Synchronisation

Josh Wilcox (jw14g24)

March 18, 2025

Table of Contents

JVM Need for Syncing

- Each thread in Java has it's own stack in the JVM
- However each thread shares the same heap
  - Any fields declared in an object may be accessible to a number of different threads at one time
  - This can cause collisions in updating and accessing variables which is bad
  - (E.g. When one thread is trying to read a value from a field, another could be trying to change that value)
- This leads to some undesired behaviour

Race Condition

- A race condition occurs when two or more threads can access shared data and they try to change it at the same time.
- Because the thread scheduling algorithm can swap between threads at any time, you don't know the order in which the threads will attempt to access the shared data.
- Therefore, the result of the change in data is dependent on the thread scheduling algorithm, which can lead to inconsistent results.
- Race conditions can cause unpredictable behavior and bugs that are hard to reproduce and debug.

See the next page for an example

Race Condition - **Example**

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

    public void increment() {
        count++;
    }

    public int getCount() {
        return count;
    }
}

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

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

        Thread thread1 = new Thread(task);
        Thread thread2 = new Thread(task);

        thread1.start();
        thread2.start();

        try {
            thread1.join();
            thread2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("Final count: " + counter.getCount());
    }
}
```

- `count++` will be split into three parts:
  - **iload** - Loads z from heap and pushes it to stack
  - **iAdd** - Increments the value at the top of the stack
  - **iStore** - Pops the value at the top of the stack to store it in z
- The scheduler can execute these at any time in the execution - which can cause shared variable problems

Synchronisation

- Operations can be **locked** - meaning that if a thread is accessing this operation *no other thread can get in*
- To implement synchronisation in Java, we can use:
  - Synchronized block
  - The `synchronized` keyword

Synchronised Block Example


```java
/**
 * This class demonstrates synchronization using a synchronized block.
 * Synchronized blocks ensure thread safety by allowing only one thread
 * to execute the critical section at a time.
 */
public class TicketRunnable implements Runnable {
    // Shared resource that multiple threads will try to access
    private int ticket = 5; // Initial ticket count

    public void run() {
        while (true) {
            // Synchronized block with 'this' as the monitor object (lock)
            // Only one thread can hold this lock at a time
            // Other threads trying to enter this block will wait until the lock is released
            synchronized (this) {
                // Critical section - code that can cause race conditions if not synchronized
                if (ticket > 0) {
                    // Displays the current thread and decrements the ticket count atomically
                    // The decrement operation (ticket--) is protected within the synchronized block
                    System.out.println(Thread.currentThread().getName() + ": ticket = " + ticket--);
                } else {
                    // No more tickets available, thread exits the loop
                    System.out.println(Thread.currentThread().getName() + ": no more tickets!");
                    break;
                }
            } // Lock is released here when the block ends

            try {
                // Sleep is outside the synchronized block
                // This is good practice - don't hold locks during long operations
                Thread.sleep(100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    public static void main(String[] args) {
        // Creating a single instance of TicketRunnable to be shared among threads
        TicketRunnable tr = new TicketRunnable();

        // Creating three threads that share the same TicketRunnable instance
        // Each thread will attempt to access the synchronized block independently
        new Thread(tr).start();
        new Thread(tr).start();
        new Thread(tr).start();
        // Without synchronization, we'd have race conditions on the ticket variable
    }
}
```

Synchronised Keyword Example


```java
/**
 * This class demonstrates synchronization using the synchronized method keyword.
 * When a method is declared as synchronized, it acquires a lock on the entire object
 * before execution, ensuring thread-safe access to shared resources.
 */
public class TicketThread implements Runnable {
    // Shared resource that multiple threads will try to access
    private int ticket = 5; // Initial ticket count

    public void run() {
        while (true) {
            // Calling the synchronized method from within run()
            // The thread does not hold any lock while in this part of run()
            this.sale();

            try {
                // Good practice: Sleep outside of synchronized sections
                // This gives other threads a chance to acquire the lock
                Thread.sleep(100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    /**
     * This method is declared as synchronized, which means:
     * 1. Only one thread can execute this method at a time
     * 2. The lock is on 'this' object (implicit)
     * 3. The entire method body becomes a critical section
     *
     * Equivalent to: synchronized(this) { method body }
     */
    public synchronized void sale() {
        // Critical section - protected by synchronization
        if (ticket > 0) {
            try {
                // Even inside a synchronized method, a Thread.sleep()
                // does NOT release the lock - other threads still cannot enter
                // this method until this thread completely exits the method
                Thread.sleep(200);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

            // The read and write operations on ticket are atomic
            // due to the synchronized method
            System.out.println(Thread.currentThread().getName() + ": ticket = " + ticket--);
        } else {
            // No more tickets, but notice there's no break statement
            // Threads will keep entering this method even when tickets are gone
            System.out.println(Thread.currentThread().getName() + ": no more tickets!");
        }
        // Lock is automatically released when method ends
    }

    public static void main(String[] args) {
        // Creating a single instance of TicketThread to be shared among threads
        TicketThread tt = new TicketThread();

        // Creating three threads with names that share the same TicketThread instance
        Thread t1 = new Thread(tt, "Window 1");
        Thread t2 = new Thread(tt, "Window 2");
        Thread t3 = new Thread(tt, "Window 3");

        // Starting all three threads - they will compete for the synchronized method's lock
        t1.start();
        t2.start();
        t3.start();
        // The synchronized method ensures that ticket sales are thread-safe
    }
}
```

Aims of Synchronisation

- Allows for **mutual exclusion** of certain points of code
  - Only one thread at a time executes a certain portion of code
- This means access to shared variables can be made unproblematic

OverSynchronising

- Placing instructions in a synced block makes their execution **atomic**
  - Means the execution of these intstructions is insensitive to the scheduling order of threads
- This can be very inneficient
  - Causes other threads to block causing heavy runtime overheads
- This means, generally, synchronisation shouldbe **avoided** unless completely necessary and safe
  - For example, only synchronise lines that have direct access to shared variables

Using Different Locks

- So far, we have only used the `this` object as the *locking object*
  - In general you can use any object as a lock
- Suppose that you want to protect two shared variables A and B so that their get/set methods are independent but safe. Suppose also that you want to provide a 'setBoth' method which allows both variables to be set together.

```
1  public class TwoSharedVars {
2      private int A = 10;
3      private int B = 20;
4      private Object lockA = new Object();
5      private Object lockB = new Object();
6
7      public int getA() {
8          synchronized (lockA) { return A; }
9      }
10     public void setA(int v) {
11         synchronized (lockA) { A = v; }
12     }
13     public int getB() {
14         synchronized (lockB) { return B; }
15     }
16     public void setB(int v) {
17         synchronized (lockB) { B = v; }
18     }
19     public void setBoth(int v, int w) {
20         synchronized (lockA) {
21             synchronized (lockB) { A = v; B = w; }
22         }
23     }
24 }
```

When to Lock

- Always lock during updates to an objects **shared fields**
- Always lock during access of possibly updated fields
- Never lock when invoking methods on other objects
  - Can lead to **deadlocks**

Deadlocks

- Happens when a thread holds a lock but is waiting for another lock
- This other lock is being heald by another thread which is waiting for another lock
- ⋮
- This other lock is being heald by another thread which is waiting for the lock being held by the *first* thread
- All threads end up waiting for each other and none can proceed
  - Like a train deadlock in factorio