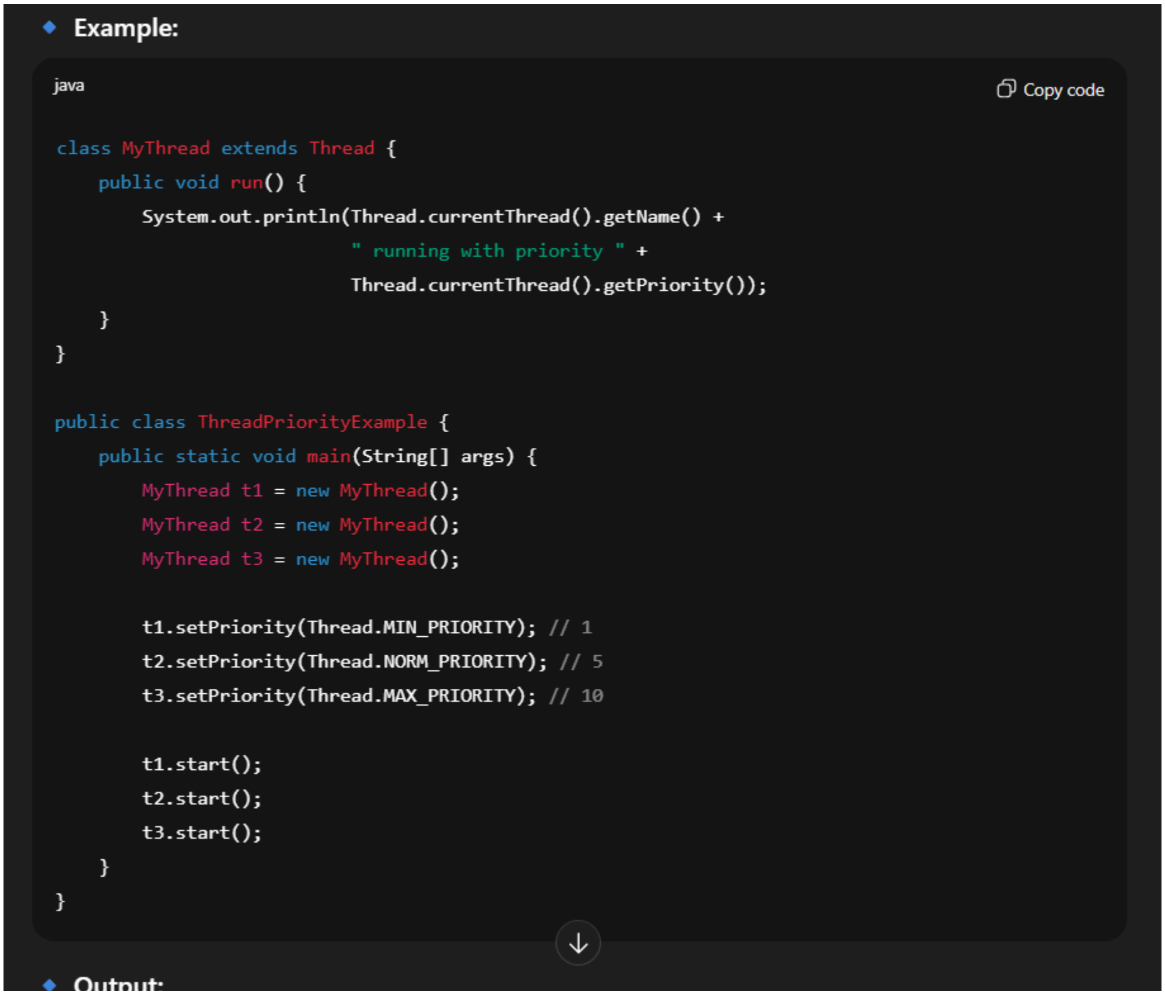
🧵 1. Thread Priority

🔹 Concept:

Each thread in Java has a **priority** (integer value between 1 and 10).

- Default: **5 (NORM_PRIORITY)**
- Lowest: **1 (MIN_PRIORITY)**
- Highest: **10 (MAX_PRIORITY)**

The **Thread Scheduler** uses priorities as a **hint**, not a strict rule — it's *platform dependent*.



