CSC 447 - Concepts of Programming Languages

Java Concurrency

Instructor: Eric J. Fredericks

Learning Objectives

❓ How do programming languages deal with concurrent execution?

- Describe threads and shared mutable state
- Identify race conditions and data races
- Apply synchronization mechanisms: locks, synchronized blocks, and atomic operations
- Describe higher-level concurrency abstractions in Java
- Compare concurrency approaches across languages

❓ Why Concurrency?

❓ Why do we need concurrent programs?

- Modern hardware has **multiple cores** — sequential programs waste resources
- Many problems are naturally concurrent: web servers, GUIs, simulations
- Concurrency vs. parallelism:
  - **Concurrency**: dealing with multiple things at once (structure)
  - **Parallelism**: doing multiple things at once (execution)
- ❗ Concurrency is a **language design** problem — how do we express and control it?
- ❓ Java was one of the first mainstream languages with built-in thread support.

Java Threads: The Basics

💡 A thread is a lightweight unit of execution.

```java
public class HelloThread {
    public static void main(String[] args) {
        Thread t = new Thread(() -> {
            System.out.println("Hello from thread: "
                + Thread.currentThread().getName());
        });
        t.start();
        System.out.println("Hello from main: "
            + Thread.currentThread().getName());
    }
}
```

```
$ javac HelloThread.java && java HelloThread
Hello from main: main
Hello from thread: Thread-0
```

❗ The order of output is **nondeterministic** — run it again and you might get a different order!

Shared Mutable State

❓ What happens when two threads modify the same variable?

```java
public class Counter {
    static int count = 0;

    public static void main(String[] args) throws InterruptedException {
        Thread t1 = new Thread(() -> {
            for (int i = 0; i < 100_000; i++) count++;
        });
        Thread t2 = new Thread(() -> {
            for (int i = 0; i < 100_000; i++) count++;
        });
        t1.start(); t2.start();
        t1.join();  t2.join();
        System.out.println("count = " + count);
    }
}
```

```
$ java Counter
count = 137462      # Expected 200000!
```

❗ **Race condition!** The result is different every time.

Why Is count++ Broken?

`count++` is not a single operation — it's three:

```
1
```

- Both threads read 42, both write 43. One increment is lost.
- This is a **data race**: unsynchronized access to shared mutable state where at least one access is a write.
- The interleaving depends on the OS thread scheduler — nondeterministic!

Synchronization: Locks

💡 A **lock** (mutex) ensures only one thread accesses a critical section at a time.

```java
public class SyncCounter {
    static int count = 0;
    static final Object lock = new Object();

    public static void main(String[] args) throws InterruptedException {
        Thread t1 = new Thread(() -> {
            for (int i = 0; i < 100_000; i++)
                synchronized (lock) { count++; }
        });
        Thread t2 = new Thread(() -> {
            for (int i = 0; i < 100_000; i++)
                synchronized (lock) { count++; }
        });
        t1.start(); t2.start();
        t1.join();  t2.join();
        System.out.println("count = " + count);
    }
}
```

```
$ java SyncCounter
count = 200000      # Correct every time!
```

synchronized Methods

💡 Java also lets you synchronize entire methods.

```java
public class BankAccount {
    private int balance;

    public BankAccount(int initial) { this.balance = initial; }

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

    public synchronized void withdraw(int amount) {
        if (balance >= amount) {
            balance -= amount;
        }
    }

    public synchronized int getBalance() {
        return balance;
    }
}
```

- ❗ `synchronized` on a method locks on `this` — only one thread can execute any `synchronized` method on that object at a time.
- ❓ Is this enough? What could go wrong?

Deadlock

❓ What happens when two threads each wait for the other's lock?

```java
public class Deadlock {
    static final Object lockA = new Object();
    static final Object lockB = new Object();

    public static void main(String[] args) {
        Thread t1 = new Thread(() -> {
            synchronized (lockA) {
                synchronized (lockB) { System.out.println("t1"); }
            }
        });
        Thread t2 = new Thread(() -> {
            synchronized (lockB) {
                synchronized (lockA) { System.out.println("t2"); }
            }
        });
        t1.start(); t2.start();
    }
}
```

- ❗ t1 holds lockA, waits for lockB. t2 holds lockB, waits for lockA.
- ❗ **Deadlock!** Both threads freeze forever.
- 💡 Java does not prevent deadlocks — it's the programmer's responsibility.

The Fundamental Problem

Shared mutable state + threads = pain.

```
1
```

- This is hard to get right, hard to test, and hard to debug.
- Can language design help?
- Recall: Rust prevents data races at compile time (aliasing XOR mutability).
- Functional programming avoids the problem: **immutable data can be freely shared**.

Atomic Operations

💡 For simple operations, Java provides lock-free atomic classes.

