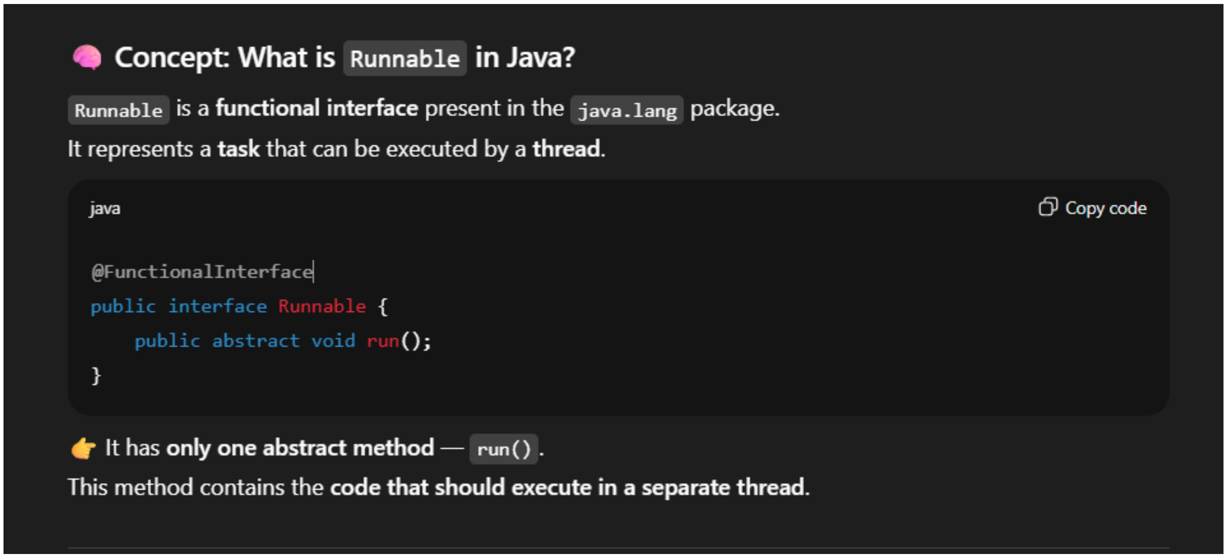
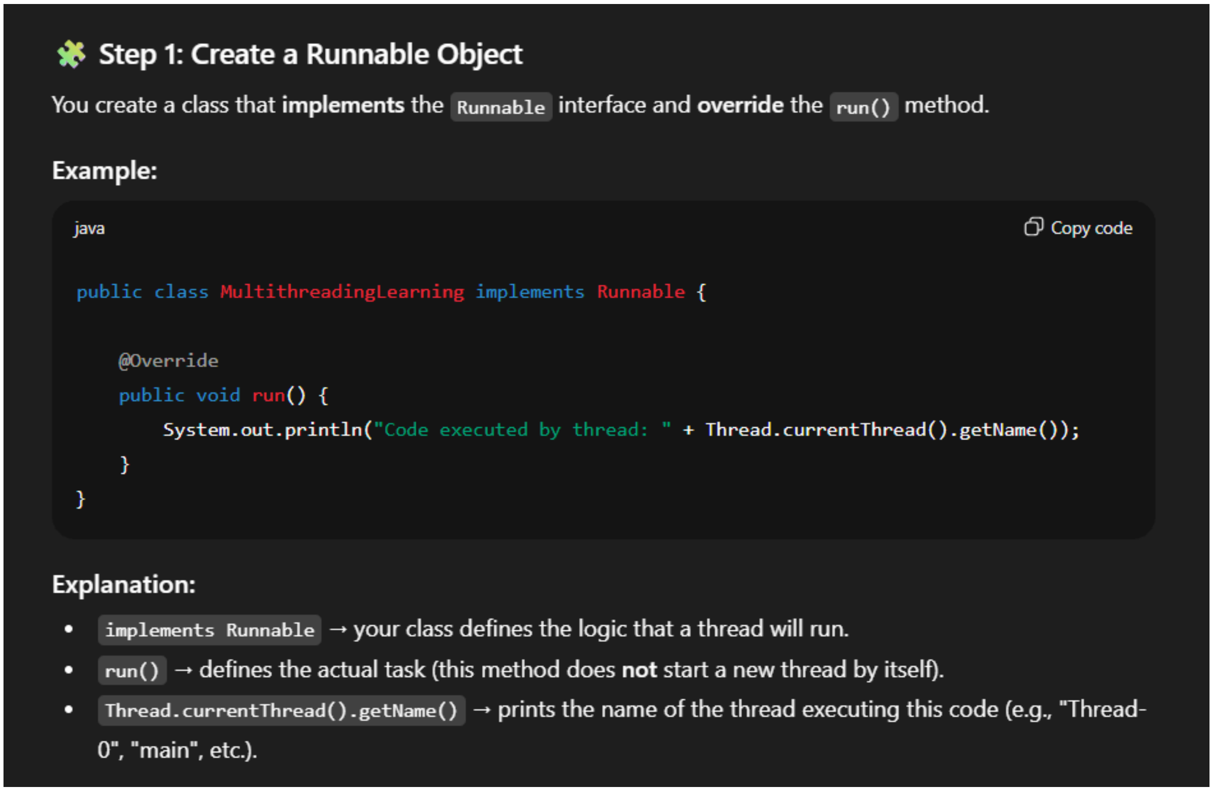


🧠 Concept: What is `Runnable` in Java?

`Runnable` is a **functional interface** present in the `java.lang` package.
It represents a **task** that can be executed by a **thread**.

```java
@FunctionalInterface
public interface Runnable {
    public abstract void run();
}
```

👉 It has **only one abstract method** — `run()`.
This method contains the **code that should execute in a separate thread**.



🧩 Step 1: Create a Runnable Object

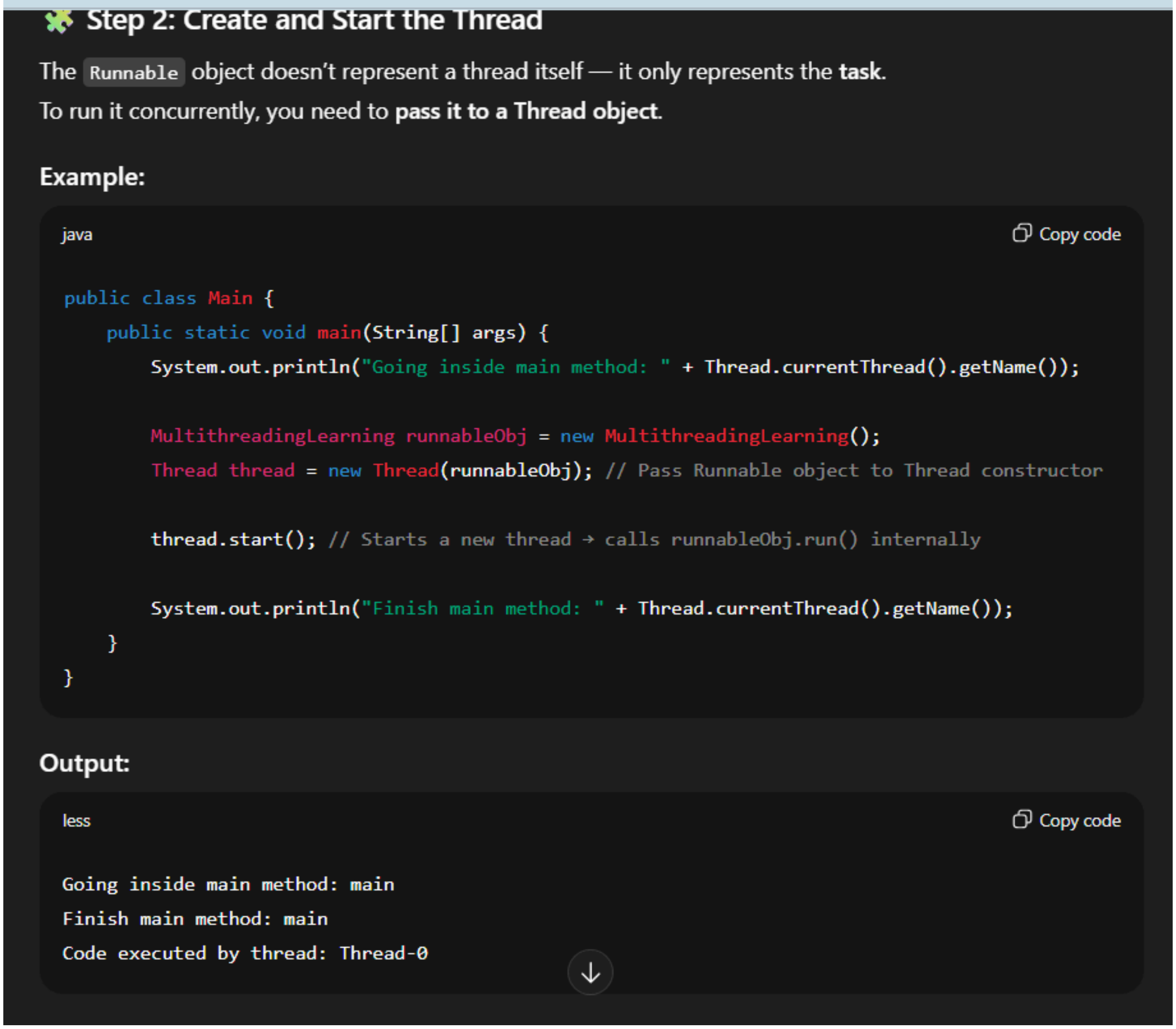
You create a class that **implements** the `Runnable` interface and **override** the `run()` method.

