برنامه نویسی پیشرفته

رفع اشک‌ال: جلسه۸

Threading

What is a Thread?

- A **thread** is the smallest unit of execution in a program.
- **Multithreading** = multiple threads run concurrently.

```
public class MyThread extends Thread {
    public void run() {
        System.out.println("Running in a thread");
   }
}
MyThread t = new MyThread();
t.start(); // runs on a separate thread
```

Question: What happens if you call t.run() instead of t.start()?

Question

```
Thread t1 = new Thread(() -> System.out.println("Thread 1"));
Thread t2 = new Thread(() -> System.out.println("Thread 2"));

t1.start();
t2.start();
System.out.println("Main done");
```

Output?

Answer

Possible Outputs:

Since all three lines run in separate threads, the output can appear in any order. Some possible outputs:

- Main done
  Thread 1
  Thread 2
- Main done
  Thread 1
  Thread 2
- Thread 1
  Main done
  Thread 2

or any other combination.

Why?

- t1.start() and t2.start() start two independent threads.
- System.out.println("Main done") runs in the main thread.
- The JVM thread scheduler decides when each thread actually runs.
- So even if t1.start() is written first, it may run before or after t2 or main.

Runnable Interface

Runnable is used to define thread behavior without extending Thread

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

Useful when your class already extends another class

Thread vs Runnable

Feature	Thread (extends)	Runnable (implements)
Extends Thread?	✅ Yes	❌ No
Can extend other class?	❌ No	✅ Yes
Reusability	❌ Low	✅ High
Used in Executors?	❌ Rare	✅ Common

Why synchronized ?

- To prevent multiple threads from accessing shared data at the same time.

```
public class Counter {
    private int count = 0;
    public synchronized void increment() {
        count++;
    }
}
```

Only one thread can execute increment() at a time.

Question: What if we remove synchronized?

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

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

    public synchronized void decrement() {
        count--;
    }

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

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

        Thread t1 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) {
                counter.increment();
            }
        });

        Thread t2 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) {
                counter.decrement();
            }
        });

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

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

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

Answer

What Happens When You Remove synchronized?

1. Thread 1 (calls increment()) and Thread 2 (calls decrement()) will both access the count variable at the same time.

2. Since there's no synchronization, Thread 1 and Thread 2 might both read the same value of count, update it, and then write the new value back.

3. This can lead to incorrect results because their updates are not coordinated.

It's non-deterministic. This means the output will vary each time you run the program.

What is ExecutorService?

ExecutorService is part of the java.util.concurrent package. It's a high-level API that manages a thread pool.

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
public class ThreadPoolExample {
    public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(3); // 3 threads in pool
        for (int i = 1; i <=  7; i++) {
            int taskId = i;
            executor.submit(() -> {
                System.out.println("Task " + taskId + " executed by " + Thread.currentThread().getName());
            });
        }
        executor.shutdown(); // No new tasks accepted, but waits for submitted tasks to finish
    }
}
```

ExecutorService

Possible output:

Task 1 executed by pool-1-thread-1
Task 3 executed by pool-1-thread-3
Task 5 executed by pool-1-thread-3
Task 6 executed by pool-1-thread-3
Task 7 executed by pool-1-thread-3
Task 2 executed by pool-1-thread-2
Task 4 executed by pool-1-thread-1

- We create a pool of 3 threads.
- We submit 5 tasks. Only 3 tasks can run at the same time.
- The remaining 2 wait until a thread becomes free.
- All threads are reused — no need to create/destroy threads manually.

Thread lifecycle

1. **New**

Thread object is created, but .start() hasn't been called yet.

Thread t = new Thread();

2. **Runnable**

.start() has been called.

Thread is ready to run, waiting for CPU time.

t.start();

3. 🔵 **Running**

The thread is actually executing on the CPU.

4. **Blocked / Waiting / Timed Waiting**

•Thread is temporarily not running.

5. 🔴 **Terminated / Dead**

Thread has finished running or was stopped due to an exception.

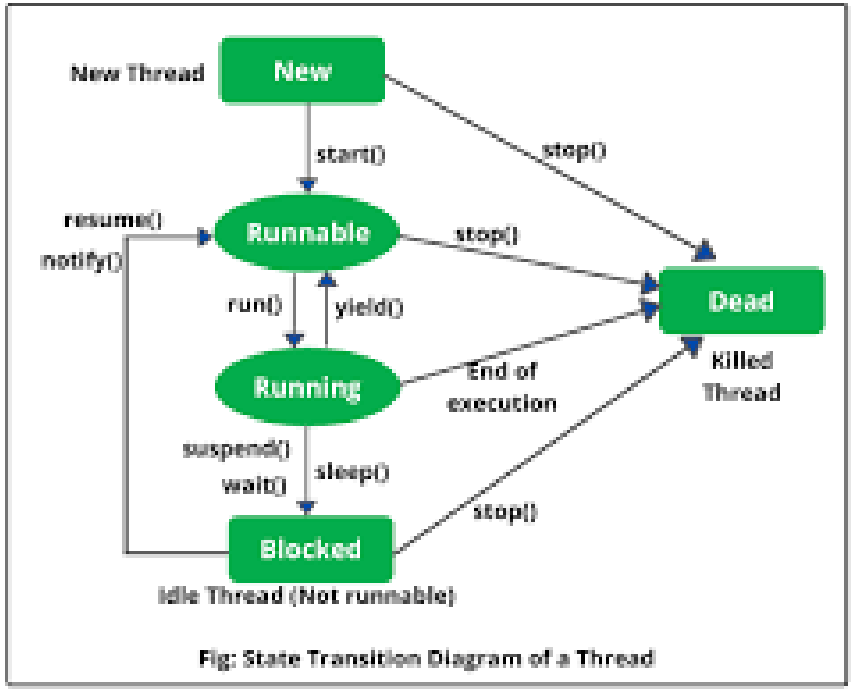

Fig: State Transition Diagram of a Thread

Code Examples

Check the code files and try to guess the outputs before running the code.