finally { b.l
        finally { a.lock.unlock(); }
    }
    Thread.onSpinWait();   // back off and retry (add a small random sleep in
}
```

`ReadWriteLock`: many readers, one writer

```
private final ReadWriteLock rw = new ReentrantReadWriteLock();
V get(K k) { rw.readLock().lock(); try { return map.get(k); } finally { rw.rea
void put(K k, V v) { rw.writeLock().lock(); try { map.put(k, v); } finally { r
```

Any number of readers may hold the read lock together; a writer needs the write lock exclusively. It pays when reads greatly outnumber writes and the guarded operation is not trivial; for a simple map, `ConcurrentHashMap` beats it. `StampedLock` (Java 8) is the faster, non-reentrant variant with *optimistic reads* for experts.

Atomics: lock-free single-variable updates

```
AtomicInteger hits = new AtomicInteger();
hits.incrementAndGet();                       // atomic ++, returns the new v
hits.addAndGet(5);
hits.compareAndSet(expected, newValue);       // the primitive everything is
hits.updateAndGet(x -> Math.max(x, candidate));       // any pure function, re
hits.accumulateAndGet(v, Integer::sum);
AtomicReference<Node> head = new AtomicReference<>();  // lock-free stacks and
AtomicLong, AtomicBoolean, AtomicIntegerArray
```

An atomic operation is a **compare-and-swap** loop: read the value, compute the new one, write it only if nothing changed in between, otherwise retry. No lock, no blocking, no deadlock; under heavy contention many threads retry, which is where `LongAdder` comes in — it keeps per-thread cells and sums them on `sum()`, so a hot counter (requests served) scales linearly. Use `AtomicInteger` when you need the current value after each update; `LongAdder` when you increment often and read rarely.

Atomics fix `count++`; they do not fix compound invariants across *two* variables (`if (a.get() > 0) b.incrementAndGet()` is still a race). For that, a lock.

`ConcurrentHashMap`: the workhorse

```
ConcurrentHashMap<String, Integer> counts = new ConcurrentHashMap<>();
counts.merge(word, 1, Integer::sum);                    // atomic per key:
counts.computeIfAbsent(key, k -> expensive(k));         // atomic: expensi
counts.putIfAbsent(k, v);
counts.getOrDefault(k, 0);
long total = counts.reduceValuesToLong(1, Integer::longValue, 0, Long::sum);
```

Internally the table is split so that updates to different buckets do not block each other (Java 8 replaced the old segment locks with per-bin CAS/synchronisation), reads never lock, and iteration is **weakly consistent** — it never throws `ConcurrentModificationException`, reflecting some state between start and end. The compound operations (`merge`, `compute*`, `putIfAbsent`) are atomic per key, which is exactly what makes the check-then-act races of a synchronised `HashMap` disappear. It rejects `null` keys and values, and `size()` is an estimate under concurrent updates. Never do `if (!map.containsKey(k)) map.put(k, v)` on it — the atomic method exists for that.

`Collections.synchronizedMap(new HashMap<>())` is the old alternative: one lock around every call, no atomic compound ops, and iteration requires you to hold the lock yourself. Use it only to wrap something `ConcurrentHashMap` cannot replace (a `TreeMap` → `ConcurrentSkipListMap` exists for that too).

The other concurrent collections

Need	Class	Note
sorted concurrent map/set	`ConcurrentSkipListMap` / `ConcurrentSkipListSet`	lock-free; O(log n)
read-mostly list (listeners)	`CopyOnWriteArrayList`	every write copies the array; iteration is a snapshot, never fails
producer–consumer queue	`ArrayBlockingQueue` (bounded), `LinkedBlockingQueue`, `PriorityBlockingQueue`	`put` blocks when full, `take` when empty
hand-off with no buffer	`SynchronousQueue`	what `newCachedThreadPool` uses
delayed tasks	`DelayQueue`	elements become available at their time
lock-free queue/deque	`ConcurrentLinkedQueue` / `ConcurrentLinkedDeque`	non-blocking; `poll` returns null when empty

`BlockingQueue` is the backbone of the producer–consumer pattern (next lesson): producers `put`, consumers `take`, the queue does all the waiting and waking that `wait` / `notify` used to.

Choosing

Can you avoid sharing? Confine to one thread, or make it immutable.

Is it one variable? An atomic.

Is it a map, list or queue? The concurrent collection.

Is it a compound invariant across several fields? `synchronized`, or `ReentrantLock` if you need `tryLock`, conditions or fairness.

Read-heavy with slow reads? `ReadWriteLock`.

In interviews, saying the order aloud is worth more than knowing every class.

Interview angle

"`ReentrantLock` versus `synchronized`?" Same mutual exclusion; the lock adds `tryLock`, timeouts, interruptible waits, fairness and multiple conditions, at the cost of manual `unlock` in `finally`.

"How does `AtomicInteger` work?" A compare-and-swap loop — no lock; `LongAdder` for hot counters.

"How is `ConcurrentHashMap` different from `Hashtable` / `synchronizedMap`?" Fine-grained per-bin locking, lock-free reads, atomic `merge` / `compute*`, weakly consistent iteration; not one global lock.

"Is `ConcurrentHashMap.size()` exact?" An estimate under concurrent modification.

"When `CopyOnWriteArrayList`?" Many reads, very few writes — listener lists.

Key takeaways

`ReentrantLock` when you need `tryLock`, timeouts, fairness or several `Condition`s; `unlock()` in `finally`, always.

Atomics are CAS loops: `incrementAndGet`, `compareAndSet`, `updateAndGet`; `LongAdder` for contended counters; they do not cover multi-variable invariants.

`ConcurrentHashMap` with `merge` / `computeIfAbsent` / `putIfAbsent` replaces every check-then-act race; no nulls, weakly consistent iteration.

`BlockingQueue` for producer–consumer; `CopyOnWriteArrayList` for read-mostly; `ConcurrentSkipListMap` for sorted.

Highest-level tool that fits: confine → immutable → atomic → concurrent collection → lock.

EXERCISE 1

Concurrent word count

Read an integer `n` and `n` words, then a thread count `k`. Thread `i` handles the words at indexes `i, i + k, i + 2k, …` and counts each with `ConcurrentHashMap.merge(word, 1, Integer::sum)` — atomic per key, no explicit lock. Join all threads, then print `<word>=<count>` for every word in natural order, and `distinct=<map size>`.

Example: `6` then `b a b c a b`, then `2` →