Example:

```java
public class MultithreadingLearning implements Runnable {

    @Override
    public void run() {
        System.out.println("Code executed by thread: " + Thread.currentThread().getName());
    }
}
```

Explanation:

- `implements Runnable` → your class defines the logic that a thread will run.
- `run()` → defines the actual task (this method does **not** start a new thread by itself).
- `Thread.currentThread().getName()` → prints the name of the thread executing this code (e.g., "Thread-0", "main", etc.).



🧩 Step 2: Create and Start the Thread

The `Runnable` object doesn't represent a thread itself — it only represents the **task**.
To run it concurrently, you need to **pass it to a Thread object**.

Example:

```java
public class Main {
    public static void main(String[] args) {
        System.out.println("Going inside main method: " + Thread.currentThread().getName());

        MultithreadingLearning runnableObj = new MultithreadingLearning();
        Thread thread = new Thread(runnableObj); // Pass Runnable object to Thread constructor

        thread.start(); // Starts a new thread → calls runnableObj.run() internally

        System.out.println("Finish main method: " + Thread.currentThread().getName());
    }
}
```

Output:

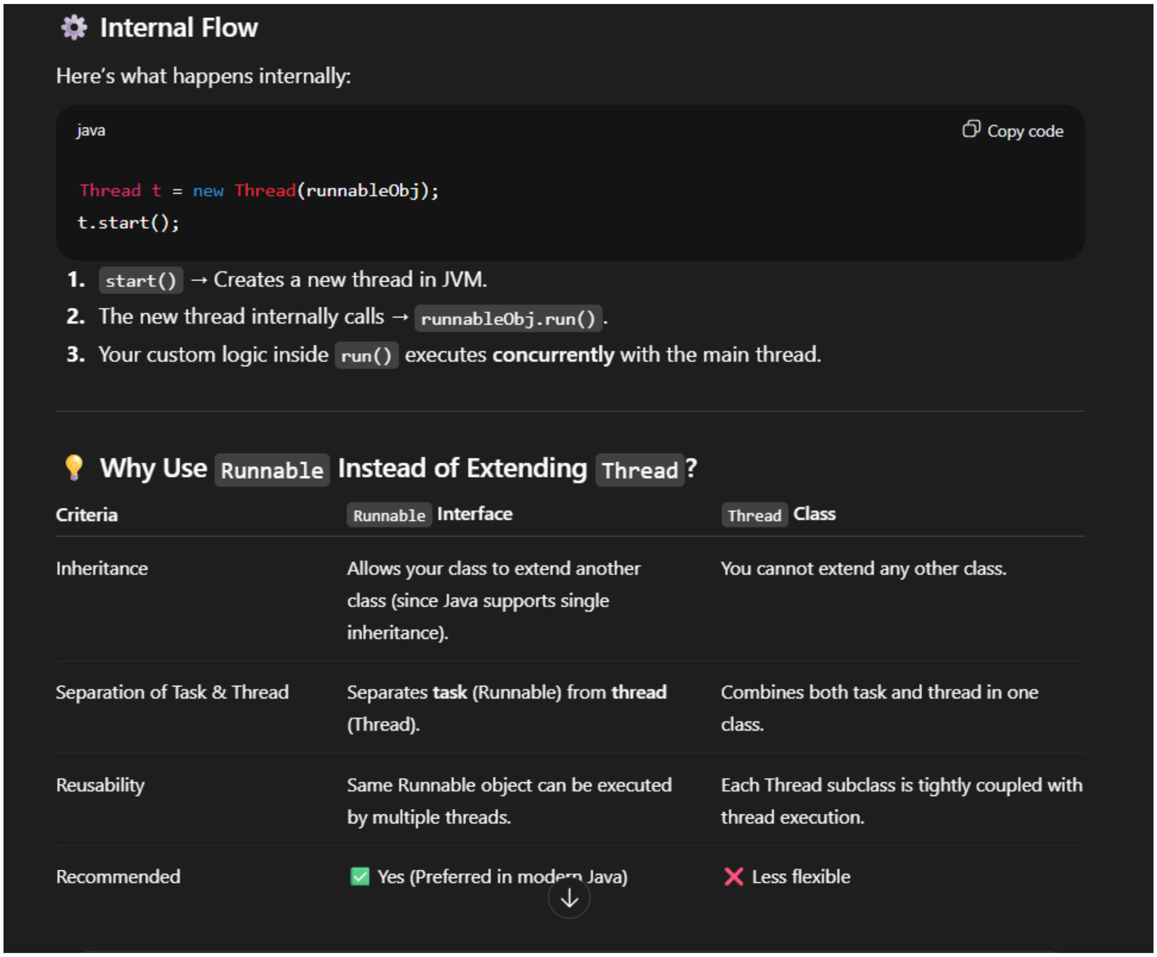
```less
Going inside main method: main
Finish main method: main
Code executed by thread: Thread-0
```

Explanation:

- `Thread thread = new Thread(runnableObj);`
    - Creates a new `Thread` object.
    - Associates your `Runnable` task (`runnableObj`) with it.
- `thread.start();`
    - Starts a **new thread**.
    - Internally, it calls `runnableObj.run()` on that new thread.

- The `main` thread continues executing independently.

Hence, the output order can vary because the main and worker threads run concurrently.



⚙️ Internal Flow

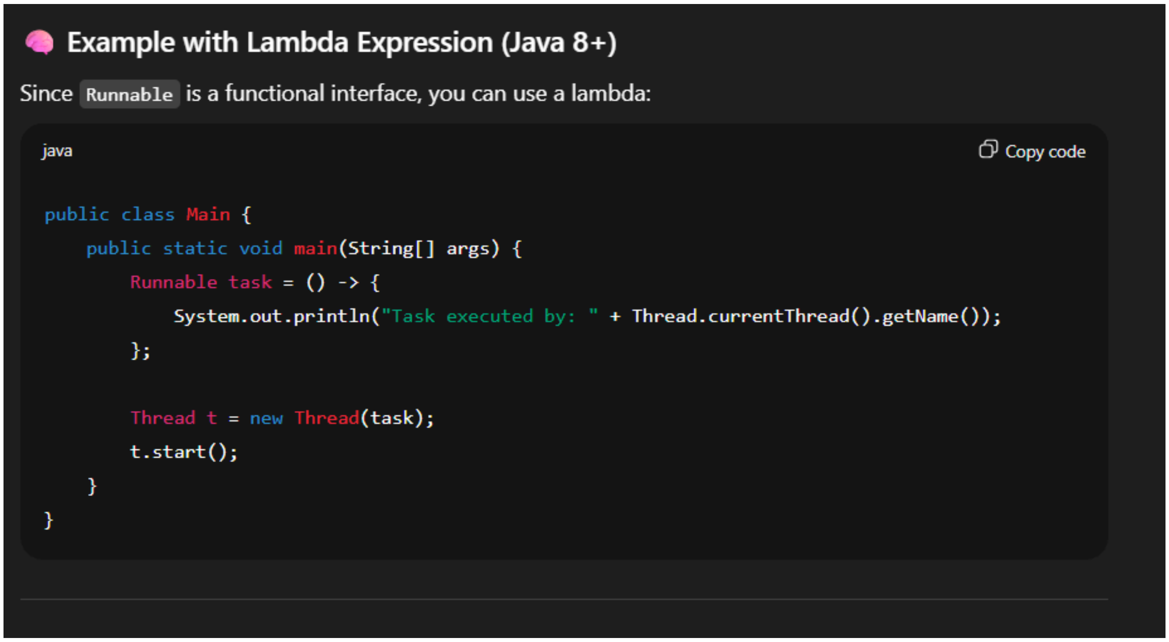
Here's what happens internally:

```java
Thread t = new Thread(runnableObj);
t.start();
```

1. `start()` → Creates a new thread in JVM.
2. The new thread internally calls → `runnableObj.run()`.
3. Your custom logic inside `run()` executes **concurrently** with the main thread.

💡 Why Use `Runnable` Instead of Extending `Thread`?

