

Java Concurrency Overview

A Quick Tour of What's Available

Today we'll cover:

- Creating and Running Threads
- Synchronization Basics (synchronized, volatile)
- Java API Support (Collections, Executors, Synchronizers)
- Quick Tips & Common Patterns

1. Creating Threads in Java

Two Main Approaches

Option A: Extend Thread

```
class MyThread extends Thread {  
    public void run() {  
        System.out.println("Hello from thread!");  
    }  
}  
  
// Usage  
MyThread t = new MyThread();  
t.start(); // NOT t.run()!
```

Option B: Implement Runnable (Preferred!)

```
class MyTask implements Runnable {  
    public void run() {  
        System.out.println("Hello from task!");  
    }  
}  
  
// Usage  
Thread t = new Thread(new MyTask());  
t.start();
```

Why Runnable? More flexible — extend another class + better separation of task and execution.

2. Synchronization Basics

Protecting Shared State

✗ Problem: Race Condition

```
class Counter {  
    private int count = 0;  
  
    public void increment() {  
        count++; // NOT atomic! (read-modify-write)  
    }  
  
    public int getCount() { return count; }  
}
```

✓ Solution 1: synchronized

```
class SafeCounter {  
    private int count = 0;  
  
    public synchronized void increment() {  
        count++; // Now thread-safe!  
    }  
  
    public synchronized int getCount() {  
        return count;  
    }  
}
```

Key Point: `synchronized` ensures mutual exclusion — only one thread at a time can execute the method.

2. Synchronization (cont.)

More Options

Synchronized Blocks

```
class BankAccount {
    private int balance = 0;
    private final Object lock = new Object();

    public void deposit(int amount) {
        synchronized(lock) {
            balance += amount;
        }
    }
}
```

volatile Keyword

```
class StatusFlag {
    private volatile boolean running = true;

    public void stop() {
        running = false; // Visible to all threads
    }

    public void work() {
        while(running) { // Always sees latest value
            // do work
        }
    }
}
```

volatile: Guarantees visibility, but NOT atomicity. Good for flags, not for counters!

3. Java API Support

Concurrent Collections

Old Way (Don't use for new code)

```
Vector<String> list = new Vector<>(); // Synchronized
Hashtable<String, Integer> map = new Hashtable<>();
```

Better: Concurrent Collections

```
// High performance, thread-safe
ConcurrentHashMap<String, Integer> map =
    new ConcurrentHashMap<>();

CopyOnWriteArrayList<String> list =
    new CopyOnWriteArrayList<>();

// For producer-consumer patterns
BlockingQueue<Task> queue =
    new LinkedBlockingQueue<>();

queue.put(task); // Blocks if full
Task t = queue.take(); // Blocks if empty
```

Why better? More scalable, better performance under concurrent access.

3. Executor Framework

Thread Pools Made Easy

✗ Manual Thread Management (Bad)

```
for (int i = 0; i < 100; i++) {  
    new Thread(() -> processRequest()).start();  
}  
// Creates 100 threads! Expensive and dangerous
```

✓ Use Executors (Good)

```
ExecutorService executor =  
    Executors.newFixedThreadPool(10);  
  
for (int i = 0; i < 100; i++) {  
    executor.execute(() -> processRequest());  
}  
  
executor.shutdown(); // Don't forget!
```

Getting Results Back

```
Future<Integer> future = executor.submit(() -> {  
    return expensiveCalculation();  
});  
  
Integer result = future.get(); // Blocks until ready
```

3. Synchronizers

Coordinating Threads

CountDownLatch - Wait for Events

```
CountDownLatch latch = new CountDownLatch(3);

// 3 worker threads
for (int i = 0; i < 3; i++) {
    new Thread(() -> {
        doWork();
        latch.countDown(); // Signal done
    }).start();
}

latch.await(); // Wait for all 3 to finish
System.out.println("All done!");
```

Semaphore - Control Access

```
Semaphore permits = new Semaphore(3);

permits.acquire(); // Get permit (blocks if none)
try {
    accessLimitedResource();
} finally {
    permits.release(); // Always release!
}
```

CyclicBarrier - Wait for Each Other

```
CyclicBarrier barrier = new CyclicBarrier(3);

// All 3 threads wait for each other
barrier.await(); // Blocks until all 3 arrive
```

4. Quick Tips & Patterns

Things to Remember

✗ Common Mistakes

- Calling `run()` instead of `start()`
- Forgetting to synchronize when needed
- Over-synchronizing (performance killer!)
- Not handling `InterruptedException` properly
- Creating too many threads manually

✓ Best Practices

- Use Executors for thread management
- Prefer concurrent collections over synchronized ones
- Use `volatile` for simple flags
- Keep synchronized blocks small and fast
- Document your thread-safety guarantees
- Test with multiple threads!

💡 When in Doubt

Ask yourself: "Can two threads access this at the same time? What happens if they do?"

Atomic Types

Lock-free primitives for thread-safe updates

Problem with regular types

```
int ticketsSold = 0;

// Multiple threads:
ticketsSold++; // NOT atomic
// Read-Modify-Write can lose updates
```

Result: race condition → incorrect counters

Atomic variables

```
// Thread-safe, indivisible operations
AtomicInteger ticketsSold = new AtomicInteger(0);

ticketsSold.incrementAndGet(); // atomic ++
int current = ticketsSold.get();
ticketsSold.addAndGet(5);
ticketsSold.compareAndSet(10, 20); // CAS

AtomicLong counter;
AtomicBoolean flag = new AtomicBoolean(false);
AtomicReference<State> state = new AtomicReference<>();
```

No synchronized needed for a single variable.

When to use

- ✓ Simple counters (tickets, scores)
- ✓ Flags (gameOver, shutdown)
- ✓ References that must update atomically



Limitations

- ✗ Cannot protect multiple correlated variables
- ✗ For complex invariants use synchronized or locks

Visual concept

```
Thread A reads 0
Thread B reads 0
Thread A writes 1
Thread B writes 1 ✗ lost update
```

AtomicInteger:

```
Thread A incrementAndGet() -> 1   
Thread B incrementAndGet() -> 2 
```

Performance: Often faster than synchronized for simple operations. Based on CPU CAS, typically lock-free.

Quick Reference Card

Java Concurrency Cheat Sheet

Creating Threads

Thread, Runnable, Callable

Synchronization

synchronized, volatile, Lock

Collections

ConcurrentHashMap,
CopyOnWriteArrayList, BlockingQueue

Executors

ExecutorService, Future,
ScheduledExecutor

Synchronizers

CountDownLatch, Semaphore,
CyclicBarrier

Atomic Types

AtomicInteger, AtomicBoolean,
AtomicReference