```
a=2
b=3
c=1
distinct=3
```

INPUT 1

```
6
b a b c a b
2
```

OUTPUT 1

```
a=2
b=3
c=1
distinct=3
```

INPUT 2

```
1
zz
3
```

OUTPUT 2

```
zz=1
distinct=1
```

REFERENCE SOLUTION

```
import java.util.*;
import java.util.concurrent.*;

public class Main {
    public static void main(String[] args) throws InterruptedException {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        String[] words = new String[n];
        for (int i = 0; i < n; i++) words[i] = in.next();
        int k = in.nextInt();
        ConcurrentHashMap<String, Integer> counts = new ConcurrentHashMap<>();
        Thread[] workers = new Thread[k];
        for (int t = 0; t < k; t++) {
            final int start = t;
            workers[t] = new Thread(() -> {
                for (int i = start; i < n; i += k) counts.merge(words[i], 1, Integer
            });
            workers[t].start();
        }
        for (Thread w : workers) w.join();
        new TreeMap<>(counts).forEach((w, c) -> System.out.println(w + "=" + c));
        System.out.println("distinct=" + counts.size());
    }
}
```

EXERCISE 2

Atomics, no locks

Read `k` (threads), `m` (rounds), then `k` integers — one value per thread. Every thread performs `m` rounds; in each round it calls `count.incrementAndGet()` on a shared `AtomicInteger`, `sum.add(value)` on a shared `LongAdder`, and `max.accumulateAndGet(value, Math::max)` on a shared `AtomicInteger`. Join all, then print `count=<k×m> sum=<m × Σvalues> max=<largest value>`.

Example: `3 4` then `5 -2 9` →

```
count=12 sum=48 max=9
```

INPUT 1

```
3 4
5 -2 9
```

OUTPUT 1

```
count=12 sum=48 max=9
```

INPUT 2

```
1 1
-7
```

OUTPUT 2

```
count=1 sum=-7 max=-7
```

REFERENCE SOLUTION

```
import java.util.*;
import java.util.concurrent.atomic.*;