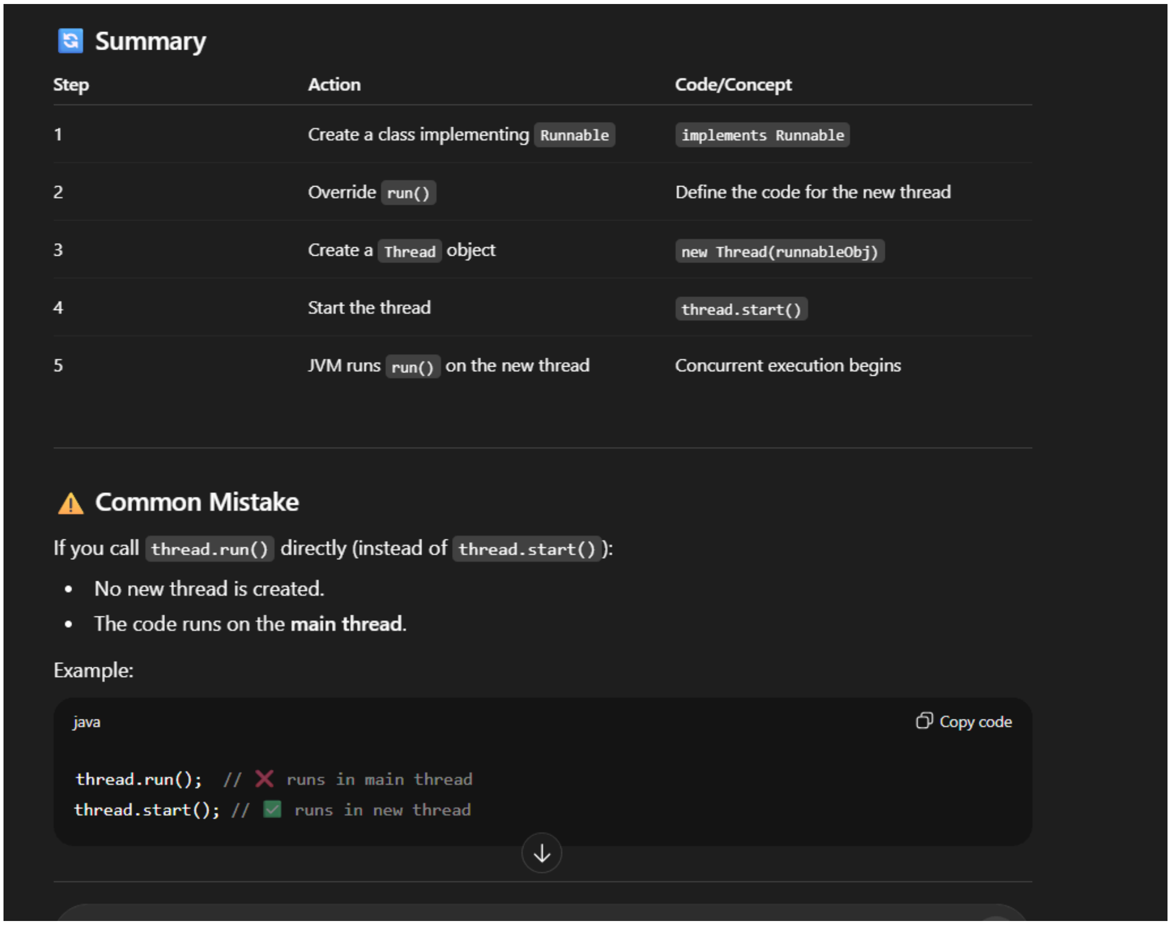
Criteria	`Runnable` Interface	`Thread` Class
Inheritance	Allows your class to extend another class (since Java supports single inheritance).	You cannot extend any other class.
Separation of Task & Thread	Separates **task** (Runnable) from **thread** (Thread).	Combines both task and thread in one class.
Reusability	Same Runnable object can be executed by multiple threads.	Each Thread subclass is tightly coupled with thread execution.
Recommended	✅ Yes (Preferred in modern Java)	❌ Less flexible



🧠 Example with Lambda Expression (Java 8+)

Since `Runnable` is a functional interface, you can use a lambda:

```java
public class Main {
    public static void main(String[] args) {
        Runnable task = () -> {
            System.out.println("Task executed by: " + Thread.currentThread().getName());
        };

        Thread t = new Thread(task);
        t.start();
    }
}
```



🔄 Summary

Step	Action	Code/Concept
1	Create a class implementing `Runnable`	`implements Runnable`
2	Override `run()`	Define the code for the new thread
3	Create a `Thread` object	`new Thread(runnableObj)`
4	Start the thread	`thread.start()`
5	JVM runs `run()` on the new thread	Concurrent execution begins

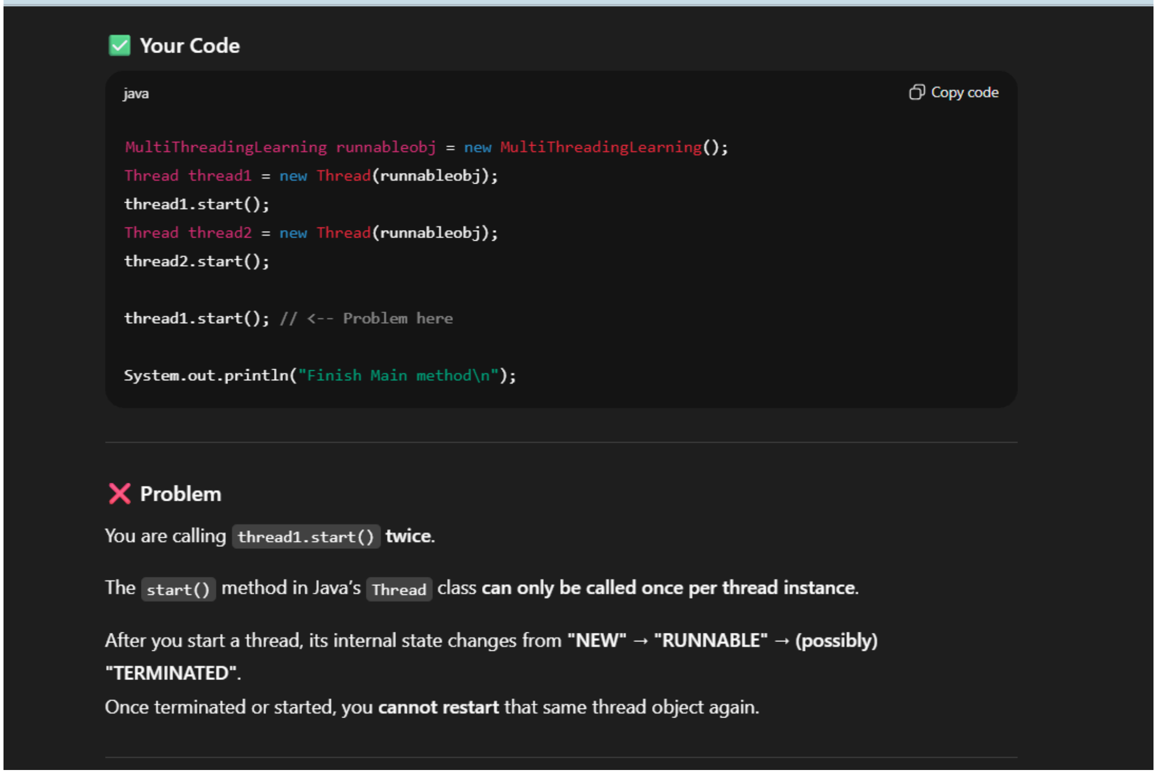
⚠️ Common Mistake

If you call `thread.run()` directly (instead of `thread.start()`):

- No new thread is created.
- The code runs on the **main thread**.

Example:

```java
thread.run();   // ❌ runs in main thread
thread.start(); // ✅ runs in new thread
```



✅ Your Code

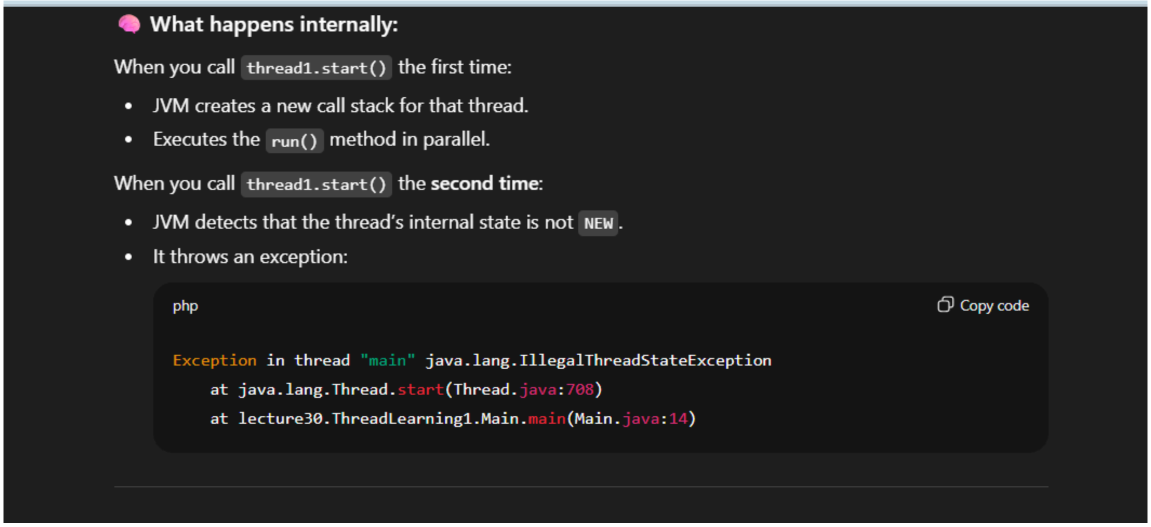
```java
MultiThreadingLearning runnableobj = new MultiThreadingLearning();
Thread thread1 = new Thread(runnableobj);
thread1.start();
Thread thread2 = new Thread(runnableobj);
thread2.start();

thread1.start(); // <-- Problem here

System.out.println("Finish Main method\n");
```

❌ Problem

You are calling `thread1.start()` **twice.**

The `start()` method in Java's `Thread` class **can only be called once per thread instance.**

After you start a thread, its internal state changes from **"NEW"** → **"RUNNABLE"** → **(possibly) "TERMINATED"**.
Once terminated or started, you **cannot restart** that same thread object again.



