Java lessons
25/09/2025

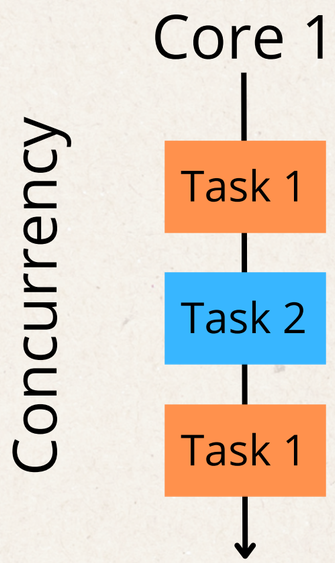
CONCURRENCY IN JAVA

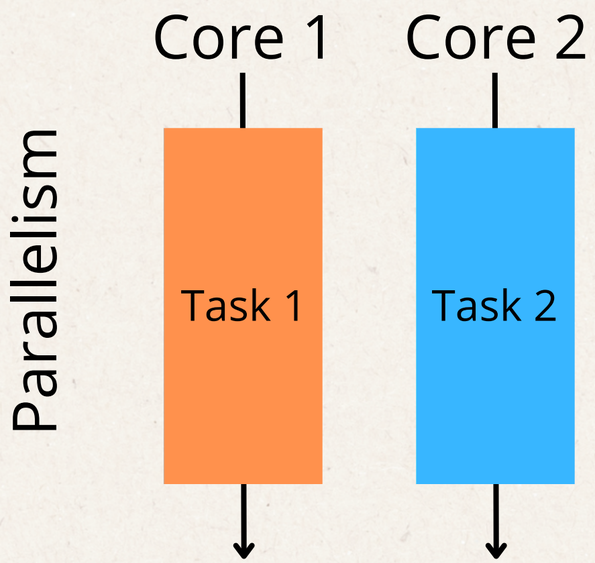
Practical guide

CREATED BY:
Emir Totić - Software Engineer

WHAT IS CONCURRENCY?

- Concurrency = managing multiple tasks at the same time
- Improves performance and CPU utilization
- Allows applications to serve many users simultaneously
- Difference:
  - Concurrency = tasks are executed interleaved (taking turns)
  - Parallelism = tasks are executed at the same time

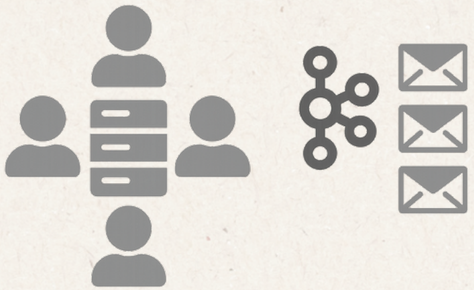




Java

WHY IS CONCURRENCY CRUCIAL IN MODERN SYSTEMS

- **Scalability** – applications can handle thousands of users at once
- **Performance** – better CPU utilization and faster response times
- **Resilience** – systems remain stable under heavy load
- **Real-world use cases**:
  - Web servers handling parallel requests
  - Message processing in Kafka topics
  - Microservices communicating asynchronously



```
ExecutorService pool = Executors.newFixedThreadPool(10);
pool.submit(() -> handleRequest("User A"));
pool.submit(() -> handleRequest("User B"));
```

Java

SYNCHRONIZED – ACCESS CONTROL

- Allows only one thread at a time to access the critical section.
- Used for thread-safe access to shared resources.
- Simple mechanism, but may reduce performance.

✅ **Use when multiple threads modify the same object or variable**

- Example: all threads use the same counter or a shared list.
- Without synchronized, a race condition may occur (two threads modify the value at the same time and the result becomes incorrect).
- synchronized guarantees that only one thread executes the critical section at a given moment.

⚠️ **Do not use for small operations that are called very frequently**

- Example: if you have a method called a million times per second, synchronized introduces lock/unlock overhead.
- This means the program will become slower because threads constantly wait for each other.
- In such cases, it’s better to use AtomicInteger or other optimized concurrency classes.

VOLATILE – VARIABLE VISIBILITY ACROSS THREADS

- Guarantees that every thread sees the most up-to-date value of the variable.
- Prevents caching of the value in thread-local memory.
- Does not solve atomic operations (e.g., counter++ is not thread-safe).

✅ When to use as a flag

- volatile is perfect when you want to control the state of a thread through a simple variable.
- Most common example: stopping a thread.
- If you set volatile boolean running = true; - every thread will always see the latest value.
- So, when one thread calls running = false; - another thread using that variable immediately sees the change and stops working.