public class Main {
    public static void main(String[] args) throws InterruptedException {
        Scanner in = new Scanner(System.in);
        int k = in.nextInt(), m = in.nextInt();
        int[] values = new int[k];
        for (int i = 0; i < k; i++) values[i] = in.nextInt();
        AtomicInteger count = new AtomicInteger();
        LongAdder sum = new LongAdder();
        AtomicInteger max = new AtomicInteger(Integer.MIN_VALUE);
        Thread[] workers = new Thread[k];
        for (int i = 0; i < k; i++) {
            final int v = values[i];
            workers[i] = new Thread(() -> {
                for (int r = 0; r < m; r++) {
                    count.incrementAndGet();
                    sum.add(v);
                    max.accumulateAndGet(v, Math::max);
                }
            });
            workers[i].start();
        }
        for (Thread w : workers) w.join();
        System.out.println("count=" + count.get() + " sum=" + sum.sum() + " max=" +
    }
}
```

QUIZ

1. `ReentrantLock.unlock()` must be called…

A. Only on error

B. In a `finally` block, so an exception cannot leave the lock held

C. Twice

D. Before `lock()`

2. `AtomicInteger.incrementAndGet()` is implemented with…

A. A `synchronized` block

B. A compare-and-swap loop — no lock

C. A spin on `volatile`

D. A `ReentrantLock`

3. `ConcurrentHashMap.computeIfAbsent(k, f)` guarantees…

A. Nothing about `f`

B. `f` runs at most once per key even under contention

C. `f` runs on every call

D. The map is locked globally

4. Iterating a `ConcurrentHashMap` while another thread modifies it…

A. Throws `ConcurrentModificationException`

B. Is weakly consistent: no exception, reflecting some state during the traversal

C. Blocks the writer

D. Returns a copy

5. `CopyOnWriteArrayList` suits…

A. Frequent writes

B. Read-mostly data like listener lists — every write copies the array

C. Sorted access

D. Producer–consumer

6. Which should you reach for **first** when two threads share a counter?

A. `ReentrantReadWriteLock`

B. `AtomicInteger` (or `LongAdder` if hot)

C. `synchronized` on a `String` literal

D. `Thread.sleep`

ANSWER KEY · LOCKS, ATOMICS AND THE CONCURRENT COLLECTIONS

1. B Unlike `synchronized`, nothing releases the lock automatically.

2. B CAS retries if another thread changed the value in between; `LongAdder` reduces that contention for hot counters.

3. B The compound operations are atomic per key — the whole point over a synchronised `HashMap`.

4. B `size()` is likewise an estimate under concurrent updates.

5. B Iteration is a snapshot and never fails; writes are O(n).

6. B Confine → immutable → atomic → concurrent collection → lock, in that order.

LESSON 6 · 15 MIN

CompletableFuture and the coordination tools

A `Future` lets you wait; it does not let you say *what happens next*. `CompletableFuture` does: it is a future you can complete by hand, chain transformations onto, combine with other futures, and recover when it fails — asynchronous code that reads like a stream pipeline. Around it sit the **synchronisers** that coordinate threads without a lock on shared data: `CountDownLatch` (wait until *n* things have happened), `CyclicBarrier` (everyone waits for everyone, repeatedly), `Semaphore` (at most *n* at once), and the `BlockingQueue` that makes producer–consumer a one-liner. This lesson covers all of them with the one question that matters for each: *what thread does the work?*

`CompletableFuture`: a future with `then`

```
CompletableFuture<Integer> price = CompletableFuture.supplyAsync(() -> fetchPr
CompletableFuture<Integer> withTax = price.thenApply(p -> p * 118 / 100);
CompletableFuture<Void> shown = withTax.thenAccept(System.out::println);
CompletableFuture<Integer> total = withTax.thenCombine(fetchShipping(), Intege
CompletableFuture<Order> order = total.thenCompose(t -> placeOrderAsync(t));
Integer value = total.join();
```

`supplyAsync(Supplier)` / `runAsync(Runnable)` start work on the **common ForkJoinPool** unless you pass an executor — always pass one for blocking I/O, or you starve parallel streams and everyone else.

`thenApply` = `map`; `thenCompose` = `flatMap` (the function returns a future — avoid `CompletableFuture<CompletableFuture<T>>`); `thenCombine` joins two; `thenAccept` / `thenRun` end the chain with a side effect.

Each `thenX` runs **on whatever thread completed the previous stage** (or immediately on the caller if it was already complete). The `thenXAsync` variants re-dispatch to a pool. This matters when the previous stage was your UI or event-loop thread.

`join()` throws `CompletionException` (unchecked) wrapping the cause; `get()` throws the checked `ExecutionException`.

Failure and recovery

```
CompletableFuture<Integer> safe = supplyAsync(() → parse(text))
    .exceptionally(ex → 0)                                    // recover with
    .thenApply(x → x * 2);