🧠 What happens internally:

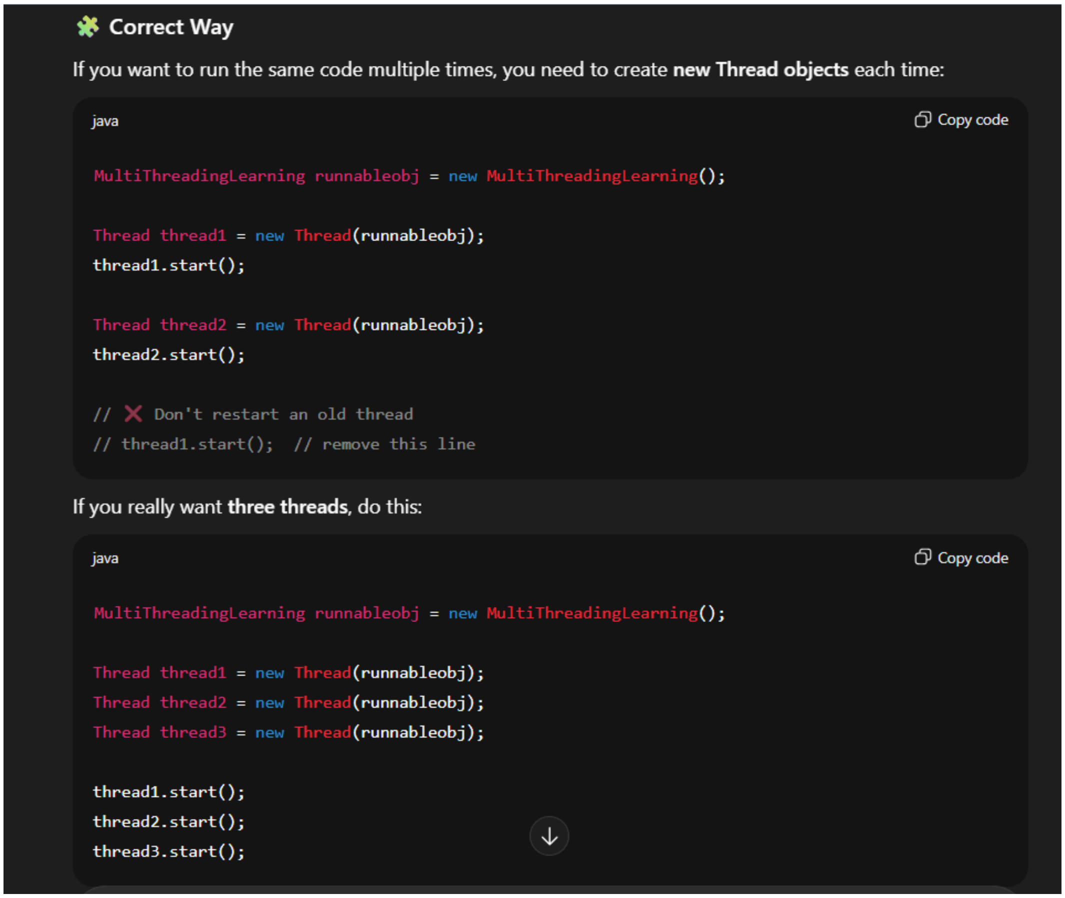
When you call `thread1.start()` the first time:

- JVM creates a new call stack for that thread.
- Executes the `run()` method in parallel.

When you call `thread1.start()` the **second time**:

- JVM detects that the thread's internal state is not `NEW`.
- It throws an exception:

```
Exception in thread "main" java.lang.IllegalThreadStateException
    at java.lang.Thread.start(Thread.java:708)
    at lecture30.ThreadLearning1.Main.main(Main.java:14)
```



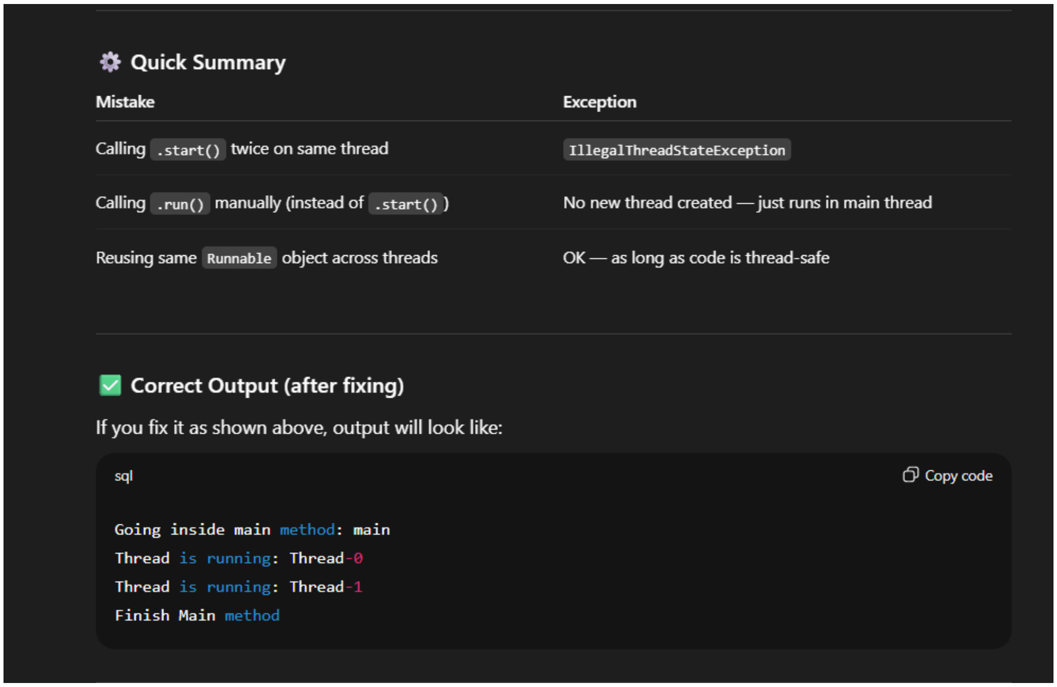
🧩 Correct Way

If you want to run the same code multiple times, you need to create **new Thread objects** each time:

```java
MultiThreadingLearning runnableobj = new MultiThreadingLearning();

Thread thread1 = new Thread(runnableobj);
thread1.start();

Thread thread2 = new Thread(runnableobj);
thread2.start();

// ❌ Don't restart an old thread
// thread1.start();  // remove this line
```

If you really want **three threads**, do this:

```java
MultiThreadingLearning runnableobj = new MultiThreadingLearning();

Thread thread1 = new Thread(runnableobj);
Thread thread2 = new Thread(runnableobj);
Thread thread3 = new Thread(runnableobj);

thread1.start();
thread2.start();
thread3.start();
```



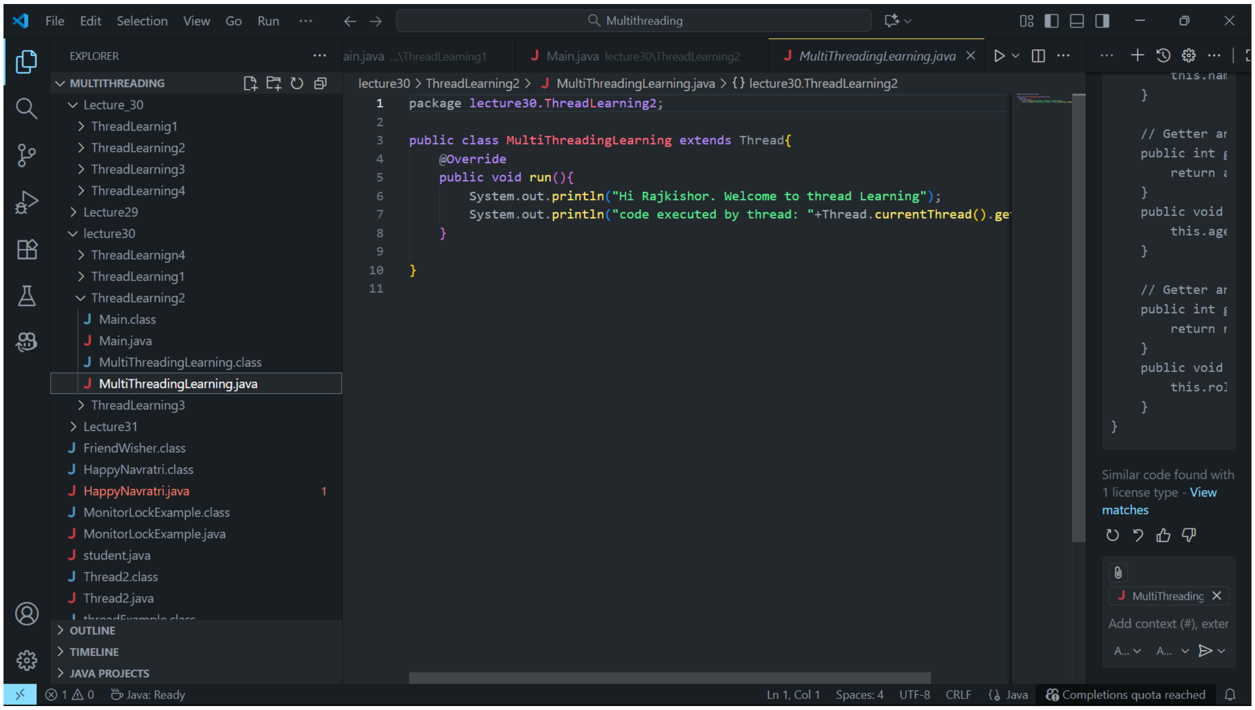
⚙️ Quick Summary

Mistake	Exception
Calling `.start()` twice on same thread	`IllegalThreadStateException`
Calling `.run()` manually (instead of `.start()`)	No new thread created — just runs in main thread
Reusing same `Runnable` object across threads	OK — as long as code is thread-safe

