# Java Concurrency

Detailed Guide to Multithreading & Modern APIs

## What is Thread?

A **thread** is the **smallest unit of execution** inside a program.

It allows a program to do **multiple tasks at the same time**.

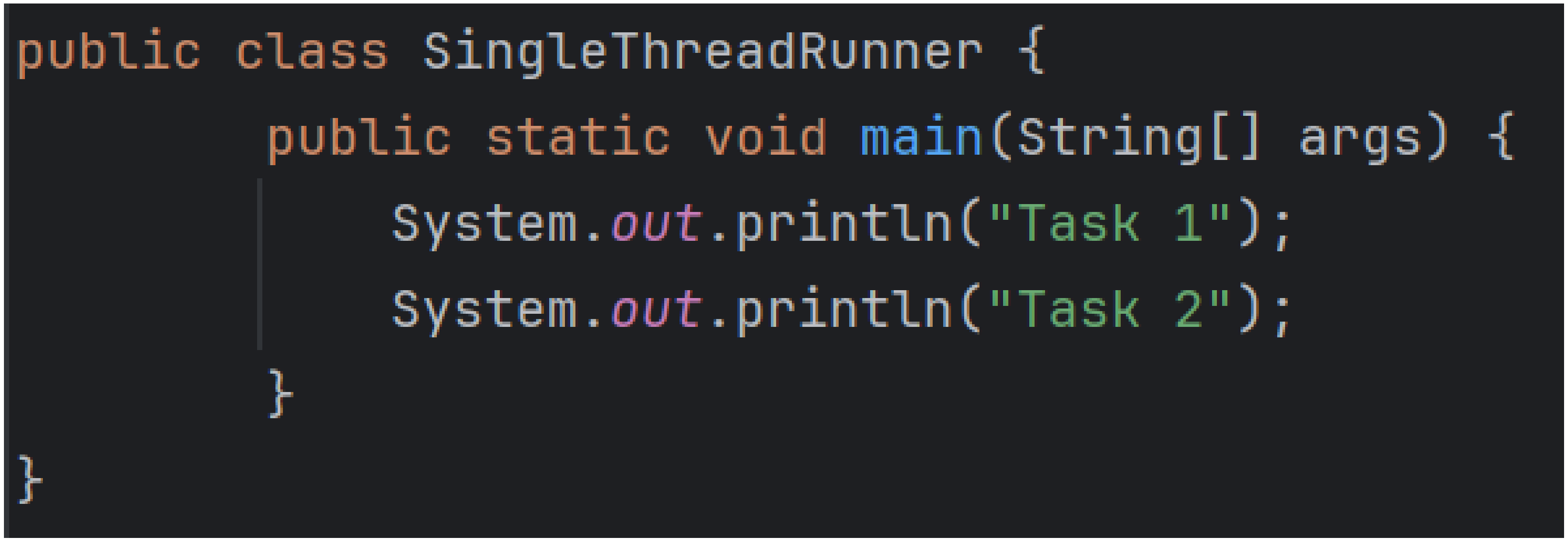
A **thread** is a lightweight process that runs **independently** inside a process.

## Process vs Thread

| Process | Thread |
|---|---|
| Running program | Path of execution |
| Has its own memory | Shares memory with other threads |
| Heavy | Lightweight |