Practical scenario: *a service continuously processing data, but you want to stop it gracefully without a "hard kill."*

⚠️ Do not use for counters and complex operations

- Problem: operations like counter++ are not simple – under the hood, they are three steps:
  a. Load the value
  b. Increment the value
  c. Save the new value
- If two threads do this simultaneously, the result may be wrong (race condition).
- volatile guarantees visibility but does not prevent these three steps from interleaving.

Practical scenario: *if multiple threads need to increment the same number, volatile won't help – you should use AtomicInteger or synchronized.*

ATOMIC INTEGER – THREAD-SAFE COUNTER

- Solves the problem that volatile cannot → provides atomic operations.
- No need for synchronized → faster and more efficient.
- Operations (increment, decrement, compareAndSet) are guaranteed to be thread-safe.

✅ When multiple threads modify a shared counter or state

- If you have a counter that all threads increment (e.g., the number of requests a server has processed), AtomicInteger ensures the result is always correct.
- Unlike volatile, where a race condition can occur, here each operation (incrementAndGet) is indivisible (atomic).
- This means there's no chance of two threads "colliding" and overwriting the result.

✅ Great for high-performance scenarios where synchronized slows things down

- synchronized puts a lock on a method or block, and each thread has to wait its turn.
- This creates a bottleneck when there are many calls.
- AtomicInteger works without a traditional lock – it uses the CAS (Compare-And-Swap) mechanism at the processor level, which is much faster.

Advantage: thread-safe and efficient

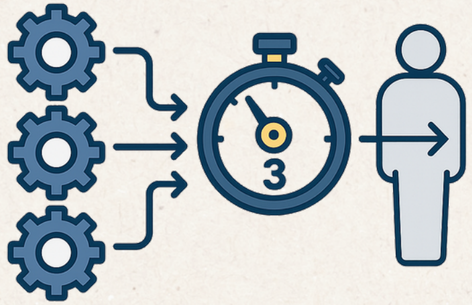
- You get the best of both worlds:
  - Safe as if you were using synchronized.
  - Fast because there's no unnecessary locking.
- That's why AtomicInteger is used in situations with a lot of reads and writes where the result must remain accurate.

COUNTDOWNLATCH – WAITING FOR TASK COMPLETION

- A mechanism that allows one or more threads to wait until a certain number of operations have finished.
- Works like a countdown – each countDown() decreases the counter, and when it reaches zero → await() continues.
- Ideal for synchronizing multiple tasks.

When to use?

- When you want the main thread to wait until multiple parallel tasks are completed.
- In integration tests – waiting for all services or simulations to finish before evaluating results.
- One-time use – once the latch reaches zero, it cannot be reset.



Java

COMPLETABLE FUTURE – ASYNCHRONOUS PROGRAMMING

- Allows tasks to run in the background (async).
- Supports chaining (linking multiple operations in a sequence).
- Useful for IO operations, API calls, databases where waiting for a result is required.
- Provides clean code instead of complicated callbacks.

When to use?

- When you want to run independent tasks in parallel.
- When multiple API calls need to be processed at the same time.
- Advantage: easier handling of asynchronous code compared to plain threads.

```
CompletableFuture.supplyAsync(() -> getData())
                 .thenApply(data -> process(data))
                 .thenAccept(System.out::println);
```

COMPARISON OF CONCURRENCY TOOLS IN JAVA

Tool	When to use?	Advantage
synchronized	Protecting critical sections	Simple mechanism
volatile	Variable visibility across threads	Lightweight and fast
AtomicInteger	Thread-safe counter, shared state	Efficient, no lock needed
CountDownLatch	Waiting for multiple tasks	Simple coordination
CompletableFuture	Asynchronous programming, API calls	Clean async code, chaining

- Each tool has its own purpose – there is no “one size fits all.”
- synchronized and volatile are the basics, but they are often not enough.
- AtomicInteger, CountDownLatch, and CompletableFuture provide more modern and efficient solutions.
- Understanding concurrency tools = a senior mindset and the key to building scalable systems.

THANK YOU FOR READING!

- If you found this guide useful, follow me for more practical topics on Java and backend architecture.
- My idea is to share knowledge about:
  - Java & Spring Boot
  - Kafka & Microservices
  - System Design & Architecture and more

Let’s connect

www.linkedin.com/in/emirtotic

https://emirtotic.github.io/portfolio-site

emirtotic@gmail.com