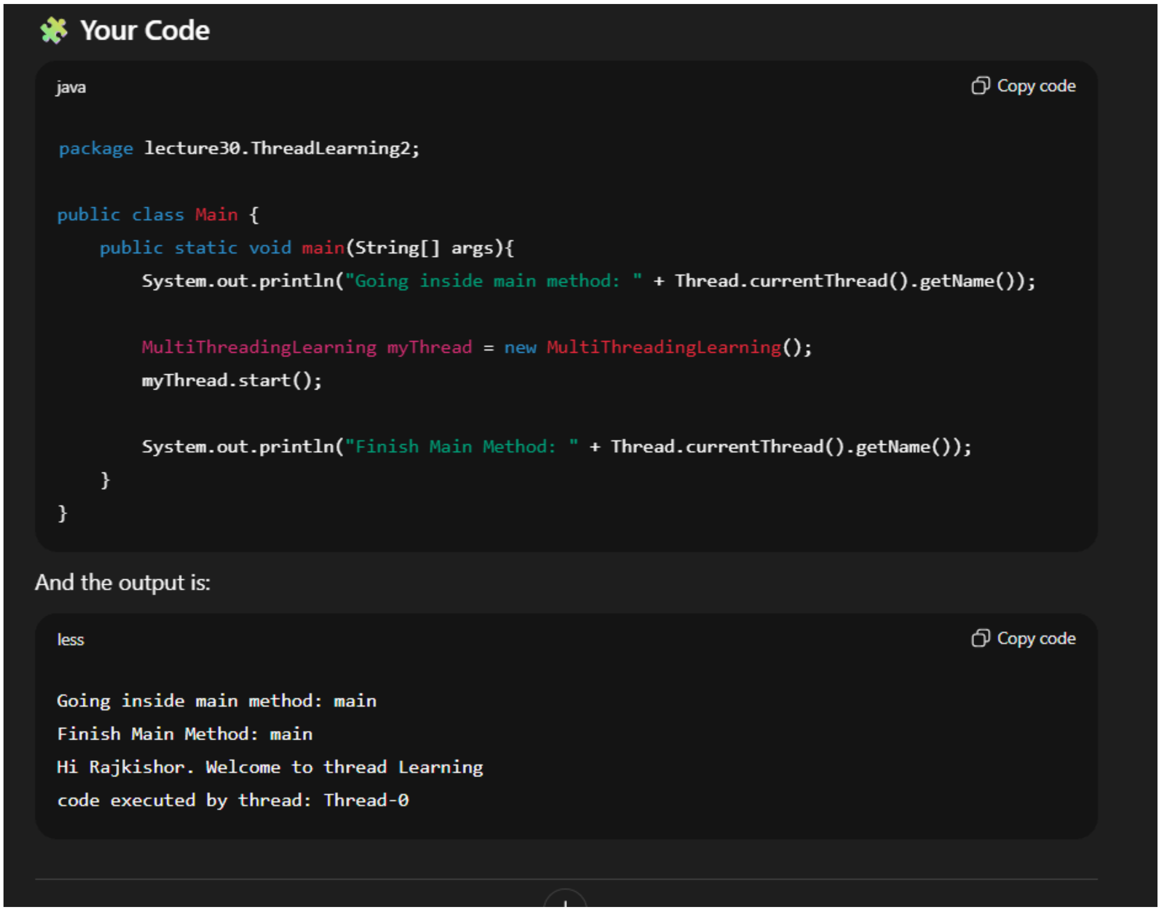
✅ Correct Output (after fixing)

If you fix it as shown above, output will look like:

```
Going inside main method: main
Thread is running: Thread-0
Thread is running: Thread-1
Finish Main method
```



```java
package lecture30.ThreadLearning2;

public class MultiThreadingLearning extends Thread{
    @Override
    public void run(){
        System.out.println("Hi Rajkishor. Welcome to thread Learning");
        System.out.println("code executed by thread: "+Thread.currentThread().ge
    }

}
```



Your Code

```java
package lecture30.ThreadLearning2;

public class Main {
    public static void main(String[] args){
        System.out.println("Going inside main method: " + Thread.currentThread().getName());

        MultiThreadingLearning myThread = new MultiThreadingLearning();
        myThread.start();

        System.out.println("Finish Main Method: " + Thread.currentThread().getName());
    }
}
```

And the output is:

```less
Going inside main method: main
Finish Main Method: main
Hi Rajkishor. Welcome to thread Learning
code executed by thread: Thread-0
```

🔹 The Reason — Parallel Execution

When you call:

```
myThread.start();
```

you are **starting a new thread** (not running it directly).

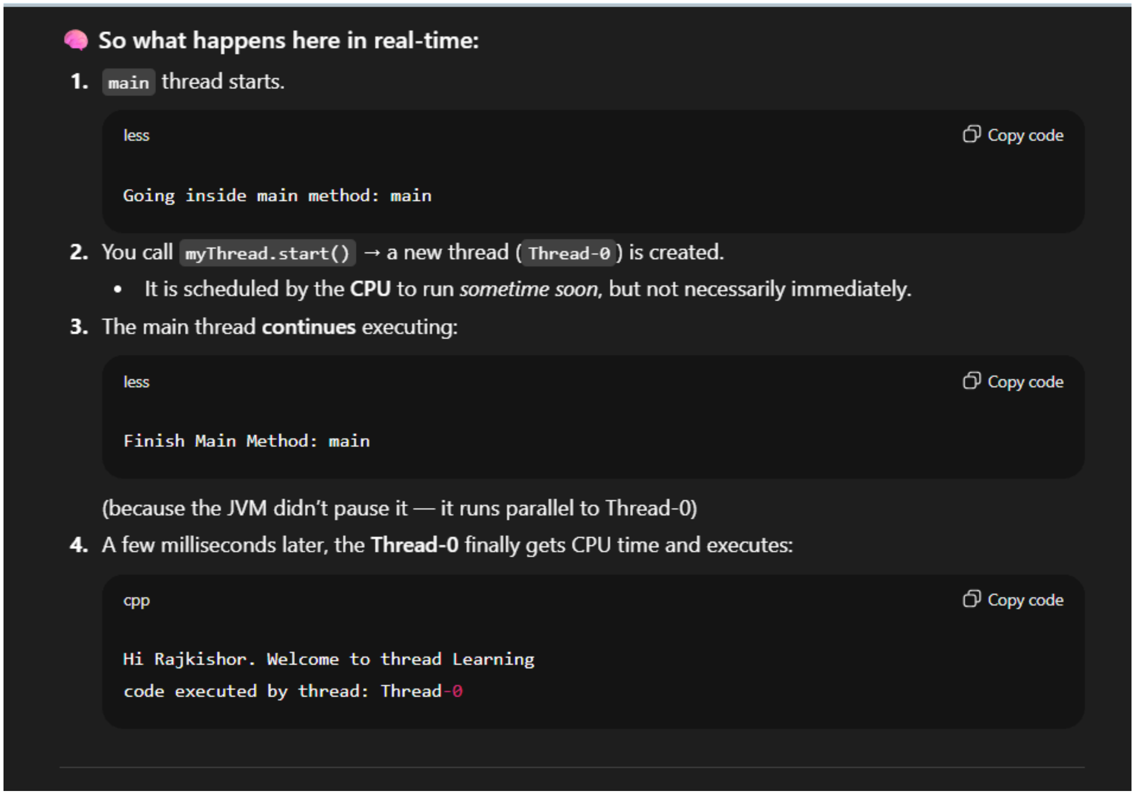
That means two threads are now running **concurrently**:

1. The **main thread**, which runs the `main()` method.
2. The **child thread (Thread-0)**, which runs the `run()` method of your `MultiThreadingLearning` class.

🔸 What `.start()` actually does

When you call `myThread.start()`:

- JVM creates a **new call stack** for that thread.
- The **new thread** begins executing **independently** from the main thread.
- The main thread does **not** wait for it to finish — it just continues executing the next statement.



🧠 **So what happens here in real-time:**

1. `main` thread starts.