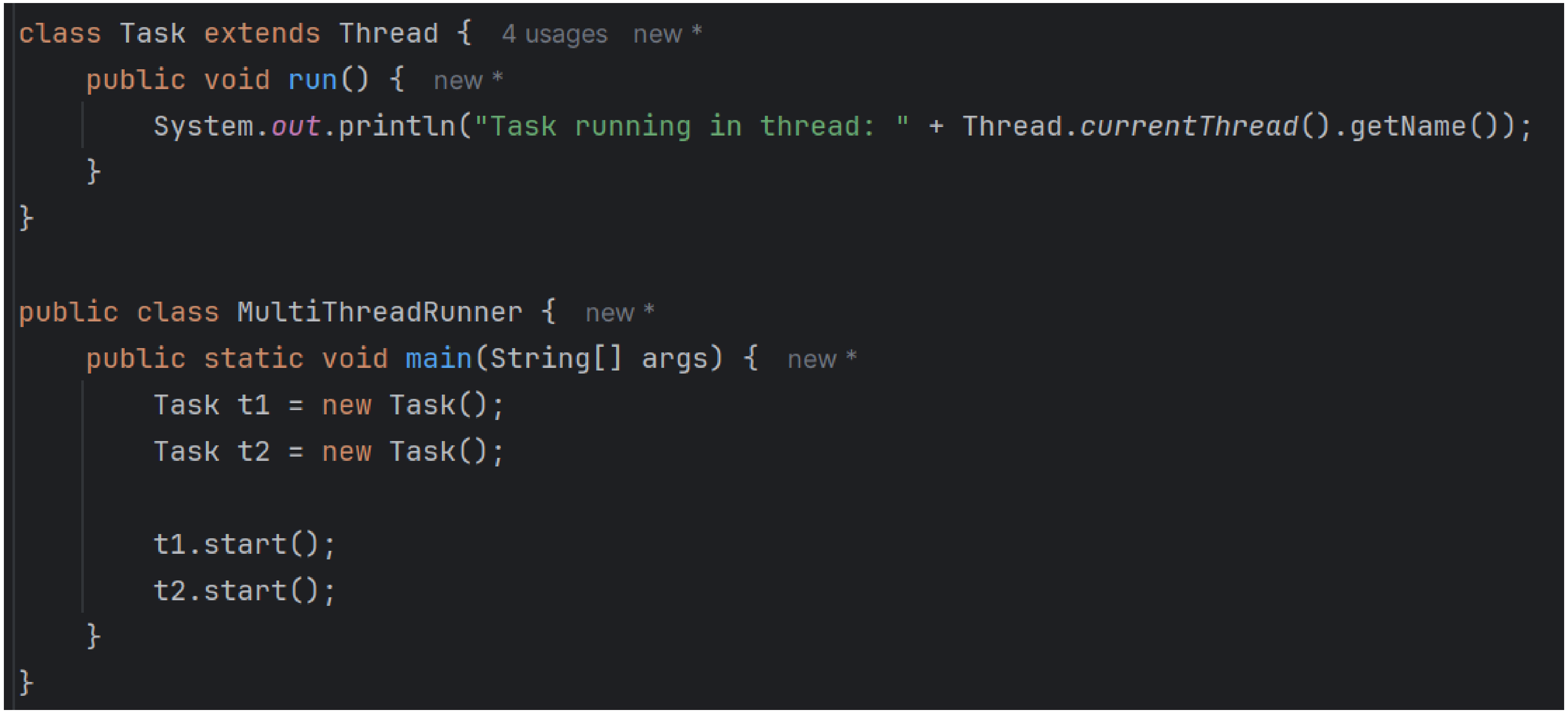
# Example (single-thread)

Tasks run **one after another**.



```java
public class SingleThreadRunner {
        public static void main(String[] args) {
            System.out.println("Task 1");
            System.out.println("Task 2");
        }
}
```

