Class-18 : Thread Safety and Concurrency

by Pial Kanti Samadder

Thread safety

"An object is thread-safe if it behaves correctly when accessed by multiple threads, without requiring additional synchronization from the client code."

In Layman's Terms:

- Write code **as if single-threaded**
- No failures even if many threads run simultaneously
- Correct results every time

What is Thread Unsafe

- When multiple threads access an object:
  - One updates it
  - Another reads it at the same time
    → Result may be **incorrect**
- **Example**
  - Counter increments but final value is wrong
- **Fix**
  - Use `synchronized` or other mechanisms to ensure safe access

Race Condition

A **race condition** occurs when two or more threads access **shared data concurrently**, and the **final outcome depends on the timing** of thread execution.

- Happens when **operations are not atomic**.
- Can lead to **inconsistent or incorrect results**.
- Hard to detect because it may not happen every time—it depends on thread scheduling.

```
public class Counter {
    private int count = 0;

    public void increment() {
        count++;
    }

    public int getCount() {
        return count;
    }
}

public class RaceConditionDemo {
    public static void main(String[] args) throws InterruptedException {
        Counter counter = new Counter();

        Runnable counterTask = () -> {
            for (int i = 0; i < 1000; i++) {
                counter.increment();
            }
        };

        Thread t1 = new Thread(counterTask);
        Thread t2 = new Thread(counterTask);

        t1.start();
        t2.start();

        t1.join();
        t2.join();

        System.out.println("Expected: 2000");
        System.out.println("Actual: " + counter.getCount());
    }
}
```

Why It Happens

Why:

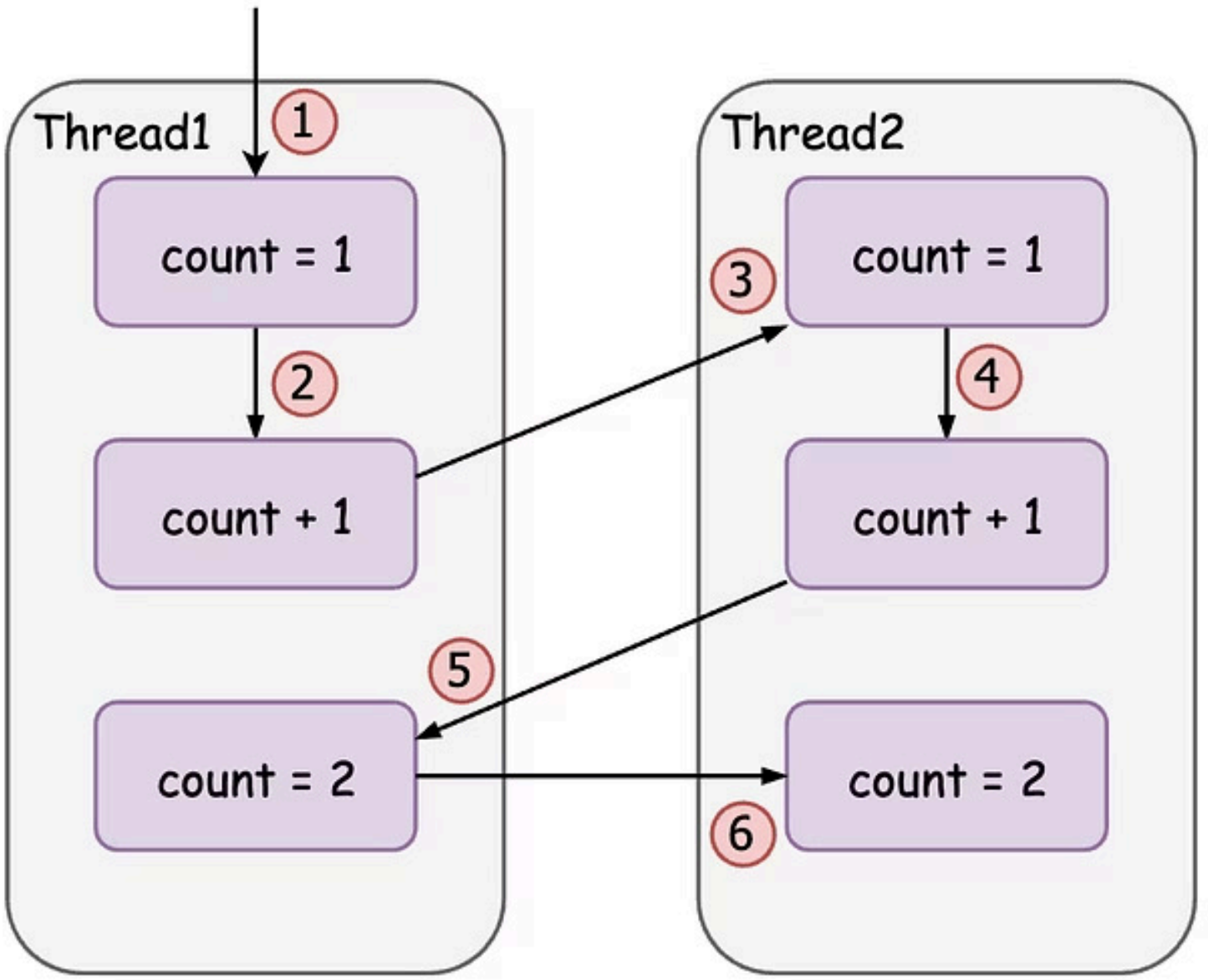
- `count++` is **not atomic**.
- Both threads may **read, increment, and write the same value simultaneously**, overwriting each other.
- Leads to **lost updates** and **inconsistent results**.