🔹 **Example:**

```java
class MyThread extends Thread {
    public void run() {
        System.out.println(Thread.currentThread().getName() +
                           " running with priority " +
                           Thread.currentThread().getPriority());
    }
}

public class ThreadPriorityExample {
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        MyThread t2 = new MyThread();
        MyThread t3 = new MyThread();

        t1.setPriority(Thread.MIN_PRIORITY); // 1
        t2.setPriority(Thread.NORM_PRIORITY); // 5
        t3.setPriority(Thread.MAX_PRIORITY); // 10

        t1.start();
        t2.start();
        t3.start();
    }
}
```

🔹 **Output:**

🔹 Output:

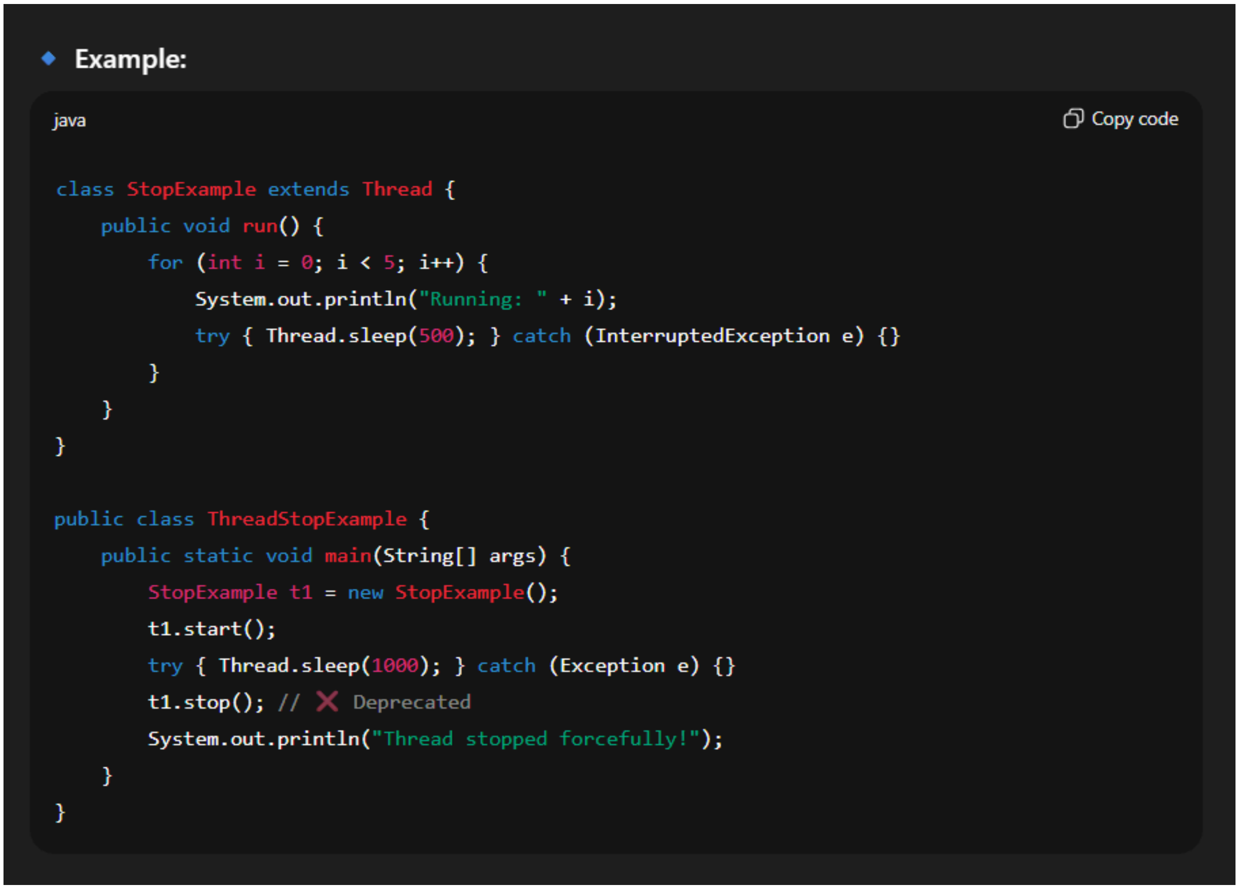
```
Thread-0 running with priority 1
Thread-1 running with priority 5
Thread-2 running with priority 10
```

👉 Note: The **order of execution is not guaranteed**. The scheduler *may* prefer higher-priority threads, but not always.

