Java Threads?!

Writing clean &

Efficient Code

Introduction

Understand Java Multi-Threading

Ever wondered how apps juggle multiple tasks at once? Let's unravel the magic behind threads, locks, and executors! This tutorial will guide you through writing code that's fast, safe, and ready for the real world—no stress, just clarity. Let's explore together! 🚀

- Threads 101
- Thread Safety
- Executor Services
- Locks & Tools
- Best Practices

Threads

What are threads?

A thread is the smallest unit of execution within a process. A process can have multiple threads running concurrently. Think of a restaurant kitchen (process) with multiple chefs (threads) preparing different dishes (tasks) simultaneously.

Example

```
class MyTask implements Runnable {
    @Override
    public void run() {
        System.out.println(
          "Task executed by thread: " + Thread.currentThread().getName()
        );
        // ... task logic ...
    }
}

MyTask task = new MyTask();
Thread thread1 = new Thread(task);
thread1.start(); // Starts the new thread
```

Process vs. Thread: A **Process** is an instance of a running program. It has its own dedicated memory space, resources. A **Thread** exists within a process. Multiple threads within the same process share the process's resources, especially memory (the heap space)

Thread Safety

What does thread safety mean?

Example

```
class UnsafeCounter {
    private int count = 0;
    // Not thread-safe! Concurrent calls may skip increments.
    void increment() { count++; }
}

class SafeCounter {
    private int count = 0;
    // Thread-safe: synchronized guarantees mutual exclusion
    synchronized void increment() { count++; }
}
```

Code is thread-safe if it behaves correctly when accessed by multiple threads. Uncontrolled concurrent access can lead to race conditions. When a thread invokes a synchronized instance method, it automatically acquires the intrinsic lock for that object. No other thread can enter any synchronized method on the same object until the first thread exits the method.

Race Condition: When multiple threads access shared mutable state (data that can be changed) concurrently, unexpected and incorrect results can occur if access is not controlled. This is called a race condition.

Executor Services

Modern Concurrency

Executor Services handle the messy work of threads—creation, recycling, and task juggling—so you don't have to. They reuse threads smartly (like a reusable crew!), slash overhead, and prevent system meltdowns. Add task queues, scheduling, and clean shutdowns? Done. Turn chaos into order for servers, data jobs, and beyond. 🧵→🚀

Example

```java
class MyTask implements Runnable {
    @Override
    public void run() {
        // ... task logic ...
    }
}

ExecutorService executor = Executors.newFixedThreadPool(4);

for (int i = 1; i ≤ 5; i++) {
  executor.submit(new MyTask());
}
executor.shutdown(); // Always shut down!
```

Thread Pooling: Reuses existing threads, reducing the overhead of thread creation.

Locks

Explicit Locks: More flexible than synchronized

Example

```
public class SharedObject {
    private ReentrantLock lock = new ReentrantLock();
    private int counter = 0;

    public void increment() {
        lock.lock(); // Acquire the lock
        try {
            counter++;
        } finally {
            lock.unlock(); // Release the lock
        }
    }
}
```

Think of locks like a restroom key at a gas station. Only one person (thread) can use it at a time—everyone else waits their turn. Java's ReentrantLock is that key, but smarter:

- Timeout? tryLock(5, SECONDS) "I'll wait 5 seconds, then leave if it's occupied."
- Fairness? new ReentrantLock(true) "First in line gets it next, no cutting!"
- Safety Net? Always unlock() in finally "Return the key even if you flood the sink!"

When? Use locks when synchronized feels too rigid like when you need bouncer-level control over who accesses what, and when. 🔐💻

Best Practices

Best Practices for Java Multithreading

1 - **Prefer Executors Over Raw Threads**:

Reduces overhead and improves resource management.

2 - **Avoid Shared Mutable State**:

Use immutable objects (e.g., final fields) where possible.

3 - **Use High-Level Tools**:

- Concurrent Collections: ConcurrentHashMap, CopyOnWriteArrayList.
- Atomic Variables: AtomicInteger, AtomicReference (lock-free thread safety).

4 - **Handle Exceptions**:

Catch exceptions in threads (uncaught exceptions can terminate the thread silently).

5 - **Avoid Deadlocks**:

- Acquire locks in a consistent order.
- Use timeouts with tryLock().

You know that 'aha!' moment when threads finally make sense? I live for those! **Let's grab virtual coffee and geek out - my treat ☕.**