Key Takeaway:

- Race conditions occur when **shared mutable state** is accessed concurrently **without proper synchronization**.

Fix:

- Use `synchronized` methods/blocks.
- Use **AtomicInteger** or other thread-safe techniques.



`AtomicInteger` in Java

- A class in `java.util.concurrent.atomic` for **atomic operations** on a single `int`.
- Operations are **indivisible**, safe from interference by other threads.
- Crucial when multiple threads update the same integer concurrently.

Why Use `AtomicInteger`?

- Prevents **inconsistent or incorrect results** in concurrent updates.
- More efficient than `synchronized` or locks—**non-blocking thread safety**.
- Uses **Compare-And-Swap (CAS)** at the hardware level for atomic updates.

How It Works (CAS – Compare And Swap):

- **CAS is the core mechanism behind `AtomicInteger`.**
- Steps:
  a. Read the current value.
  b. Compute the new value.
  c. Atomically compare the current value with the expected old value:
    - If it matches, update it to the new value.
    - If it doesn't, retry.
- This **lock-free mechanism** ensures **atomicity** even under high concurrency.

Updated Code with `AtomicInteger`

```
public class Counter {
    private AtomicInteger count = new AtomicInteger(0);

    // Thread-safe increment
    public void increment() {
        count.incrementAndGet(); // Atomic increment
    }

    public int getCount() {
        return count.get();
    }
}
```

Thread Safety with `synchronized`

- `synchronized` is a **keyword in Java**.
- Ensures **mutual exclusion**, allowing **only one thread at a time** to execute a block of code or method.

Why Use `synchronized`?

- Prevents **race conditions** when multiple threads access **shared data**.
- Ensures **consistent and correct results** in a multi-threaded environment.

In Java, **synchronized comes in two main types**:

1. Synchronized method
2. Synchronized block

Synchronized Method

- Locks the **entire method** for a given object.
- Only one thread can execute the method on the same object at a time.

Updated Code with Synchronized Method

```
public class Counter {
    private int count = 0;

    public synchronized void increment() {
        count++; // Thread-safe
    }

    public synchronized int getCount() {
        return count;
    }
}
```

Synchronized Block

- Locks **only a portion of code** instead of the whole method.
- Can lock on **any object** (not necessarily `this`).
- Useful for **fine-grained synchronization** to improve performance.