   ```less
   Going inside main method: main
   ```

2. You call `myThread.start()` → a new thread (`Thread-0`) is created.
   - It is scheduled by the **CPU** to run *sometime soon*, but not necessarily immediately.
3. The main thread **continues** executing:

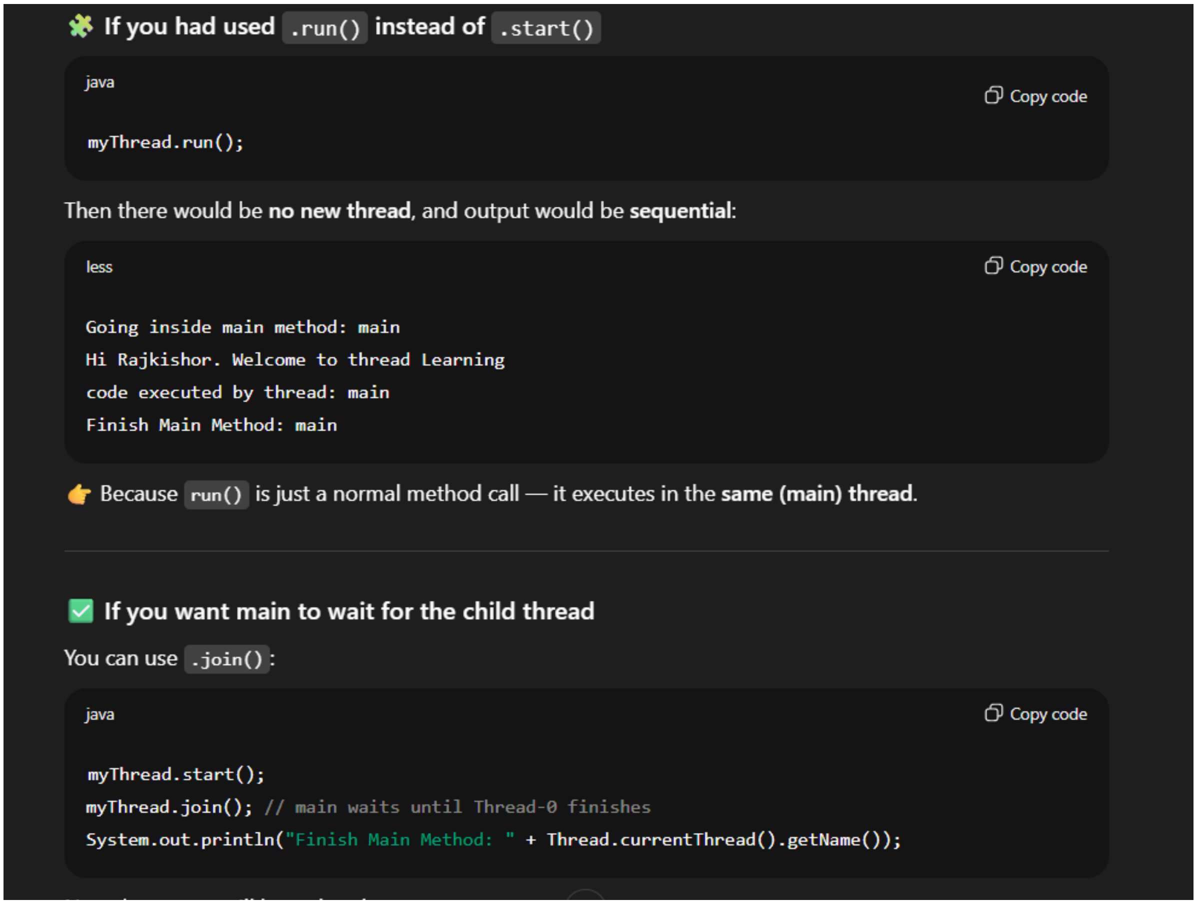
   ```less
   Finish Main Method: main
   ```

   (because the JVM didn't pause it — it runs parallel to Thread-0)
4. A few milliseconds later, the **Thread-0** finally gets CPU time and executes:

   ```cpp
   Hi Rajkishor. Welcome to thread Learning
   code executed by thread: Thread-0
   ```

🔹 Key Concept: Thread Scheduling

- Threads are managed by the **CPU scheduler**, not by Java directly.
- Java asks the OS to "run this thread," but **it can't control exactly when** it will start running.
- Depending on the CPU load, OS, and timing, sometimes the new thread starts **after** the main thread finishes, or even before it (if the scheduler picks it earlier).



🧩 If you had used `.run()` instead of `.start()`

```java
myThread.run();
```

Then there would be **no new thread**, and output would be **sequential**:

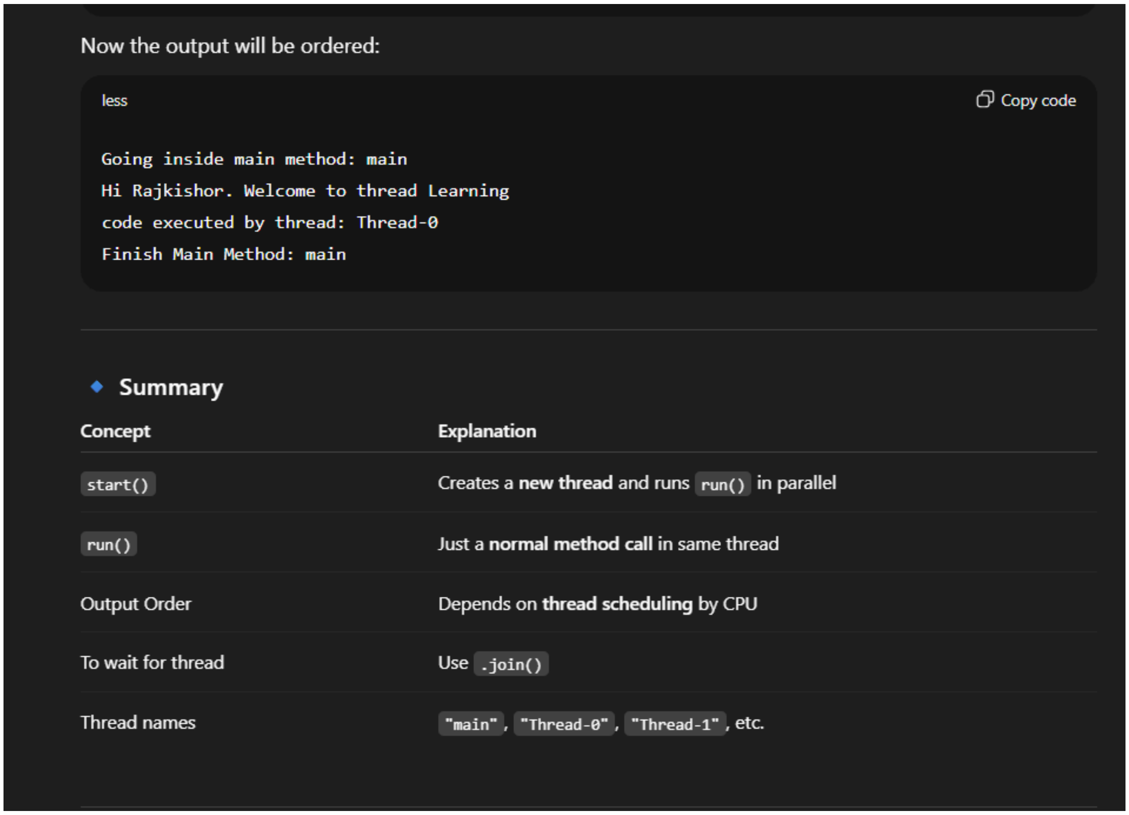
```less
Going inside main method: main
Hi Rajkishor. Welcome to thread Learning
code executed by thread: main
Finish Main Method: main
```

👉 Because `run()` is just a normal method call — it executes in the **same (main) thread**.

✅ If you want main to wait for the child thread

You can use `.join()`:

```java
myThread.start();
myThread.join(); // main waits until Thread-0 finishes
System.out.println("Finish Main Method: " + Thread.currentThread().getName());
```

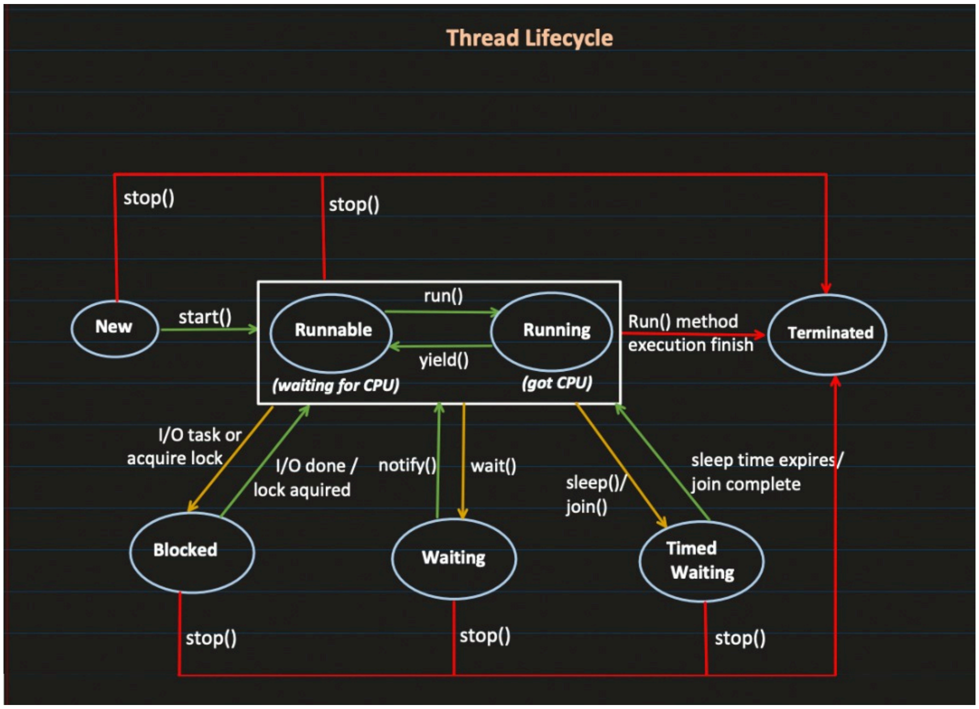


Now the output will be ordered:

```less
Going inside main method: main
Hi Rajkishor. Welcome to thread Learning
code executed by thread: Thread-0
Finish Main Method: main
```

🔹 Summary

Concept	Explanation
`start()`	Creates a **new thread** and runs `run()` in parallel
`run()`	Just a **normal method call** in same thread
Output Order	Depends on **thread scheduling** by CPU
To wait for thread	Use `.join()`
Thread names	`"main"`, `"Thread-0"`, `"Thread-1"`, etc.