🛑 2. stop() — ❌ Deprecated

🔹 Concept:

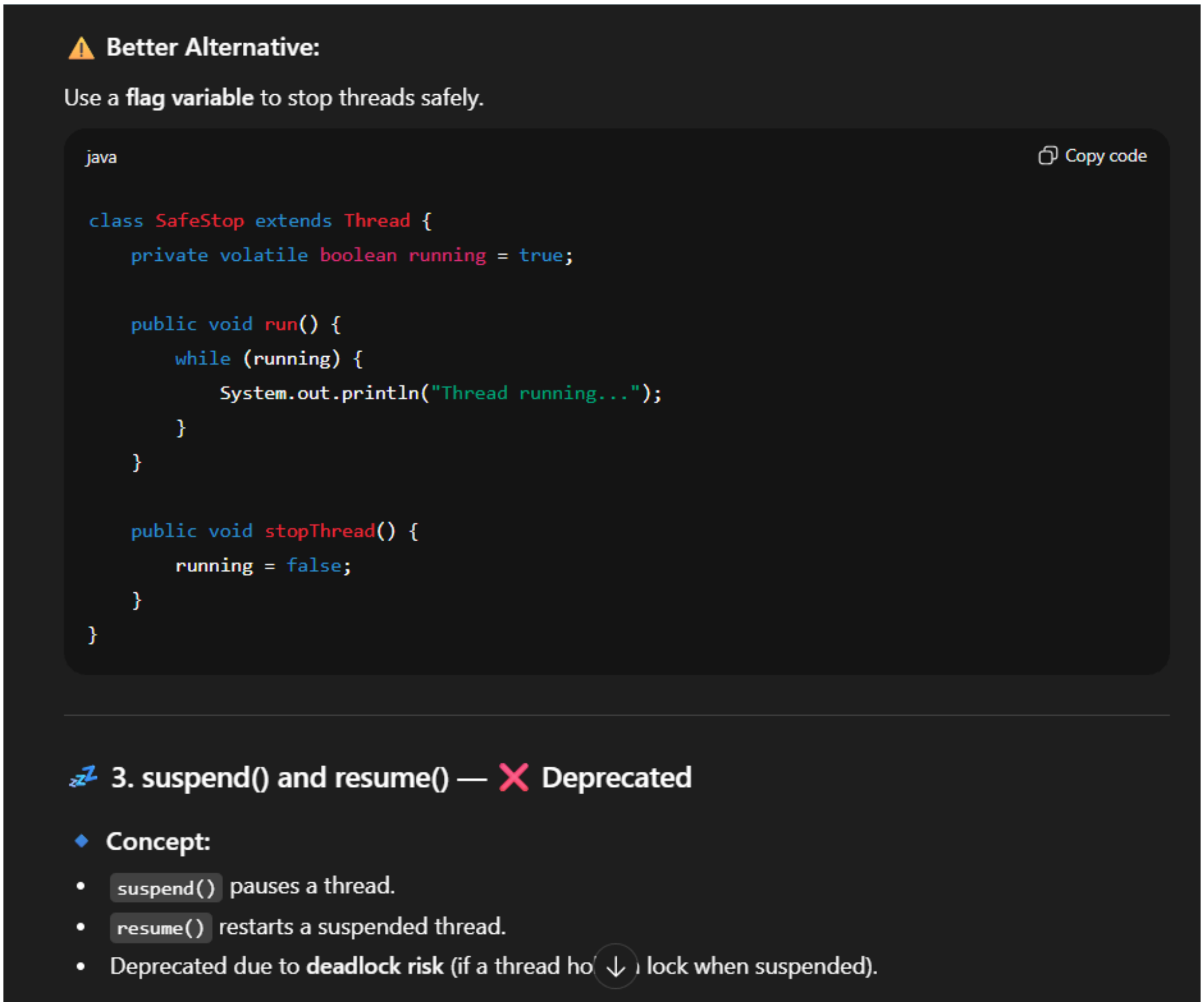
- Used to **forcefully terminate** a thread.
- Deprecated because it can leave shared resources in an **inconsistent state** (it kills the thread without releasing locks).



🔹 Example:

```java
class StopExample extends Thread {
    public void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println("Running: " + i);
            try { Thread.sleep(500); } catch (InterruptedException e) {}
        }
    }
}

public class ThreadStopExample {
    public static void main(String[] args) {
        StopExample t1 = new StopExample();
        t1.start();
        try { Thread.sleep(1000); } catch (Exception e) {}
        t1.stop(); // ❌ Deprecated
        System.out.println("Thread stopped forcefully!");
    }
}
```



⚠️ Better Alternative:

Use a **flag variable** to stop threads safely.

```java
class SafeStop extends Thread {
    private volatile boolean running = true;

    public void run() {
        while (running) {
            System.out.println("Thread running...");
        }
    }

    public void stopThread() {
        running = false;
    }
}
```