CompletableFuture<String> reported = supplyAsync(() → risky())
    .handle((value, ex) → ex == null ? "ok " + value : "failed: " + ex.getCau
    .whenComplete((v, ex) → log(v, ex));                       // observe witl
```

An exception in any stage skips every following `thenX` and lands in the next `exceptionally` / `handle` / `whenComplete` — exactly like a thrown exception skipping statements to a `catch`. The cause is wrapped in `CompletionException`; unwrap with `getCause()`. Since Java 12, `exceptionallyCompose` recovers with another future, and `orTimeout(1, SECONDS)` / `completeOnTimeout(default, …)` (Java 9) give you deadlines without a watchdog thread.

Waiting for many

```
List<CompletableFuture<Integer>> parts = ids.stream().map(id → supplyAsync(()
CompletableFuture<Void> all = CompletableFuture.allOf(parts.toArray(Completabl
List<Integer> results = all.thenApply(v → parts.stream().map(CompletableFutur
Object first = CompletableFuture.anyOf(a, b, c).join();      // the first to c
```

`allOf` returns `CompletableFuture<Void>` — it tells you *when*, not *what*; collect the values from the original futures afterwards. Failure of any part fails `allOf`.

You can also complete one by hand: `new CompletableFuture<T>()` then `complete(value)` / `completeExceptionally(ex)` from a callback — the bridge from callback-style APIs into the futures world.

`CountDownLatch`: wait for *n* events

```
CountDownLatch ready = new CountDownLatch(workers);
for (int i = 0; i < workers; i++) pool.execute(() → { prepare(); ready.countl
ready.await();          // blocks until the count reaches zero; then everythir
```

One-shot: the count never resets. Uses: wait for *n* threads to finish or to reach a starting line (a latch of 1 that `main` counts down to start everyone at once — the fairest way to begin a

benchmark). `await(timeout, unit)` returns `false` on timeout instead of waiting forever.

`CyclicBarrier`: everyone waits for everyone, again and again

```
CyclicBarrier round = new CyclicBarrier(k, () → System.out.println("round do
// each worker, per round:  doMyPart(); round.await();
```

`await()` blocks until `k` threads have called it, runs the optional barrier action on the last arriving thread, releases all, and **resets** for the next round — the natural shape of a simulation with phases. A thread that leaves breaks the barrier for everyone (`BrokenBarrierException`). `Phaser` is the flexible version with a variable number of parties.

`Semaphore`: at most *n* at a time

```
Semaphore permits = new Semaphore(10);
permits.acquire();  try { callRateLimitedApi(); } finally { permits.release();
```

A counting semaphore limits concurrency to a resource: connection slots, API rate, a bounded pool of hand-made objects. `tryAcquire()` for non-blocking, a fair constructor for FIFO. Note `release()` does not check that the caller ever acquired — a semaphore is a counter, not a lock.

Producer–consumer with a `BlockingQueue`

```
BlockingQueue<Integer> queue = new ArrayBlockingQueue<>(100);
Thread producer = new Thread(() → { for (int i = 1; i ⩽ n; i++) queue.put(i)
Thread consumer = new Thread(() → { for (int x; (x = queue.take()) ≠ POISON
```

`put` blocks when the queue is full (back-pressure: a slow consumer throttles the producer), `take` blocks when empty. The **poison pill** — a sentinel value the consumer recognises — is the standard way to say "no more"; with several consumers, send one pill per consumer. `offer(x, timeout, unit)` and `poll(timeout, unit)` give up instead of blocking forever. Every hand-off through the queue is a happens-before edge, so whatever the producer built before `put` is visible to the consumer after `take` with no other synchronisation.

Fork/join

`ForkJoinPool` runs **`RecursiveTask`s** that split themselves (`fork()`) and join their halves — the engine under parallel streams and `CompletableFuture`'s async methods. Work-stealing lets idle threads take tasks from busy ones' deques. You rarely write a `RecursiveTask` by hand today; you should know that the common pool has `cores - 1` threads and that blocking inside it is the cardinal sin.

Interview angle

"`Future` versus `CompletableFuture`?" Pull versus push: `CompletableFuture` composes (`thenApply`, `thenCompose`, `thenCombine`, `allOf`), recovers (`exceptionally`, `handle`) and can be completed manually.

"`thenApply` versus `thenCompose`?" `map` versus `flatMap` — compose when the function returns a future.

"`CountDownLatch` versus `CyclicBarrier`?" One-shot count-to-zero waited on by others versus a reusable meeting point for the parties themselves.

"How do you stop a consumer thread cleanly?" A poison pill through the queue, or interruption.

"What thread runs `thenApply`?" The one that completed the previous stage (or the caller if already complete); `thenApplyAsync` re-dispatches.

Key takeaways

`supplyAsync` → `thenApply` / `thenCompose` / `thenCombine` → `join`; pass an executor for blocking work; `Async` variants change threads.

Failures skip to `exceptionally` / `handle`; `allOf` for many, `anyOf` for the first; `orTimeout` for deadlines.

Latch: wait for *n* one-off events. Barrier: *k* parties meet repeatedly, with an action per trip. Semaphore: *n* permits.

`BlockingQueue` + poison pill = producer–consumer with back-pressure and built-in visibility.

The common pool is shared: never block in it.

EXERCISE 1

A pipeline of futures

Read two tokens `a` and `b`. Build `fa = supplyAsync(parse a).exceptionally(e -> 0)` and `fb` likewise, so that a non-numeric token becomes `0`. Then `squared = fa.thenApply(x -> x * x)`, `combined = squared.thenCombine(fb, Integer::sum)`, and `ratio = fa.thenCombine(fb, (x, y) -> x / y).handle(...)` producing `ratio=<value>` or `ratio=error <simple class name of the cause>`. Print, in order, `a=<fa>`, `b=<fb>`, `squared=<>`, `combined=<>` and the ratio line, using `join()`.

Example: `6 3` →