🧠 Understanding Java Thread Lifecycle

A thread in Java goes through several **well-defined states** from creation to completion.
Let's follow the lifecycle in order:

🟢 1. New State

When you create a thread using the `new` keyword:

```
Thread t = new Thread();
```

or

```
Thread t = new Thread(new MyRunnable());
```

- the thread is **created but not yet started**.

🧩 **State:** NEW
➡️ **Transition to next state:** When you call `t.start()`

🟡 2. Runnable State (Ready to Run)

When you call:

```
t.start();
```

- the thread enters the **Runnable** state.

📖 Runnable means **eligible to run**, but not yet running.
It’s **waiting for the CPU** to assign time for execution.

🧩 **State:** RUNNABLE
➡️ **Transition to next state:** When the **CPU scheduler** picks this thread to execute.

🟢 3. Running State

- Once the CPU scheduler gives CPU time to the thread, it moves to the **Running** state.
- The thread’s `run()` method starts executing.

🧩 **State:** RUNNING
➡️ **Transition to next state:**

- If `run()` finishes → moves to **Terminated**
- If it calls `sleep()`, `wait()`, or does I/O → goes to **Blocked / Waiting / Timed Waiting**

🟠 4. Blocked State

- A thread enters **Blocked** state when it’s **waiting to acquire a lock** or **perform I/O operation**.

Example:

```
synchronized(lock) {
    // critical section
}
```

If another thread already holds the lock, this thread becomes **Blocked** until the lock is released.

🧩 **State:** BLOCKED
➡️ **Back to Runnable:** when I/O completes or lock is acquired.

🟡 5. Waiting State

- A thread enters **Waiting** when it's waiting **indefinitely** for another thread to perform an action.

Example:

```
object.wait();   // waits until notified
```

or waiting for another thread's `join()` to finish.

🧩 **State:** WAITING
➡️ **Back to Runnable:** when another thread calls `notify()` or `notifyAll()`.

🟠 6. Timed Waiting State

- A thread is in **Timed Waiting** when it waits for a **specific period of time**.

Example:

```
Thread.sleep(2000);   // waits for 2 seconds
t.join(5000);         // waits max 5 seconds
object.wait(1000);    // waits 1 second for notification
```

🧩 **State:** TIMED WAITING
➡️ **Back to Runnable:** when time expires or condition is met.

🔴 7. Terminated (Dead) State

- When the `run()` method finishes execution (normally or by exception), the thread enters **Terminated** state.

🧩 **State:** TERMINATED
➡️ No further transitions — the thread **cannot be restarted** (calling `start()` again throws `IllegalThreadStateException`).

🔁 Lifecycle Transitions (Summary from Diagram)

From	To	Trigger
New	Runnable	`start()`
Runnable	Running	CPU schedules thread
Running	Runnable	`yield()` (voluntarily gives up CPU)
Running	Blocked	Waiting for lock or I/O
Blocked	Runnable	Lock/I/O acquired or done
Running	Waiting	`wait()` or waiting for another thread
Waiting	Runnable	`notify()` or `notifyAll()`
Running	Timed Waiting	`sleep()` or `join(timeout)`
Timed Waiting	Runnable	Timeout expires
Running	Terminated	`run()` method finishes or exception occurs

🧩 Quick Analogy

Think of threads like **people waiting to use a computer**:

Thread State	Analogy

New	Created, not arrived yet
Runnable	Waiting in queue for the CPU
Running	Currently using the CPU
Blocked	Waiting for I/O or lock
Waiting	Waiting for another person’s signal
Timed Waiting	Waiting for a fixed time
Terminated	Done using the CPU and gone home

🧠 Bonus Tip: Methods That Cause State Changes

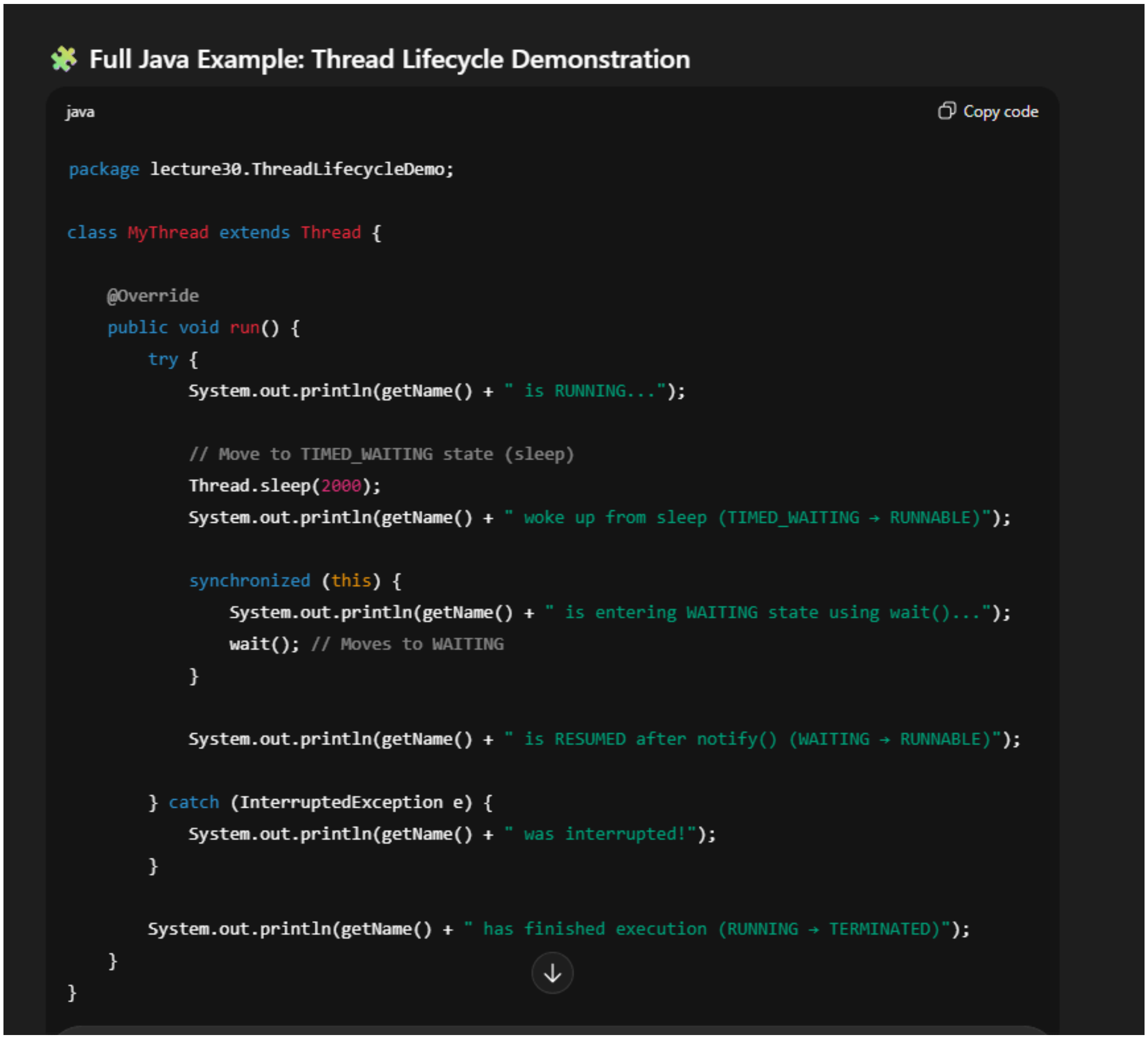
Method	Moves Thread To	Description
`start()`	Runnable	Starts a new thread
`run()`	Running	Actual execution logic
`sleep(ms)`	Timed Waiting	Pauses for a given time
`wait()`	Waiting	Waits for notification
`notify()`	Runnable	Wakes up waiting thread
`join()`	Waiting	Main waits for another thread
`yield()`	Runnable	Suggests scheduler to switch thread
`interrupt()`	Changes to Terminated or resumes	Interrupts a sleeping/waiting thread

✅ Final Summary

- **New → start() → Runnable → Running → (Waiting / Timed Waiting / Blocked) →** back to Runnable → **Terminated**