💤 3. suspend() and resume() — ❌ Deprecated

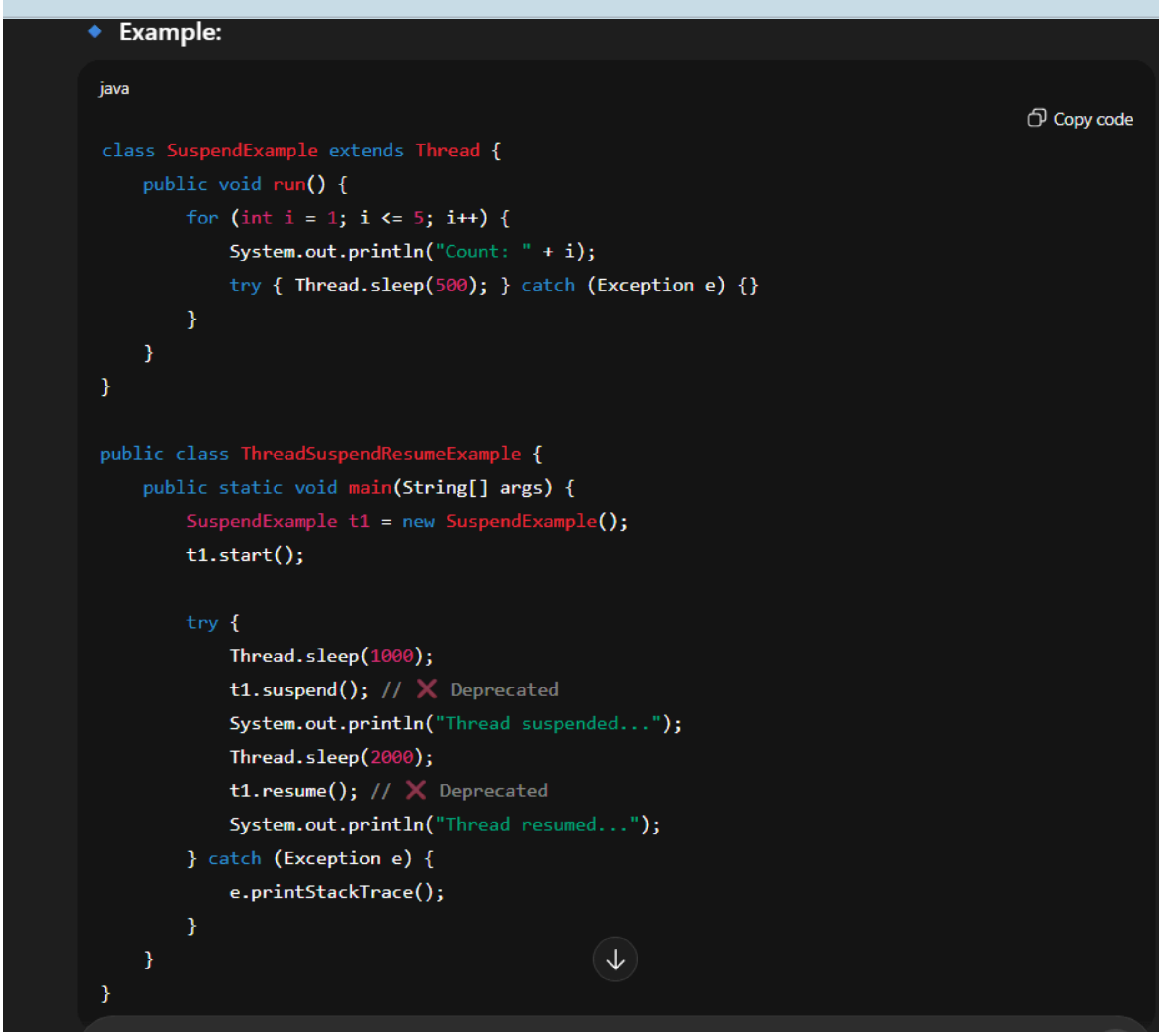
🔹 **Concept:**

- `suspend()` pauses a thread.
- `resume()` restarts a suspended thread.
- Deprecated due to **deadlock risk** (if a thread ho↓ lock when suspended).

💤 3. suspend() and resume() — ❌ Deprecated

🔹 Concept:

- `suspend()` pauses a thread.
- `resume()` restarts a suspended thread.
- Deprecated due to **deadlock risk** (if a thread holds a lock when suspended).