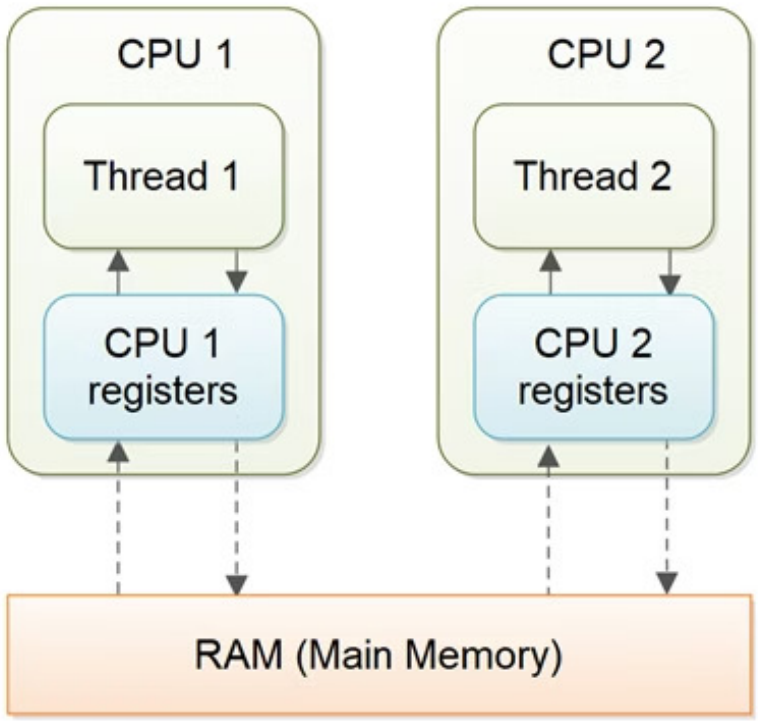
Updated Code with Synchronized Block

```
public class Counter {
    private int count = 0;
    private final Object lock = new Object(); // Lock object

    // Thread-safe increment using synchronized block
    public void increment() {
        synchronized (lock) {
            count++;
        }
    }

    // Thread-safe getter using synchronized block
    public int getCount() {
        synchronized (lock) {
            return count;
        }
    }
}
```

Variable Visibility Problems in Java

In a multithreaded application where the threads operate on non-volatile variables, each thread may copy variables from main memory into a CPU registers while working on them, for performance reasons.

If your computer **contains more than one CPU**, each thread may run on a different CPU. That means, that **each thread may copy the variables into the CPU registers of different CPUs**. This is illustrated here:



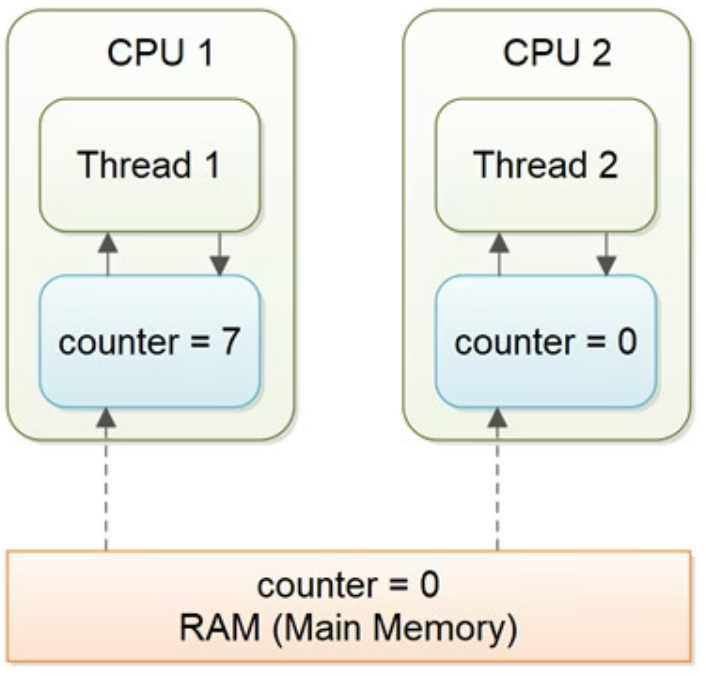
With non-volatile variables there are no guarantees about when the Java Virtual Machine (JVM) reads data from main memory into CPU registers, or writes data from CPU registers to main memory.

Imagine a situation in which two or more threads have access to a shared object which contains a counter variable declared like this:

```
public class SharedObject {
    public int counter = 0;
}
```

Imagine too, that only Thread 1 increments the `counter` variable, but both Thread 1 and Thread 2 may read the `counter` variable from time to time.

If the `counter` variable is not declared `volatile` there is no guarantee about when the value of the `counter` variable is written from the CPU registers back to main memory. This means, that the `counter` variable value in the CPU register may not be the same as in main memory. This situation is illustrated here:



The problem with threads not seeing the latest value of a variable because it has not yet been written back to main memory by another thread, is called a **"visibility" problem**. The updates of one thread are not visible to other threads.

`volatile` Keyword

- Marks a variable as **stored in main memory**, not just CPU registers.
- Ensures **every read** comes from main memory, and **every write** goes to main memory.
- Guarantees **visibility of changes across threads**.

Why Use `volatile`?

- In multi-threaded programs, **threads may cache variables in CPU registers**, leading to **stale or inconsistent values**.
- Non-volatile variables **may not reflect the latest updates** made by other threads.
- Declaring a variable `volatile` **solves visibility problems**: updates made by one thread are immediately visible to others.