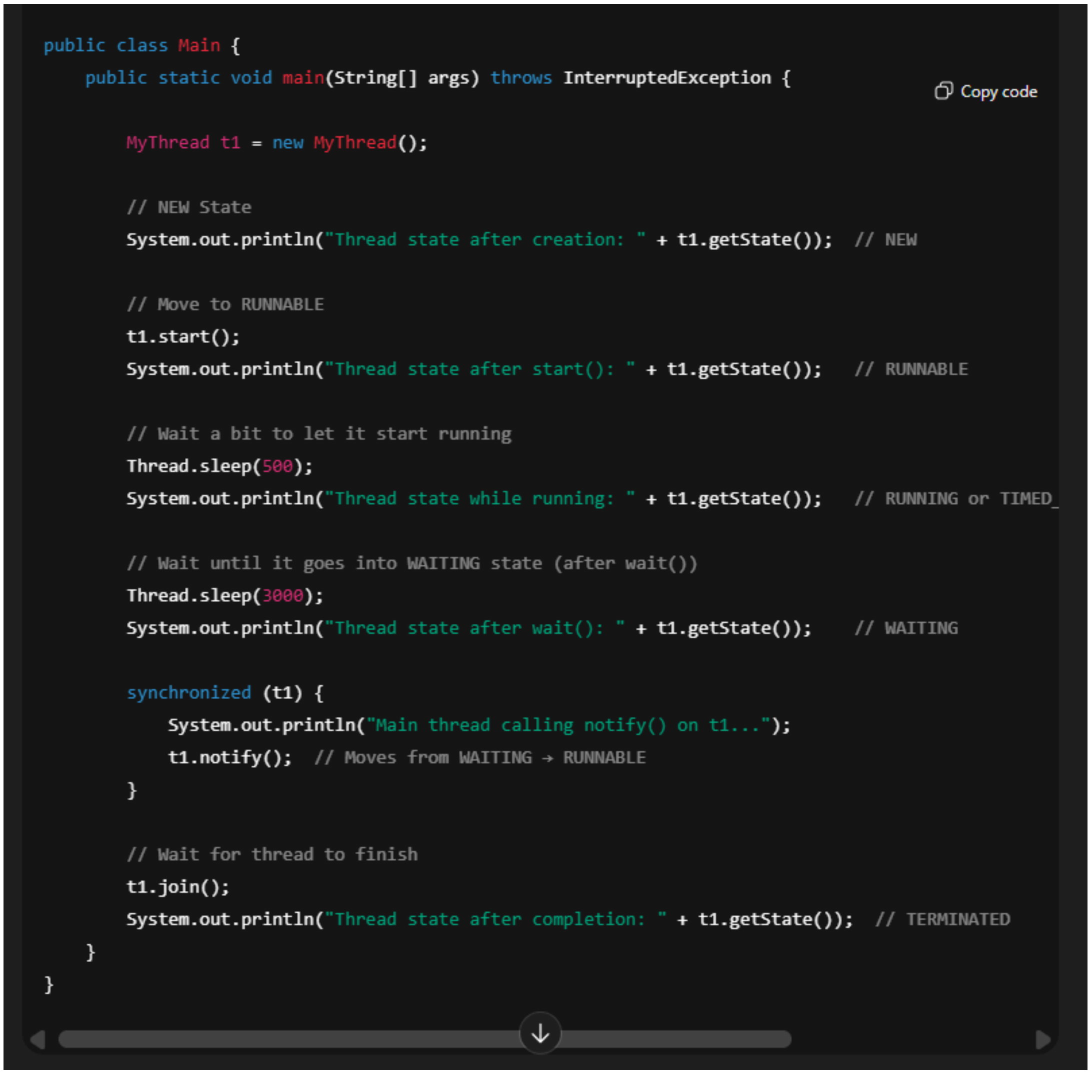
- The **JVM scheduler** decides which thread runs at any given time.
- Once **Terminated**, a thread **cannot be restarted**.

Perfect 👍 Let’s go step-by-step through a **practical Java code example** that clearly demonstrates the **thread lifecycle transitions** (`NEW → RUNNABLE → RUNNING → WAITING → TIMED_WAITING → TERMINATED`).



🧩 Full Java Example: Thread Lifecycle Demonstration

```java
package lecture30.ThreadLifecycleDemo;

class MyThread extends Thread {

    @Override
    public void run() {
        try {
            System.out.println(getName() + " is RUNNING...");

            // Move to TIMED_WAITING state (sleep)
            Thread.sleep(2000);
            System.out.println(getName() + " woke up from sleep (TIMED_WAITING → RUNNABLE)");

            synchronized (this) {
                System.out.println(getName() + " is entering WAITING state using wait()...");
                wait(); // Moves to WAITING
            }

            System.out.println(getName() + " is RESUMED after notify() (WAITING → RUNNABLE)");

        } catch (InterruptedException e) {
            System.out.println(getName() + " was interrupted!");
        }

        System.out.println(getName() + " has finished execution (RUNNING → TERMINATED)");
    }
}
```



```java
public class Main {
    public static void main(String[] args) throws InterruptedException {

        MyThread t1 = new MyThread();

        // NEW State
        System.out.println("Thread state after creation: " + t1.getState());  // NEW

        // Move to RUNNABLE
        t1.start();
        System.out.println("Thread state after start(): " + t1.getState());   // RUNNABLE

        // Wait a bit to let it start running
        Thread.sleep(500);
        System.out.println("Thread state while running: " + t1.getState());   // RUNNING or TIMED_

        // Wait until it goes into WAITING state (after wait())
        Thread.sleep(3000);
        System.out.println("Thread state after wait(): " + t1.getState());    // WAITING

        synchronized (t1) {
            System.out.println("Main thread calling notify() on t1...");
            t1.notify();  // Moves from WAITING → RUNNABLE
        }

        // Wait for thread to finish
        t1.join();
        System.out.println("Thread state after completion: " + t1.getState());  // TERMINATED
    }
}
```

🧩 Sample Output (approximate)

```
Thread state after creation: NEW
Thread state after start(): RUNNABLE
MyThread-0 is RUNNING...
Thread state while running: TIMED_WAITING
MyThread-0 woke up from sleep (TIMED_WAITING → RUNNABLE)
MyThread-0 is entering WAITING state using wait()...
Thread state after wait(): WAITING
Main thread calling notify() on t1...
MyThread-0 is RESUMED after notify() (WAITING → RUNNABLE)
MyThread-0 has finished execution (RUNNING → TERMINATED)
Thread state after completion: TERMINATED
```

🧠 Explanation of Transitions

Stage	Code	Description
NEW	`new MyThread()`	Thread object created but not started.
RUNNABLE	`t1.start()`	Thread eligible for running.
RUNNING	Scheduler gives CPU → executes `run()`.	
TIMED_WAITING	Inside `run()`, `Thread.sleep(2000)` pauses execution for 2s.	
WAITING	Inside `run()`, `wait()` suspends thread until `notify()` is called.	
RUNNABLE (again)	After `notify()`, thread becomes runnable again.	
TERMINATED	After `run()` finishes, thread dies.	

🔍 Concepts Shown

Lifecycle State	Trigger in Code
NEW	`new MyThread()`
RUNNABLE	`t1.start()`
RUNNING	Thread scheduler picks it
TIMED_WAITING	`Thread.sleep(2000)`
WAITING	`wait()`
RUNNABLE (again)	`notify()`
TERMINATED	`run()` method finishes

🧩 Context Recap

Inside the `run()` method of `MyThread`, we have this code:

```
synchronized (this) {
    System.out.println(getName() + " is entering WAITING state using
wait()...");
    wait(); // Thread-0 goes into WAITING state here
}
```

Later, in the `main()` thread, we wrote:

```
synchronized (t1) {
    System.out.println("Main thread calling notify() on t1...");
    t1.notify();  // wakes up Thread-0
}
```

🧠 What Happens Step-by-Step

🪜 Step 1: `wait()` called by Thread-0

- When the child thread (`t1`) executes `wait()`, it **temporarily releases the lock** on `this` (which is `t1` object itself) and goes into the **WAITING** state.

- It pauses indefinitely until **another thread** calls `notify()` or `notifyAll()` **on the same object**.

So at this point:

- `Thread-0` → **WAITING**

- `main` → still running

🪜 Step 2: `notify()` called by Main Thread

```
synchronized (t1) {
    t1.notify();
}
```

✅ This means:

> "Hey, I'm notifying (waking up) a thread that is currently waiting on the object `t1`."

- Only **threads waiting on this same object (`t1`)** can be notified.
- The waiting thread (`Thread-0`) moves from **WAITING** → **RUNNABLE**.
- However, it won't run immediately — it has to **reacquire the lock** on the object before continuing.

🪜 Step 3: Thread-0 resumes

Once the main thread exits the synchronized block (releasing the lock on `t1`), the waiting thread (`Thread-0`) reacquires the lock and resumes execution right after the `wait()` line.

It prints:

```
MyThread-0 is RESUMED after notify() (WAITING → RUNNABLE)
```

Then it completes and moves to **TERMINATED** state.

🔹 Important Rules About `wait()` and `notify()`

Rule	Explanation
Must be inside a `synchronized` block	Both `wait()` and `notify()` require the thread to **own the object's monitor (lock)**
`wait()` releases the lock	The waiting thread gives up the lock and sleeps

`notify()` doesn't release lock immediately	It just signals one waiting thread; lock is released only when the notifying thread exits its synchronized block
`notifyAll()`	Wakes up **all** threads waiting on the same object
Object monitor	Each Java object has a built-in monitor lock used by synchronized, wait, notify

---

🔍 Visualization of What's Happening

Time	Thread	Action	State
1	main	starts `t1`	RUNNABLE
2	Thread-0	runs and calls `wait()`	WAITING
3	main	enters synchronized(t1), calls `t1.notify()`	Thread-0 → RUNNABLE
4	main	exits synchronized block	releases lock
5	Thread-0	reacquires lock and resumes	RUNNING
6	Thread-0	finishes run()	TERMINATED

🧩 Analogy (Easy to Remember)

Imagine:

- `Thread-0` is **sleeping** in a locked room (`wait()`).
- `main` thread comes to the door and **knocks** (`notify()`).
- But Thread-0 can't leave **until** `main` unlocks and leaves the room (i.e., exits synchronized block).
- Once the door is unlocked, Thread-0 wakes up and continues.

✅ In One Line

`notify()` is a **signal** sent by one thread to another that says:

> “Hey, you can wake up now — the thing you were waiting for has happened.”