🔹 **Example:**

```java
class SuspendExample extends Thread {
    public void run() {
        for (int i = 1; i <= 5; i++) {
            System.out.println("Count: " + i);
            try { Thread.sleep(500); } catch (Exception e) {}
        }
    }
}

public class ThreadSuspendResumeExample {
    public static void main(String[] args) {
        SuspendExample t1 = new SuspendExample();
        t1.start();

        try {
            Thread.sleep(1000);
            t1.suspend(); // ❌ Deprecated
            System.out.println("Thread suspended...");
            Thread.sleep(2000);
            t1.resume(); // ❌ Deprecated
            System.out.println("Thread resumed...");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

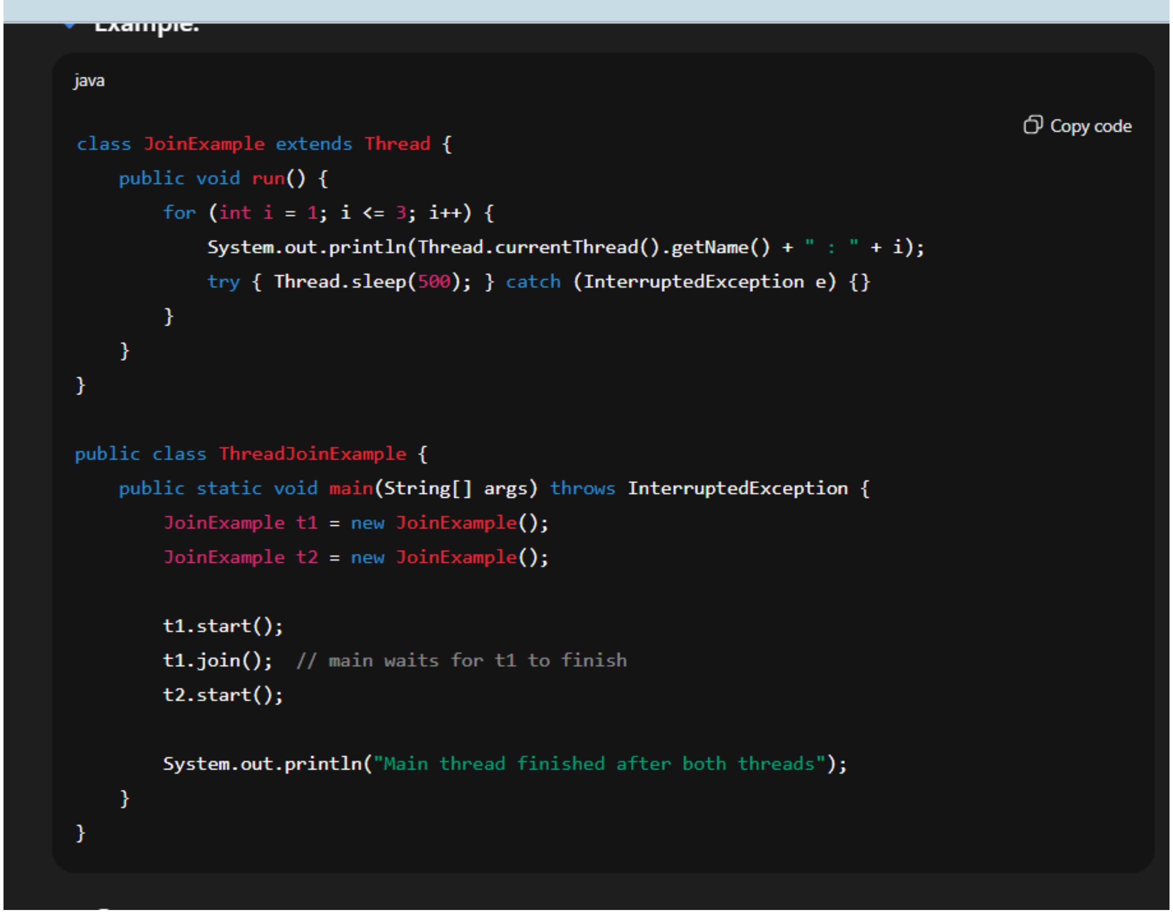
⚠️ Better Alternative:

Use a **custom pause mechanism** with `wait()` / `notify()`.

🤝 4. join()

🔹 Concept:

- Used to make one thread **wait** for another to **finish execution** before proceeding.
- Commonly used when you want **sequential execution** of threads.



