12.5: Race Condition

Introduction:

Threads play an essential role in multitasking and concurrent execution in programming. As we've discussed, you can either manually create threads or rely on frameworks to provide them. Although creating threads manually is not always common, sometimes it's necessary, especially when you want a specific task to execute concurrently to improve performance.

We can create threads to handle tasks that require rapid completion, depending on the requirements. In such cases, we might manually create two or more threads at the same time. However, this introduces some challenges, such as synchronization issues and race conditions.

Understanding Mutation:

Mutation, in simple terms, means changing the state or value of something. In programming, mutation occurs when a variable's value is altered during the execution.

For example, consider a variable i that has a value of 10. If we change it to 12, it is referred to as a mutation of that variable. In the context of threads, a race condition arises when two or more threads access and change the same variable simultaneously.

This results in **inconsistent values** due to unpredictable behavior, making the scenario unstable.

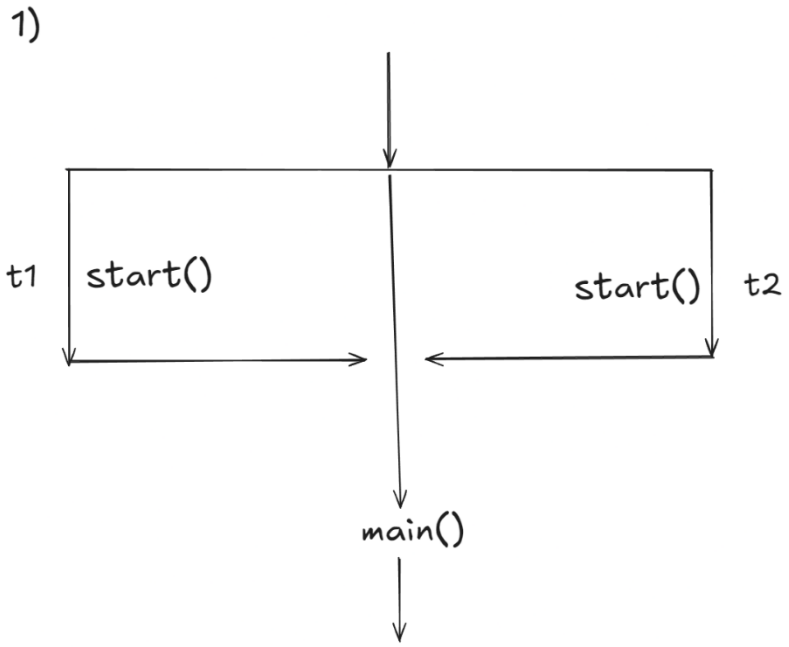
Let's illustrate this with a simple analogy:

Example:

Suppose you are going to an ATM to withdraw money. Imagine there is only one entrance to the ATM booth, and only one person can enter at a time. Now, if you and another person arrive at the same time and try to enter, what would happen? You won't both be able to get in simultaneously. This situation mirrors the behavior of threads when they attempt to access the same resource simultaneously, causing a conflict.

This is why, when working with threads, we should avoid using **mutable variables**. Alternatively, we can use **immutable data** or ensure that the methods manipulating shared data are **thread-safe**.

Understanding the join() Method:



The **join()** method ensures that the main thread waits for the other threads (t1 and t2) to finish execution before proceeding. This is necessary to ensure that the final value of count is printed after both threads have completed their tasks.