```
a=6
b=3
squared=36
combined=39
ratio=2
```

Example: `x 0` → `a=0`, `b=0`, `squared=0`, `combined=0`, `ratio=error ArithmeticException`.

INPUT 1

```
6 3
```

OUTPUT 1

```
a=6
b=3
squared=36
combined=39
ratio=2
```

INPUT 2

```
x 0
```

OUTPUT 2

```
a=0
b=0
squared=0
combined=0
ratio=error ArithmeticException
```

REFERENCE SOLUTION

```
import java.util.*;
import java.util.concurrent.*;

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        String a = in.next(), b = in.next();
        CompletableFuture<Integer> fa = CompletableFuture.supplyAsync(() -> Integer.
        CompletableFuture<Integer> fb = CompletableFuture.supplyAsync(() -> Integer.
        CompletableFuture<Integer> squared = fa.thenApply(x -> x * x);
        CompletableFuture<Integer> combined = squared.thenCombine(fb, Integer::sum);
        CompletableFuture<String> ratio = fa.thenCombine(fb, (x, y) -> x / y)
            .handle((v, e) -> e == null ? "ratio=" + v : "ratio=error " + e.getCause
        System.out.println("a=" + fa.join());
        System.out.println("b=" + fb.join());
        System.out.println("squared=" + squared.join());
        System.out.println("combined=" + combined.join());
        System.out.println(ratio.join());
    }
}
```

EXERCISE 2

Producer, consumer, poison pill

Read an integer `n`. A **producer** thread puts `1 … n` into an `ArrayBlockingQueue<Integer>` of capacity 4, then a poison pill `-1`. A **consumer** thread takes until it sees the pill, tracking how many values it consumed, their sum, and the largest. Join both threads and print `consumed=<n> sum=<n(n+1)/2> max=<n or 0>`. With capacity 4 and a large `n` the producer must block — the queue's back-pressure at work.

Example: `5` →

```
consumed=5 sum=15 max=5
```

INPUT 1

```
5
```

OUTPUT 1

```
consumed=5 sum=15 max=5
```

INPUT 2

```
0
```

OUTPUT 2

```
consumed=0 sum=0 max=0
```

REFERENCE SOLUTION

```
import java.util.*;
import java.util.concurrent.*;

public class Main {
    static final int POISON = -1;