```java
class JoinExample extends Thread {
    public void run() {
        for (int i = 1; i <= 3; i++) {
            System.out.println(Thread.currentThread().getName() + " : " + i);
            try { Thread.sleep(500); } catch (InterruptedException e) {}
        }
    }
}

public class ThreadJoinExample {
    public static void main(String[] args) throws InterruptedException {
        JoinExample t1 = new JoinExample();
        JoinExample t2 = new JoinExample();

        t1.start();
        t1.join();  // main waits for t1 to finish
        t2.start();

        System.out.println("Main thread finished after both threads");
    }
}
```

🔹 Output:

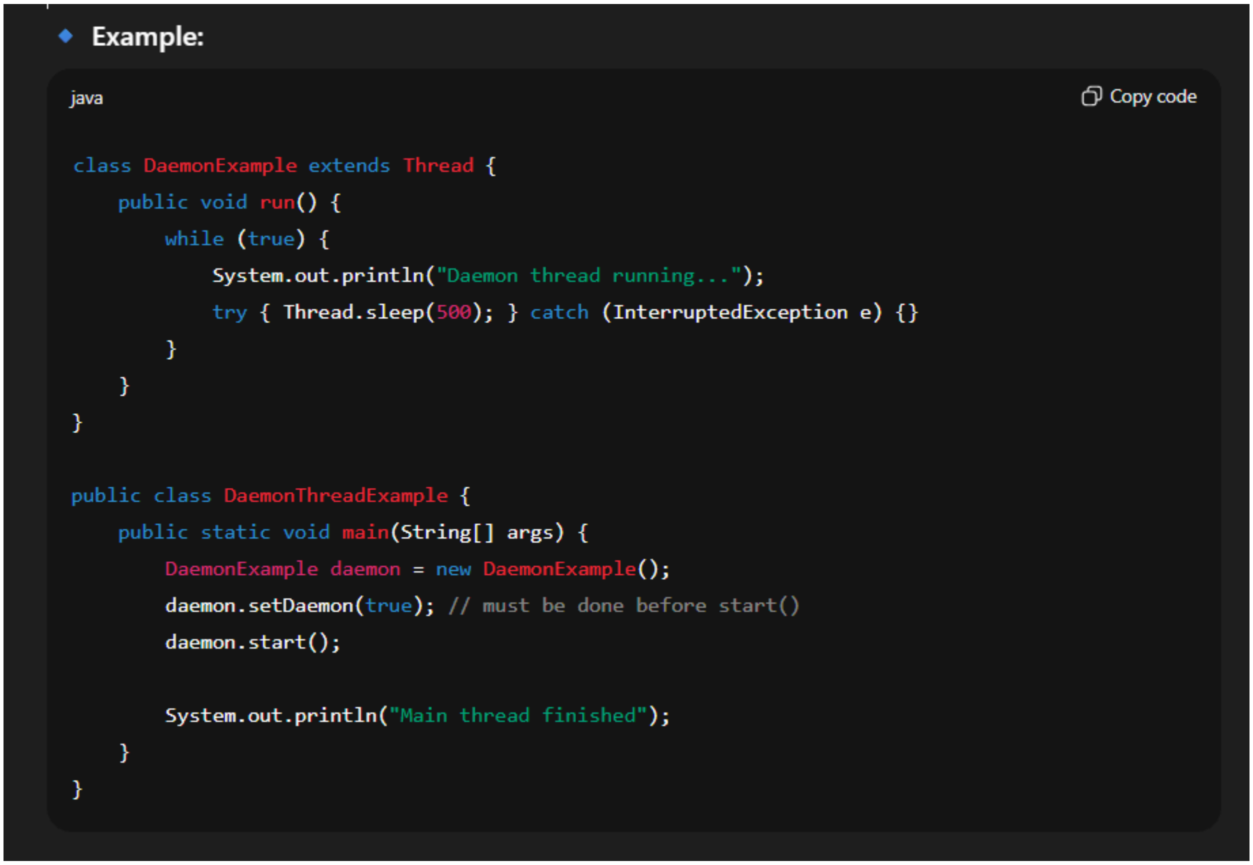
```
Thread-0 : 1
Thread-0 : 2
Thread-0 : 3
Thread-1 : 1
Thread-1 : 2
Thread-1 : 3
Main thread finished after both threads
```

👻 5. Daemon Threads

🔹 Concept:

- Daemon threads are **background service threads**.
- They **terminate automatically** when all **user (non-daemon)** threads finish.

- Used for background tasks like **garbage collection**, **logging**, etc.



🔹 **Example:**

java

Copy code

```java
class DaemonExample extends Thread {
    public void run() {
        while (true) {
            System.out.println("Daemon thread running...");
            try { Thread.sleep(500); } catch (InterruptedException e) {}
        }
    }
}

public class DaemonThreadExample {
    public static void main(String[] args) {
        DaemonExample daemon = new DaemonExample();
        daemon.setDaemon(true); // must be done before start()
        daemon.start();

        System.out.println("Main thread finished");
    }
}
```

🔹 Output:

```
Daemon thread running...
Main thread finished
```

👉 After main thread ends, daemon thread also stops **automatically**.

⚙️ Summary Table

Method	Description	Deprecated	Safe Alternative
`setPriority()`	Sets thread's priority (1–10)	❌ No	Use for scheduling hints only

`stop()`	Forcefully terminates thread	✅ Yes	Use flag variable
`suspend()`	Pauses thread	✅ Yes	Use `wait()`/`notify()`
`resume()`	Resumes suspended thread	✅ Yes	Use `notify()`
`join()`	Waits for another thread to finish	❌ No	Safe and recommended
`setDaemon(true)`	Marks thread as background	❌ No	Safe and recommended

Let’s understand the difference between **`await()`** / **`signal()`** (from `java.util.concurrent.locks.Condition`) and **`wait()`** / **`notify()`** (from `java.lang.Object`) — step by step.

🔹 1. Background — Both Serve the Same Purpose

Both sets of methods are used for **thread communication** —
that is, for one thread to **wait** until another thread **notifies** it that a certain condition has changed.

But they belong to **different concurrency APIs**.

🔸 `wait()` and `notify()` → Old (Intrinsic Lock Mechanism)

These come from the **Object class**, used with **synchronized blocks**.

🔹 Syntax Example

```
synchronized(lock) {
    while(!condition) {
        lock.wait();   // Thread waits, releases the lock
    }
    // Proceed after being notified
}
```

and somewhere else:

```
synchronized(lock) {
    condition = true;
    lock.notify();     // Wakes up one waiting thread
}
```

🔹 Key Points

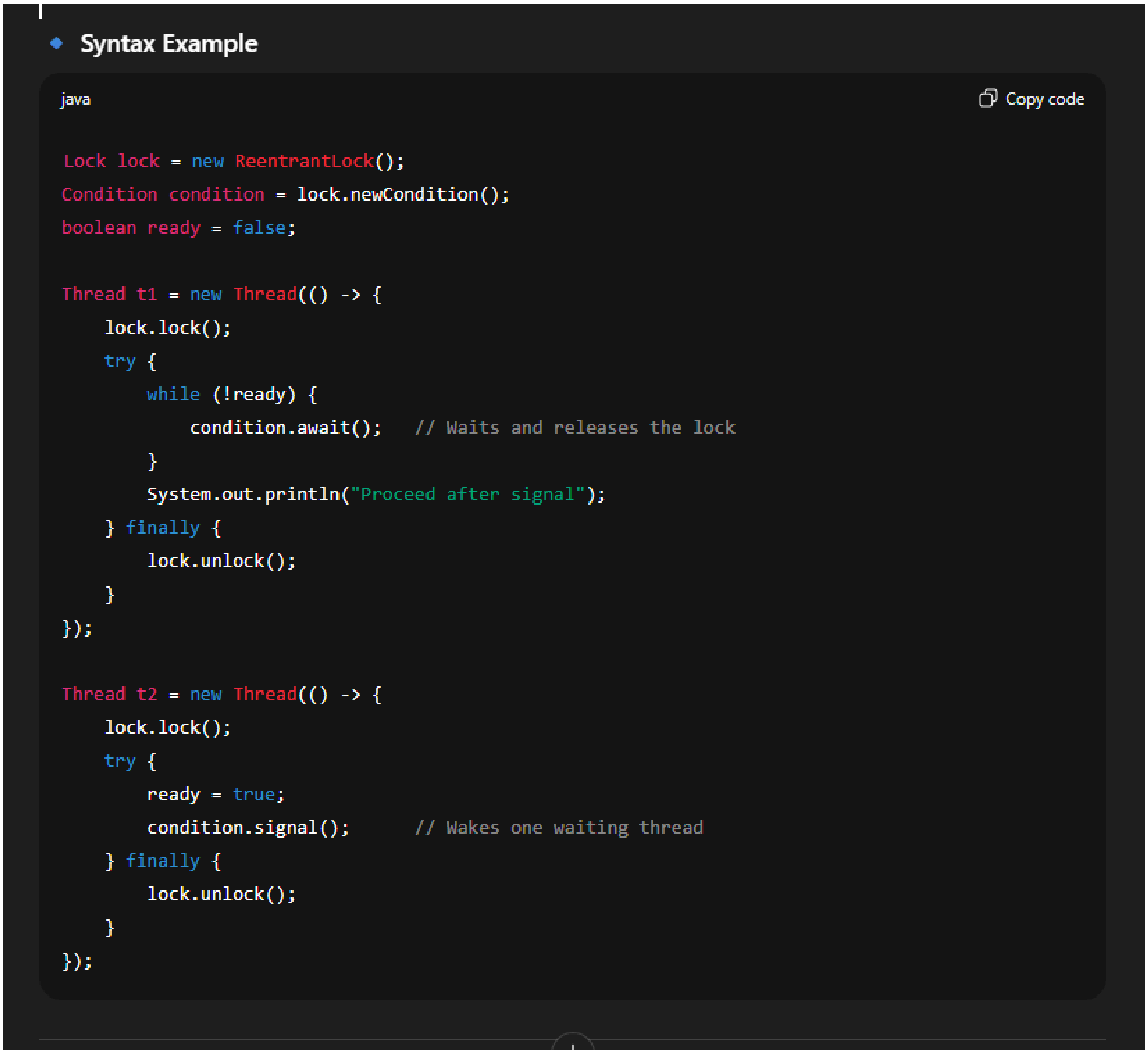
Aspect	**`wait()` / `notify()` / `notifyAll()`**
Defined in	`java.lang.Object`
Works with	`synchronized` blocks/methods
Lock Type	**Intrinsic (monitor)** lock
Methods	`wait()`, `notify()`, `notifyAll()`
Must hold lock?	✅ Yes, you must call inside `synchronized`
Releases lock while waiting?	✅ Yes
Reacquires lock when notified?	✅ Yes
Can have spurious wakeups?	✅ Yes — so always check condition in a loop

---

🔸 `await()` and `signal()` → New (Explicit Lock Mechanism)

These are from **`java.util.concurrent.locks.Condition`**, used with **`ReentrantLock`** or other Lock implementations.

They're a **more flexible, modern alternative** to `wait/notify`.



Syntax Example