Example:

```
public class SharedObject {
   public volatile int counter = 0;
}
```

- Thread 1 updates `counter`.
- Thread 2 reads `counter`.
- With `volatile`, **Thread 2 sees the latest value immediately**.

Important Notes:

- `volatile` **does not guarantee atomicity**: compound operations like `counter++` are **still not thread-safe**.
- Suitable for **flags, status indicators, or simple shared variables** where **only visibility matters**.

Executors & Thread Pools in Java

- In Java, creating threads manually with `new Thread()` can be **inefficient and hard to manage**.
- **Executors and Thread Pools** provide a **high-level API** to manage threads efficiently.
- Benefits:
  - Reuse threads instead of creating new ones every time.
  - Better resource management and performance.
  - Simplifies thread life-cycle management.

Executor Framework

- **Executor** is an **interface** that represents an object that executes submitted tasks.
- Tasks are represented as **Runnable** or **Callable** objects.
- Example:

```
Executor executor = Executors.newFixedThreadPool(2);
executor.execute(() -> System.out.println("Task executed"));
```

- Advantages:
  - Decouples task submission from thread creation.
  - Provides a **central place to manage concurrency**.

Thread Pools

- A **thread pool** is a collection of worker threads waiting for tasks.
- Submitted tasks are **queued** and executed by available threads.
- **Types of Thread Pools:**
  a. **Fixed Thread Pool** – fixed number of threads.
  b. **Cached Thread Pool** – dynamically grows and shrinks.
  c. **Single Thread Executor** – only one thread executes tasks sequentially.
  d. **Scheduled Thread Pool** – executes tasks periodically or after a delay.

Example – Fixed Thread Pool

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class ThreadPoolDemo {
    public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(3);

        for (int i = 1; i <= 5; i++) {
            int taskId = i;
            executor.execute(() -> {
                System.out.println("Task " + taskId + " is running by " + Thread.currentThread().getName());
            });
        }