    public static void main(String[] args) throws InterruptedException {
        int n = new Scanner(System.in).nextInt();
        BlockingQueue<Integer> queue = new ArrayBlockingQueue<>(4);
        int[] stats = new int[3];   // consumed, sum, max
        Thread producer = new Thread(() -> {
            try {
                for (int i = 1; i <= n; i++) queue.put(i);
                queue.put(POISON);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        });
        Thread consumer = new Thread(() -> {
            try {
                for (int x; (x = queue.take()) != POISON; ) {
                    stats[0]++;
                    stats[1] += x;
                    stats[2] = Math.max(stats[2], x);
                }
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        });
        producer.start();
        consumer.start();
        producer.join();
        consumer.join();
        System.out.println("consumed=" + stats[0] + " sum=" + stats[1] + " max=" + s
    }
}
```

QUIZ

1. `thenCompose` rather than `thenApply` when…

A. The function is slow

B. The function itself returns a `CompletableFuture` — like `flatMap`

C. You need a timeout

D. There are two futures

2. `supplyAsync(task)` without an executor runs the task on…

A. A new thread

B. The common `ForkJoinPool` — so never block there

C. The calling thread

D. The main thread

3. A stage throws. The following `thenApply` stages…

A. Run with `null`

B. Are skipped until an `exceptionally` / `handle` / `whenComplete`

C. Retry

D. Block

4. `CountDownLatch` versus `CyclicBarrier`:

A. Identical

B. A latch counts down once and is waited on by others; a barrier is a reusable meeting point for the parties themselves

C. A barrier cannot have an action

D. A latch resets

5. `BlockingQueue.put` on a full `ArrayBlockingQueue` …

A. Throws

B. Blocks until space frees — back-pressure on the producer

C. Drops the element

D. Grows the queue

6. Predict: `CompletableFuture.allOf(f1, f2).join()` returns…

A. A list of both results

B. `null` — `allOf` is `CompletableFuture<Void>`; read the results from `f1` and `f2` afterwards

C. The first result

D. A combined string

ANSWER KEY · COMPLETABLEFUTURE AND THE COORDINATION TOOLS

1. B `thenApply` would give `CompletableFuture<CompletableFuture<T>>`.

2. B Pass your own executor for blocking I/O.

3. B Exactly like a thrown exception skipping statements to a `catch`; the cause is wrapped in `CompletionException`.

4. B Latch: wait for *n* events. Barrier: *k* threads meet every round, with an optional action per trip.

5. B `offer` returns `false` instead; `offer(x, timeout, unit)` waits with a deadline.

6. B `allOf` tells you *when*; the values live in the original futures, whose `join()` is now instant.

CHECKPOINT · 30 MIN

Checkpoint — Concurrency

This checkpoint covers threads and the interrupt protocol, races and `synchronized`, the Java Memory Model with `volatile` and safe publication, executors and `Future`s, `ReentrantLock`, atomics and the concurrent collections, `CompletableFuture`, and the coordination tools — latch, barrier, semaphore and blocking queue.

How it works. Fifteen questions and three programs; 70% on the questions and every program accepted clears the module.

Before you start, make sure you can answer:

- `start()` versus `run()`, and how a thread is stopped politely.
- Why `count++` is a race, and the two guarantees `synchronized` gives.
- What `volatile` does and does not do, and why double-checked locking needs it.
- Which happens-before edge makes results written by a worker visible after `join()`.
- Fixed versus cached pool, what `Future.get()` throws, and the shutdown sequence.
- Why `ConcurrentHashMap.merge` removes the check-then-act race a synchronised `HashMap` has.
- `thenApply` versus `thenCompose`, and which thread runs a `thenX` stage.
- `CountDownLatch` versus `CyclicBarrier`, and how a poison pill ends a consumer.

The programs are a parallel row-sum on a fixed pool with results in order, a bank that transfers money between accounts under per-account locks acquired in a global order, and a phased simulation on a `CyclicBarrier` whose barrier action reports each round.

EXERCISE 1

Row sums on a pool

Read `r`, `c` and a pool size `p`, then an `r × c` matrix of integers. On a fixed pool of `p` threads, submit one `Callable<Long>` per row with `invokeAll`, then print `row <i>: <sum>` for every row in order and `total=<sum of all>`. Shut the pool down, await termination, and print `terminated=true`.

Example: `2 3 2` then `1 2 3` / `4 5 6` →

```
row 0: 6
row 1: 15
total=21
terminated=true
```

INPUT 1

```
2 3 2
1 2 3
4 5 6
```

OUTPUT 1

```
row 0: 6
row 1: 15
total=21
terminated=true
```

INPUT 2

```
1 1 1
-9
```

OUTPUT 2

```
row 0: -9
total=-9
terminated=true
```

REFERENCE SOLUTION

```
import java.util.*;
import java.util.concurrent.*;