```java
Lock lock = new ReentrantLock();
Condition condition = lock.newCondition();
boolean ready = false;

Thread t1 = new Thread(() -> {
    lock.lock();
    try {
        while (!ready) {
            condition.await();   // Waits and releases the lock
        }
        System.out.println("Proceed after signal");
    } finally {
        lock.unlock();
    }
});

Thread t2 = new Thread(() -> {
    lock.lock();
    try {
        ready = true;
        condition.signal();      // Wakes one waiting thread
    } finally {
        lock.unlock();
    }
});
```

Key Points

Aspect	**`await()` / `signal()` / `signalAll()`**
Defined in	`java.util.concurrent.locks.Condition`
Works with	`Lock` / `ReentrantLock`

Lock Type	**Explicit** lock
Methods	`await()`, `signal()`, `signalAll()`
Must hold lock?	✅ Yes, before calling
Releases lock while waiting?	✅ Yes
Reacquires lock after signal?	✅ Yes
Supports multiple condition objects	✅ Yes (unlike Object’s single monitor)

🔹 2. Comparison Table

Feature	**`wait()` / `notify()`**	**`await()` / `signal()`**
Package	`java.lang.Object`	`java.util.concurrent.locks.Condition`
Works with	`synchronized`	`Lock` (`ReentrantLock`)
Lock type	Intrinsic (monitor)	Explicit
Multiple conditions per lock	❌ No	✅ Yes
Must hold lock before call	✅ Yes	✅ Yes
Lock released during wait	✅ Yes	✅ Yes
Can respond to interrupts	✅ Yes (`InterruptedException`)	✅ Yes
Typical usage	Older threading code	Modern concurrent code
Performance	Less flexible	More fine-grained control
Replacement for	—	`wait()` / `notify()`

🔹 3. Example Difference in Practice

Using `wait()` / `notify()`

```
synchronized(lock) {
    while(!ready)
        lock.wait();
    System.out.println("Running...");
}

synchronized(lock) {
    ready = true;
    lock.notify();
}
```

Using `await()` / `signal()`

```
lock.lock();
try {
    while(!ready)
        condition.await();
    System.out.println("Running...");
} finally {
    lock.unlock();
}

lock.lock();
try {
    ready = true;
    condition.signal();
} finally {
    lock.unlock();
}
```

🔹 4. When to Use Which

Use Case	Recommendation
Simple synchronization (old code)	`wait()` / `notify()`
Complex multithreading (modern apps)	`await()` / `signal()` with `ReentrantLock`
Need multiple conditions on same lock	Use `Condition` (await/signal)
Working with Executors or concurrency utilities	Prefer `Condition` or higher-level tools (like `BlockingQueue`, `Semaphore`, `CountDownLatch`)

🧠 Summary

- `wait()` / `notify()` → legacy, tied to intrinsic locks (`synchronized`).
- `await()` / `signal()` → modern, tied to explicit locks (`ReentrantLock`).
- Both are used for **inter-thread coordination**, but `Condition` objects give **more flexibility and better design** in concurrent systems.