Virtual Threads: Lightweight Concurrency for Everyone

Traditional Threads

- Java threads = OS threads (1:1 mapping)
- OS threads = expensive (memory + context switching)
- Thread-per-request style = simple, but doesn’t scale

🛑 Result: **Throughput capped by OS thread limits**

Async / Reactive Programming

- Non-blocking APIs + callbacks (`CompletableFuture`, reactive frameworks)
- Better scalability but…
  - ❌ Harder to read (no loops, try/catch)
  - ❌ Debugging and profiling painful
  - ❌ Stack traces lose meaning

What they are:

- Instances of `java.lang.Thread`
- **M:N scheduling**: Many (M) virtual threads mapped to fewer (N) OS threads
- Lightweight, cheap, and plentiful

📌 **Analogy**: Like OS “virtual memory”, but for threads

Virtual Threads - Goals 🎯

- Keep the **thread-per-request** model
- Near-optimal hardware utilization
- Compatible with existing code (`ThreadLocal`, debugging, profiling)
- No new programming model to learn

Virtual Threads - Using Virtual Threads (Example)

```
try (var executor = Executors.newVirtualThreadPerTaskExecutor()) {
    IntStream.range(0, 10_000).forEach(i ->
        executor.submit(() -> {
            Thread.sleep(Duration.ofSeconds(1));
            return i;
        })
    );
}
```

✅ 10,000 concurrent tasks → runs fine

❌ 10,000 OS threads → likely crash

Virtual Threads - When to Use?

Best for **high concurrency I/O-bound** tasks (HTTP calls, DB queries, etc.)

Not faster than platform threads → they give **scale, not speed**

Don’t pool virtual threads — create one per task!

Virtual Threads - Observability 🔍

Virtual threads = fully supported by tooling

Debuggers, profilers, **JFR**

New **thread dumps** grouped by task

JSON export via `jcmd`

Virtual Threads - The Pinning Issue (Java 21 → 23)

- Virtual threads could get **pinned** (stuck to carrier OS thread):
  - Inside `synchronized` blocks
  - Native calls / FFM API

⚠️ Pinned thread = carrier blocked → scalability suffers

Workarounds:

- Replace `synchronized` with `ReentrantLock`
- Monitor with `-Djdk.tracePinnedThreads` or **JFR events**

Synchronize Virtual Threads Without Pinning

- Virtual threads can now acquire/release monitors **independently of carriers**
- ✅ `synchronized` no longer causes pinning
- ✅ `ConcurrentHashMap` and many libraries safe
- 🗑️ `-Djdk.tracePinnedThreads` removed → not needed anymore
- Monitoring only needed for **native calls**

Virtual Threads - Forgive `synchronized` ❤️

In 2023: “Avoid synchronized, use locks!”

In 2025: “Use the best tool for the job”

JEP 491 officially says: choose between `synchronized` and `ReentrantLock`
based on semantics, **not pinning fears**

Virtual Threads - Key Takeaways ✅

Virtual threads = **simple + scalable** concurrency

Ideal for **server apps** (thread-per-request style preserved)

Tooling ready: debugging, JFR, thread dumps

Java 24: **pinning issue solved**

Java 25: Virtual threads are **mature and production-ready**