public class Main {
    public static void main(String[] args) throws Exception {
        Scanner in = new Scanner(System.in);
        int r = in.nextInt(), c = in.nextInt(), p = in.nextInt();
        long[][] m = new long[r][c];
        for (int i = 0; i < r; i++) for (int j = 0; j < c; j++) m[i][j] = in.nextLor
        ExecutorService pool = Executors.newFixedThreadPool(p);
        List<Callable<Long>> tasks = new ArrayList<>();
        for (long[] row : m) tasks.add(() -> Arrays.stream(row).sum());
        List<Future<Long>> futures = pool.invokeAll(tasks);
        long total = 0;
        for (int i = 0; i < r; i++) {
            long s = futures.get(i).get();
            total += s;
            System.out.println("row " + i + ": " + s);
        }
        System.out.println("total=" + total);
        pool.shutdown();
        pool.awaitTermination(10, TimeUnit.SECONDS);
        System.out.println("terminated=" + pool.isTerminated());
    }
}
```

EXERCISE 2

Transfers with ordered locks

Read an integer `n` and `n` opening balances, then `t` and `t` transfers `from to amount`. Give **each account its own** `ReentrantLock`. Run every transfer on its own thread; a transfer locks the two accounts **in ascending index order** (lower index first — the rule that makes deadlock impossible), moves the money (balances may go negative), and unlocks both in `finally` blocks. Join all, then print `balances=<Arrays.toString>`, `total=<sum>` and `preserved=<total equals opening total>`.

Example: `3` then `100 50 0`, `3` then `0 1 30`, `1 2 60`, `2 0 10` →

```
balances=[80, 20, 50]
total=150
preserved=true
```

INPUT 1

```
3
100 50 0
3
0 1 30
1 2 60
2 0 10
```

OUTPUT 1

```
balances=[80, 20, 50]
total=150
preserved=true
```

INPUT 2

```
2
10 10
2
0 1 5
1 0 5
```

OUTPUT 2

```
balances=[10, 10]
total=20
preserved=true
```

REFERENCE SOLUTION

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

public class Main {
    static int[] balances;
    static ReentrantLock[] locks;

    static void transfer(int from, int to, int amount) {
        int first = Math.min(from, to), second = Math.max(from, to);
        locks[first].lock();
        try {
            if (second ≠ first) locks[second].lock();
            try {
                balances[from] -= amount;
                balances[to] += amount;
            } finally {
                if (second ≠ first) locks[second].unlock();
            }
        } finally {
            locks[first].unlock();
        }
    }

    public static void main(String[] args) throws InterruptedException {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        balances = new int[n];
        locks = new ReentrantLock[n];
        for (int i = 0; i < n; i++) { balances[i] = in.nextInt(); locks[i] = new Ree
        int opening = Arrays.stream(balances).sum();
        int t = in.nextInt();
        List<Thread> threads = new ArrayList<>();
        for (int i = 0; i < t; i++) {
            int from = in.nextInt(), to = in.nextInt(), amount = in.nextInt();
            Thread th = new Thread(() -> transfer(from, to, amount));
            threads.add(th);
            th.start();
        }
        for (Thread th : threads) th.join();
        int total = Arrays.stream(balances).sum();
        System.out.println("balances=" + Arrays.toString(balances));
        System.out.println("total=" + total);
        System.out.println("preserved=" + (total == opening));
    }
}
```

EXERCISE 3

Rounds at the barrier

Read `k` (parties), `r` (rounds) and `k` integer values. Create a `CyclicBarrier(k, action)` whose action prints `round <j>: total=<running total>` — it runs once per round, on the last thread to arrive. Each of the `k` threads, for rounds `j = 1 … r`, adds `value × j` to a shared `AtomicLong` and then `await()`s the barrier. Join all threads and print `final=<total>`. After round `j` the total is `Σvalues × (1 + 2 + … + j)` — the barrier makes that deterministic.

Example: `2 3` then `1 2` →

```
round 1: total=3
round 2: total=9
round 3: total=18
final=18
```

INPUT 1

```
2 3
1 2
```

OUTPUT 1

```
round 1: total=3
round 2: total=9
round 3: total=18
final=18
```

INPUT 2

```
1 2
5
```

OUTPUT 2

```
round 1: total=5
round 2: total=15
final=15
```

REFERENCE SOLUTION

```java
import java.util.*;
import java.util.concurrent.*;
import java.util.concurrent.atomic.*;

public class Main {
    public static void main(String[] args) throws InterruptedException {
        Scanner in = new Scanner(System.in);
        int k = in.nextInt(), r = in.nextInt();
        int[] values = new int[k];
        for (int i = 0; i < k; i++) values[i] = in.nextInt();
        AtomicLong total = new AtomicLong();
        AtomicInteger round = new AtomicInteger();
        CyclicBarrier barrier = new CyclicBarrier(k, () -> {
            System.out.println("round " + round.incrementAndGet() + ": total=" + tot
        });
        Thread[] workers = new Thread[k];
        for (int i = 0; i < k; i++) {
            final int v = values[i];
            workers[i] = new Thread(() -> {
                try {
                    for (int j = 1; j <= r; j++) {
                        total.addAndGet((long) v * j);
                        barrier.await();
                    }
                } catch (InterruptedException | BrokenBarrierException e) {
                    Thread.currentThread().interrupt();
                }
            });
            workers[i].start();
        }
        for (Thread w : workers) w.join();
        System.out.println("final=" + total.get());
    }
}
```