Fig(1).without implementation of join() method

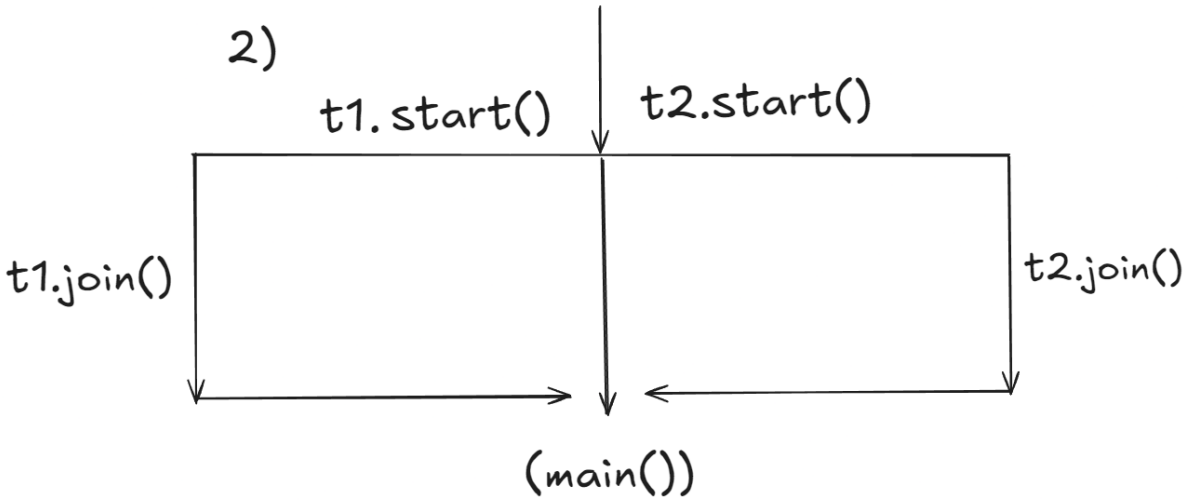


Fig. 2 with implementation of join() method

What is Thread Safety?

Thread safety means ensuring that a method or a block of code is executed by only one thread at a time. This prevents conflicts and inconsistencies. When a

method is thread-safe, it ensures that no other thread can access or modify the shared resource while it's in use by one thread.

Let's explore this further using an example.

Example without Thread Safety:

```
class Counter {
    int count;

    public void increment() {
        count++;
    }
}

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

        Runnable obj1 = () -> {
            for(int i = 1; i <= 1000; i++) {
                c.increment();
            }
        };

        Runnable obj2 = () -> {
            for(int i = 1; i <= 1000; i++) {
                c.increment();
            }
        };

        Thread t1 = new Thread(obj1);
        Thread t2 = new Thread(obj2);

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

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

        System.out.println(c.count);
    }
}
```

Output (varies each time the program is executed):

- 1750
- 1985
- 2000
- 1905

Explanation:

- In the above program, we created a Counter class with an increment() method that increases the value of the variable count.
- We then created two threads (t1 and t2) that execute the increment() method 1000 times each.
- Ideally, the output should be 2000, but due to race conditions, the threads interfere with each other, resulting in an inconsistent count value.

The inconsistency occurs because both threads access the shared variable count simultaneously. To solve this issue, we need to make the increment() method **thread-safe** using the synchronized keyword.

Example with Thread Safety:

```
class Counter {
    int count;

    public synchronized void increment() {
        count++;
    }
}

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

        Runnable obj1 = () -> {
            for(int i = 1; i <= 1000; i++) {
                c.increment();
            }
        };

        Runnable obj2 = () -> {
            for(int i = 1; i <= 1000; i++) {
```

```
                c.increment();
            }
        };

        Thread t1 = new Thread(obj1);
        Thread t2 = new Thread(obj2);

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

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

        System.out.println(c.count);
    }
}
```

Output:

- 2000 (Consistent output)

Explanation:

- By using the synchronized keyword on the increment() method, we ensure that only one thread can access this method at a time.
- This prevents the race condition and results in a consistent value of 2000.

Why Do Race Conditions Happen?

Race conditions happen because of the unpredictability of thread execution. If two threads access the same variable simultaneously, the outcome depends on the order in which they execute. For a small number of iterations, the problem might not be noticeable, but as the number of iterations increases (e.g., 10,000), the issue becomes prominent and can lead to **data loss** or **incorrect values**.

Best Practices to Avoid Race Conditions:

1. **Use the synchronized keyword:** Ensures that only one thread can access the critical section of code.
2. **Avoid shared mutable state:** Wherever possible, use immutable objects.
3. **Use Thread-safe Collections:** Use ConcurrentHashMap or other thread-safe collections instead of regular ones like HashMap.