        executor.shutdown();
    }
}
```

Observation:

- Only 3 threads are created.
- Tasks are executed as threads become available.
- Reuses threads from the pool instead of creating new ones.

Virtual Threads

Java virtual threads are a new thread construct added to Java from Java 19. *Java virtual threads* are different from the original platform threads in that virtual threads are much more lightweight in terms of how many resources (RAM) they demand from the system to run.

What is a Platform Thread?

A *platform thread* is implemented as a thin wrapper around an operating system (OS) thread. A platform thread runs Java code on its underlying OS thread, and the platform thread captures its OS thread for the platform thread's entire lifetime. Consequently, the number of available platform threads is limited to the number of OS threads.

Platform threads typically have a large thread stack and other resources that are maintained by the operating system. They are suitable for running all types of tasks but may be a limited resource.

What is a Virtual Thread?

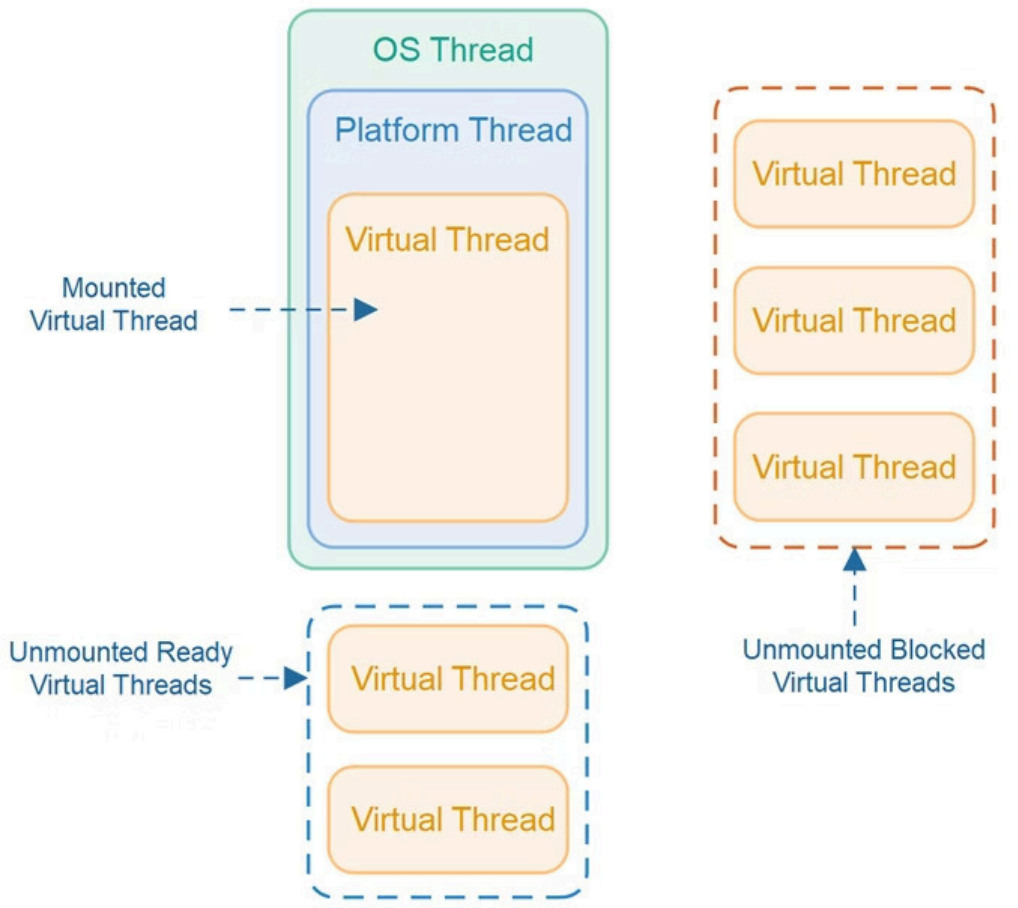
Like a platform thread, a *virtual thread* is also an instance of `java.lang.Thread`. However, **a virtual thread isn't tied to a specific OS thread**. A virtual thread still runs code on an OS thread. However, when code running in a virtual thread calls a blocking I/O operation, the Java runtime suspends the virtual thread until it can be resumed. The OS thread associated with the suspended virtual thread is now free to perform operations for other virtual threads.

Virtual threads are implemented in a similar way to virtual memory. To simulate a lot of memory, an operating system maps a large virtual address space to a limited amount of RAM. Similarly, to simulate a lot of threads, the Java runtime maps a large number of virtual threads to a small number of OS threads.

Virtual threads are suitable for running tasks that spend most of the time blocked, often waiting for I/O operations to complete. However, they aren't intended for long-running CPU-intensive operations.

Java Virtual Thread Diagram

Here is a diagram showing Java virtual threads - executed by platform threads - which are again executed by OS threads. While a platform thread can only execute a single virtual thread at a time, it has the ability to switch to executing a different virtual thread when the currently executed virtual thread makes a blocking call (e.g. network or concurrency data structure). This is explained in more detail in later sections of this Java virtual thread tutorial.



Virtual Threads are Mounted to Platform Threads

Java virtual threads are executed by platform threads. A platform thread can only execute one virtual thread at a time. While the virtual thread is being executed by a platform thread - the virtual thread is said to be *mounted* to that thread.

New virtual threads are queued up until a platform thread is ready to execute it. When a platform thread becomes ready, it will take a virtual thread and start executing it.

A virtual thread that executes some blocking network call (IO) will be *unmounted* from the platform thread while waiting for the response. In the meantime the platform thread can execute another virtual thread.

No Time Slicing Between Virtual Threads

There is no time slicing happening between virtual threads. In other words, the platform thread does not switch between executing multiple virtual threads - except in the case of blocking network calls. As long as a virtual thread is running code and is not blocked waiting for a network response - the platform thread will keep executing the same virtual thread.

Creating a Java Virtual Thread

To create a new virtual thread in Java, you use the new Thread.ofVirtual() factory method, passing an implementation of the **Runnable interface**. Here is an example of creating a Java virtual thread:

```
Runnable runnable = () -> {
    for(int i=0; i<10; i++) {
        System.out.println("Index: " + i);
    }
};

Thread vThread = Thread.ofVirtual().start(runnable);
vThread.join();
```

This example creates a virtual thread and starts it immediately, executing the Runnable passed to the `start()` method.

If you do not want the virtual thread to start immediately, you can use the unstarted() method instead. Here is an example of creating an unstarted virtual thread:

```
Thread vThreadUnstarted = Thread.ofVirtual().unstarted(runnable);
```

To start an unstarted virtual thread you just call the start() method on it, like this:

```
vThreadUnstarted.start();
```

ExecutorService Using Virtual Threads

It is possible to create a Java `ExecutorService` that uses virtual threads internally. Example:

```
try (ExecutorService executor = Executors.newVirtualThreadPerTaskExecutor()) {
    for (int i = 1; i <= 5; i++) {
        int taskId = i;
        executor.submit(() -> System.out.println("Task " + taskId + " running in " + Thread.currentThread()));
    }
}
```