CHECKPOINT QUESTIONS

1. `new Thread(r).run()` executes `r` …

A. On a new thread

B. On the calling thread — no concurrency

C. Twice

D. Never

2. Swallowing `InterruptedException` with an empty `catch` …

A. Is fine

B. Loses the stop request; propagate it or call `Thread.currentThread().interrupt()`

C. Stops the thread

D. Is required

3. `synchronized` provides…

A. Atomicity only

B. Mutual exclusion and visibility (release/acquire), reentrantly

C. Visibility only

D. Fairness

4. Two threads lock A then B and B then A respectively. The risk is…

A. Starvation

B. Deadlock

C. Livelock

D. Nothing

5. `volatile` guarantees…

A. Atomic compound updates

B. Visibility and ordering of reads and writes to that field

C. Mutual exclusion

D. Faster access

6. Which action creates a happens-before edge that lets `main` read a worker's results?

A. `Thread.sleep(50)` in `main`

B. `worker.join()`

C. `System.out.println` in the worker

D. Declaring the array `static`

7. A production thread pool should have…

A. Unbounded threads

B. A bounded queue and a bounded thread count with an explicit rejection policy

C. One thread

D. No queue

8. A task submitted with `submit()` throws; nothing is printed because…

A. The pool swallows it

B. The exception is stored in the `Future` until `get()` is called

C. It was a daemon

D. Exceptions are disabled in pools

9. `ReentrantLock.tryLock()` …

A. Blocks like `lock()`

B. Returns `false` immediately if the lock is held — the tool for deadlock avoidance with back-off

C. Throws when held

D. Is deprecated

10. `LongAdder` beats `AtomicLong` when…

A. You need the exact value after every update

B. Many threads increment often and the value is read rarely

C. The value is small

D. Never

11. `ConcurrentHashMap` versus `Collections.synchronizedMap(new HashMap<>())`:

A. Same thing

B. Per-bin locking, lock-free reads, atomic `merge` / `compute*`, weakly consistent iteration versus one global lock with no atomic compound operations

C. The synchronised map is faster

D. `ConcurrentHashMap` allows null

12. Predict: `supplyAsync(() → 1).thenApply(x → x / 0).exceptionally(e → -1).join()` returns…

A. `1`

B. `-1`

C. Throws `ArithmeticException`

D. `0`

13. A `thenApply` stage runs on…

A. Always a new thread

B. The thread that completed the previous stage (or the caller if already complete); `thenApplyAsync` re-dispatches

C. The main thread

D. The GC thread

14. The poison-pill pattern…

A. Kills the consumer thread

B. Sends a sentinel value through the queue so the consumer exits its loop cleanly

C. Uses `Thread.stop`

D. Requires `notifyAll`

15. Blocking inside a parallel stream or `supplyAsync` without an executor is bad because…

A. It is slow

B. It parks a common-pool thread, starving every other user of the pool

C. It throws

D. It deadlocks immediately

ANSWER KEY · CHECKPOINT — CONCURRENCY

1. B Only `start()` spawns a thread.

2. B The flag is cleared when the exception is thrown; restore it for callers higher up.

3. B Both guarantees, on the monitor named; fairness needs `ReentrantLock(true)`.

4. B Inconsistent lock ordering; fix by ordering, one lock, or `tryLock`.

5. B A stop flag or a published immutable reference — not `count++`.

6. B Join, volatile, locks and `java.util.concurrent` hand-offs; nothing else.

7. B `ThreadPoolExecutor` with `ArrayBlockingQueue` and `CallerRunsPolicy` / `AbortPolicy`; the factories leave one side unbounded.

8. B Always collect futures, or use `execute()` for loud failures.

9. B `tryLock(timeout, unit)` waits up to a deadline.

10. B Per-thread cells avoid CAS contention; `sum()` combines them.

11. B And `ConcurrentHashMap` rejects `null` keys and values.

12. B The failure skips to `exceptionally`, which recovers with −1.

13. B Which matters when the previous stage completed on an event-loop or UI thread.

14. B One pill per consumer when there are several.

15. B The common `ForkJoinPool` has about `cores - 1` threads shared JVM-wide; blocking I/O belongs on its own executor.