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

public class AtomicCounter {
    static AtomicInteger count = new AtomicInteger(0);

    public static void main(String[] args) throws InterruptedException {
        Thread t1 = new Thread(() -> {
            for (int i = 0; i < 100_000; i++) count.incrementAndGet();
        });
        Thread t2 = new Thread(() -> {
            for (int i = 0; i < 100_000; i++) count.incrementAndGet();
        });
        t1.start(); t2.start();
        t1.join();  t2.join();
        System.out.println("count = " + count.get());
    }
}
```

```
$ java AtomicCounter
count = 200000
```

💡 `AtomicInteger` uses hardware-level compare-and-swap (CAS) — no lock needed!

volatile Keyword

❓ What about visibility between threads?

```
1 public class Visibility {
2     static boolean running = true;   // not volatile!
3
4     public static void main(String[] args) throws InterruptedException {
5         Thread t = new Thread(() -> {
6             while (running) { /* spin */ }
7             System.out.println("Stopped!");
8         });
9         t.start();
10        Thread.sleep(100);
11        running = false;   // might never be seen by t!
12    }
13 }
```

❗ Without `volatile`, the JVM may **cache** `running` in a register — the thread never sees the update!

💡 `volatile` forces reads/writes to go through main memory.

❗ `volatile` guarantees visibility, but **not** atomicity of compound operations.

The Java Memory Model

❓ Why do we need a memory model?

- Modern CPUs reorder instructions and cache memory for performance
- Without rules, threads might see **stale** or **inconsistent** data
- The Java Memory Model (JMM) defines **happens-before** relationships:
  - Unlock on a monitor *happens-before* subsequent lock on that monitor
  - Write to `volatile` field *happens-before* subsequent read
  - `Thread.start()` *happens-before* any action in the started thread
  - All actions in a thread *happen-before* `Thread.join()` returns
- ❗ Without happens-before, you have **no guarantee** about what one thread sees from another.
- 💡 This is a **language-level** specification — it constrains what the JVM and CPU can optimize.

Thread Pools: Don't Create Raw Threads

💡 Creating a thread per task is wasteful. Use a thread pool.

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

public class PoolExample {
    public static void main(String[] args) throws Exception {
        ExecutorService pool = Executors.newFixedThreadPool(4);

        List<Future<Integer>> futures = new ArrayList<>();
        for (int i = 0; i < 10; i++) {
            final int n = i;
            futures.add(pool.submit(() -> {
                Thread.sleep(100);
                return n * n;
            }));
        }
        for (Future<Integer> f : futures) {
            System.out.println(f.get());    // blocks until result ready
        }
        pool.shutdown();
    }
}
```

- 💡 `ExecutorService` manages a pool of reusable threads.
- 💡 `Future<T>` is a handle to a result that will be available later.

CompletableFuture: Composable Async

💡 `CompletableFuture` lets you chain async operations — functional style!

```
 1 import java.util.concurrent.CompletableFuture;
 2
 3 public class AsyncChain {
 4     public static void main(String[] args) {
 5         CompletableFuture
 6             .supplyAsync(() -> "Hello")
 7             .thenApply(s -> s + " World")
 8             .thenApply(String::toUpperCase)
 9             .thenAccept(System.out::println)
10             .join();   // wait for completion
11     }
12 }
```

```
1 HELLO WORLD
```

💡 This is like `map` and `flatMap` on `Future` in Scala.

❗ No explicit threads, no locks — the framework handles scheduling.

Concurrent Collections

Java provides thread-safe collection implementations.

Unsafe

```
1 // HashMap is NOT thread-safe
2 Map<String, Integer> map = new HashMap<>();
3
4 // Two threads modifying:
5 // → ConcurrentModificationException
6 // → or silent corruption!
```

Safe

```
1 // ConcurrentHashMap IS thread-safe
2 Map<String, Integer> map =
3     new ConcurrentHashMap<>();
4
5 // Two threads modifying:
6 // → correct behavior, no locks needed
7 //   by the caller
```

- `ConcurrentHashMap` uses fine-grained locking internally — much better than wrapping everything in `synchronized`.
- `Collections.synchronizedMap()` exists but is coarse-grained — locks the whole map.

Producer-Consumer Pattern

A classic concurrency pattern using `BlockingQueue`.

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

public class ProducerConsumer {
    public static void main(String[] args) {
        BlockingQueue<Integer> queue = new ArrayBlockingQueue<>(10);

        Thread producer = new Thread(() -> {
            for (int i = 0; i < 20; i++) {
                try {
                    queue.put(i);      // blocks if queue is full
                    System.out.println("Produced: " + i);
                } catch (InterruptedException e) { break; }
            }
        });

        Thread consumer = new Thread(() -> {
            for (int i = 0; i < 20; i++) {
                try {
                    int val = queue.take();  // blocks if queue is empty
                    System.out.println("Consumed: " + val);
                } catch (InterruptedException e) { break; }
            }
        });

        producer.start(); consumer.start();
    }
}
```

`BlockingQueue` handles all the synchronization — no manual locking!

Virtual Threads (Java 21+)

💡 Java 21 introduced **virtual threads** — lightweight, JVM-managed threads.

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

public class VirtualThreads {
    public static void main(String[] args) throws Exception {
        try (var executor = Executors.newVirtualThreadPerTaskExecutor()) {
            for (int i = 0; i < 100_000; i++) {
                final int n = i;
                executor.submit(() -> {
                    Thread.sleep(1000);
                    return n;
                });
            }
        }   // waits for all tasks to complete
        System.out.println("Done!");
    }
}
```

- ❗ 100,000 OS threads would crash. 100,000 virtual threads is fine!
- 💡 Virtual threads are extremely cheap and heap-allocated — create one per task, no pooling.
- 💡 The JVM multiplexes them onto OS threads, unmounting transparently on block. Similar to Go's goroutines.
- 💡 Existing blocking code works as-is. Avoid synchronized/JNI — they pin virtual threads to OS threads!
- 💡 Similar motivation to Go's goroutines.

Concurrency Across Languages

❓ How do other languages approach concurrency?

Language	Primary Model	Key Idea
Java	Shared memory + locks	Threads share heap; `synchronized` for safety
Scala	Actors (Akka) / Futures	Message passing; immutable data preferred
Rust	Ownership + `Send` / `Sync`	Compiler prevents data races
Go	Goroutines + channels	"Don't communicate by sharing; share by communicating"
Erlang	Actor model	Lightweight processes; no shared state at all
JavaScript	Event loop + Promises	Single-threaded; async I/O via callbacks
Haskell	STM (Software Transactional Memory)	Composable atomic transactions

- 💡 The shared-memory model (Java, C) is the lowest-level approach.
- 💡 Higher-level models (actors, channels, STM) try to make concurrency safer by design.

The Functional Perspective

❓ Why does immutability help with concurrency?

Mutable: need synchronization

```java
// Shared mutable list — danger!
List<String> shared = new ArrayList<>();

// Thread 1
shared.add("hello");

// Thread 2
shared.add("world");

// → race condition!
```

Immutable: freely shareable

```scala
// Immutable list — no danger!
val shared = List("hello", "world")

// Any thread can read — no locks
// No thread can modify — no races
```

- 💡 Immutable data is inherently thread-safe.
- 💡 This is why functional programming and concurrency are such natural partners.
- ❗ Recall: the aliasing + mutation combination is the root of most concurrency bugs.

Exercise: Spot the Bug

❓ What's wrong with this code?

```
1 public class LazyInit {
2     private static Map<String, String> cache;
3
4     public static String get(String key) {
5         if (cache == null) {
6             cache = new HashMap<>();
7             cache.put("greeting", "hello");
8         }
9         return cache.get(key);
10    }
11 }
```

- Two threads call `get()` at the same time
- Both see `cache == null`
- Both create a new `HashMap`
- One thread's map (and its entries) is lost
- ❗ This is the **check-then-act** race condition — extremely common!

Exercise: Spot the Bug (2)

❓ Is this fix correct?

```
1 public class LazyInit {
2     private static volatile Map<String, String> cache;
3
4     public static String get(String key) {
5         if (cache == null) {
6             synchronized (LazyInit.class) {
7                 if (cache == null) {       // double-checked locking
8                     Map<String, String> temp = new HashMap<>();
9                     temp.put("greeting", "hello");
10                    cache = temp;
11                }
12            }
13        }
14        return cache.get(key);
15    }
16 }
```

- 💡 **Double-checked locking** — fast path avoids synchronization.
- ❗ Only correct with `volatile`! Without it, the JMM allows reordering that lets another thread see a partially constructed map.
- ❗ Getting concurrency right is *hard*. Prefer higher-level abstractions when possible.

Design Advice

💡 Practical guidelines for concurrent programs:

1. **Prefer immutability** — immutable objects are always thread-safe
2. **Minimize shared mutable state** — the less sharing, the fewer bugs
3. **Use high-level abstractions** — `ExecutorService`, `ConcurrentHashMap`, `BlockingQueue` over raw `synchronized`
4. **Don't reinvent the wheel** — `java.util.concurrent` exists for a reason
5. **Make state ownership clear** — who is responsible for this data?

💡 Notice how guideline #5 echoes Rust's ownership model!

❗ Java gives you the *tools* to write safe concurrent code. Rust gives you *guarantees*.

Summary

- **Threads** share memory — this is powerful but dangerous.
- **Race conditions** arise from unsynchronized access to shared mutable state.
- `synchronized` and **locks** enforce mutual exclusion, but introduce deadlock risks.
- `volatile` ensures visibility between threads, but not atomicity.
- **Atomic classes** provide lock-free thread-safe operations for simple cases.
- **The Java Memory Model** defines happens-before rules that govern what threads can see.
- **Higher-level abstractions** (`ExecutorService`, `CompletableFuture`, `BlockingQueue`) reduce error-prone manual synchronization.
- **Virtual threads** (Java 21) make thread-per-task practical.
- **Immutability** is the simplest path to thread safety — functional programming and concurrency are natural partners.
- The concurrency landscape ranges from shared memory (Java/C) to message passing (Go/Erlang) to compile-time enforcement (Rust).