# Example (multi-thread)



```
class Task extends Thread {  4 usages  new *
    public void run() {  new *
        System.out.println("Task running in thread: " + Thread.currentThread().getName());
    }
}

public class MultiThreadRunner {  new *
    public static void main(String[] args) {  new *
        Task t1 = new Task();
        Task t2 = new Task();

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

## Why threads are important?

Improves **performance**

Keeps apps **responsive**

Used in **web servers, Spring Boot, microservices**

## What is a Thread?

Smallest unit of execution.

Analogy:

**Process** = Restaurant
**Threads** = Chefs working simultaneously in same kitchen

Threads share Heap, but have unique Stacks.

The **Thread Lifecycle** describes the **states a thread goes through** from creation to termination.

## Thread Lifecycle

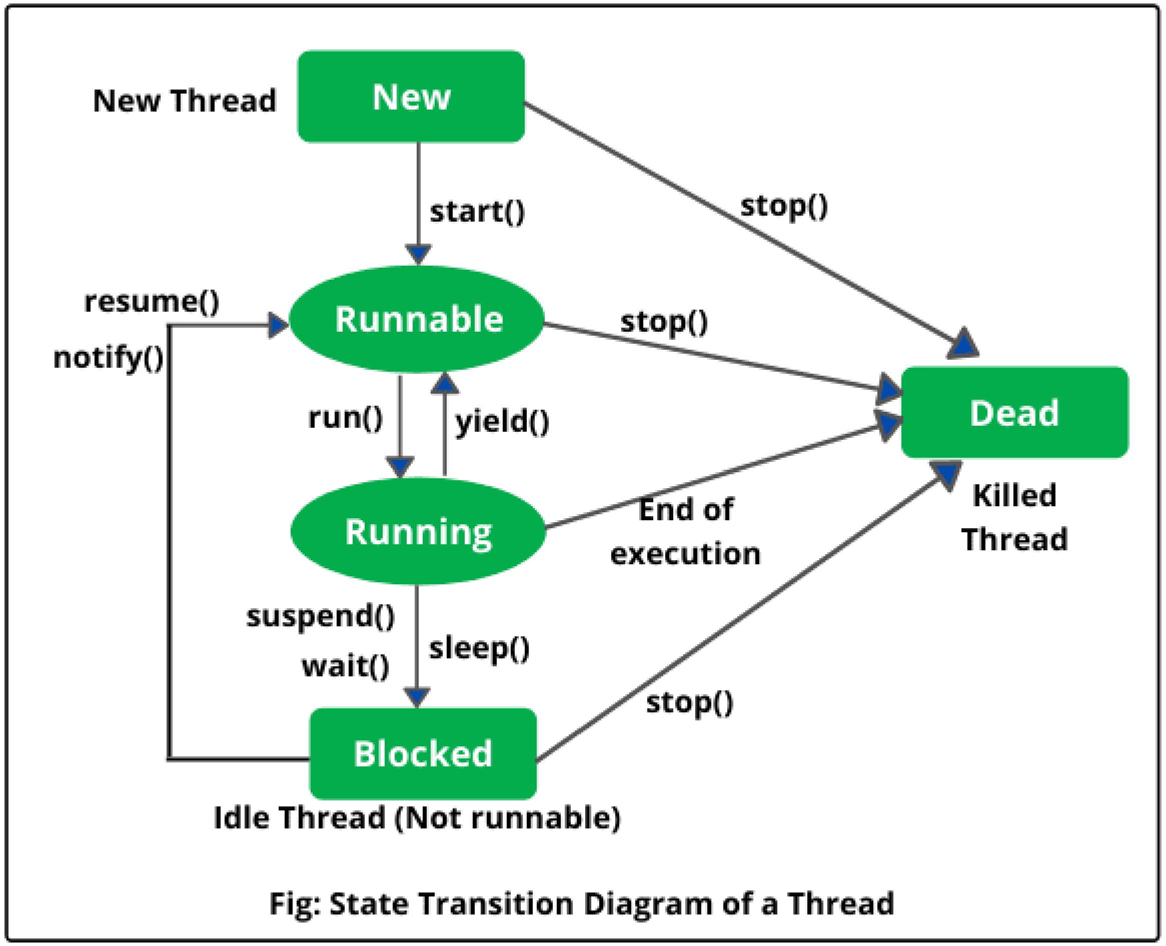


Fig: State Transition Diagram of a Thread

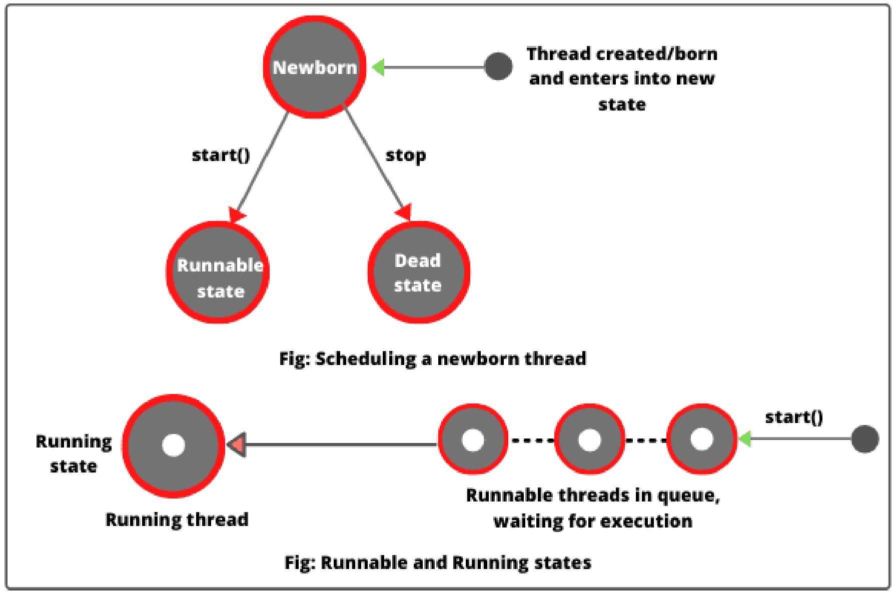
## Thread Lifecycle

- The Java thread life cycle consists of six distinct states—New, Runnable, Blocked, Waiting, Timed Waiting, and Terminated—defined by Thread.State.
- A thread starts in the **New** state, becomes **Runnable** via start(), executes in the **Running** (part of Runnable) state, enters **Waiting/Timed Waiting/Blocked** when paused, and finally moves to **Terminated** upon completion.

## Thread Lifecycle States

- **New (Born):** A thread is in this state when an instance of the Thread class is created, but the start() method has not yet been invoked.
- Thread t = new Thread(); // NEW

## NEW


Newborn
Thread created/born and enters into new state
start()
stop
Runnable state
Dead state
Fig: Scheduling a newborn thread
Running state
start()
Running thread
Runnable threads in queue, waiting for execution
Fig: Runnable and Running states

```
// Create a new thread. This thread is in NEW state.
    MyThread thread = new MyThread();
    Thread th = new Thread(thread);

 // Check thread state (NEW)
    System.out.println("Thread state before start: " + thread.getState());
```

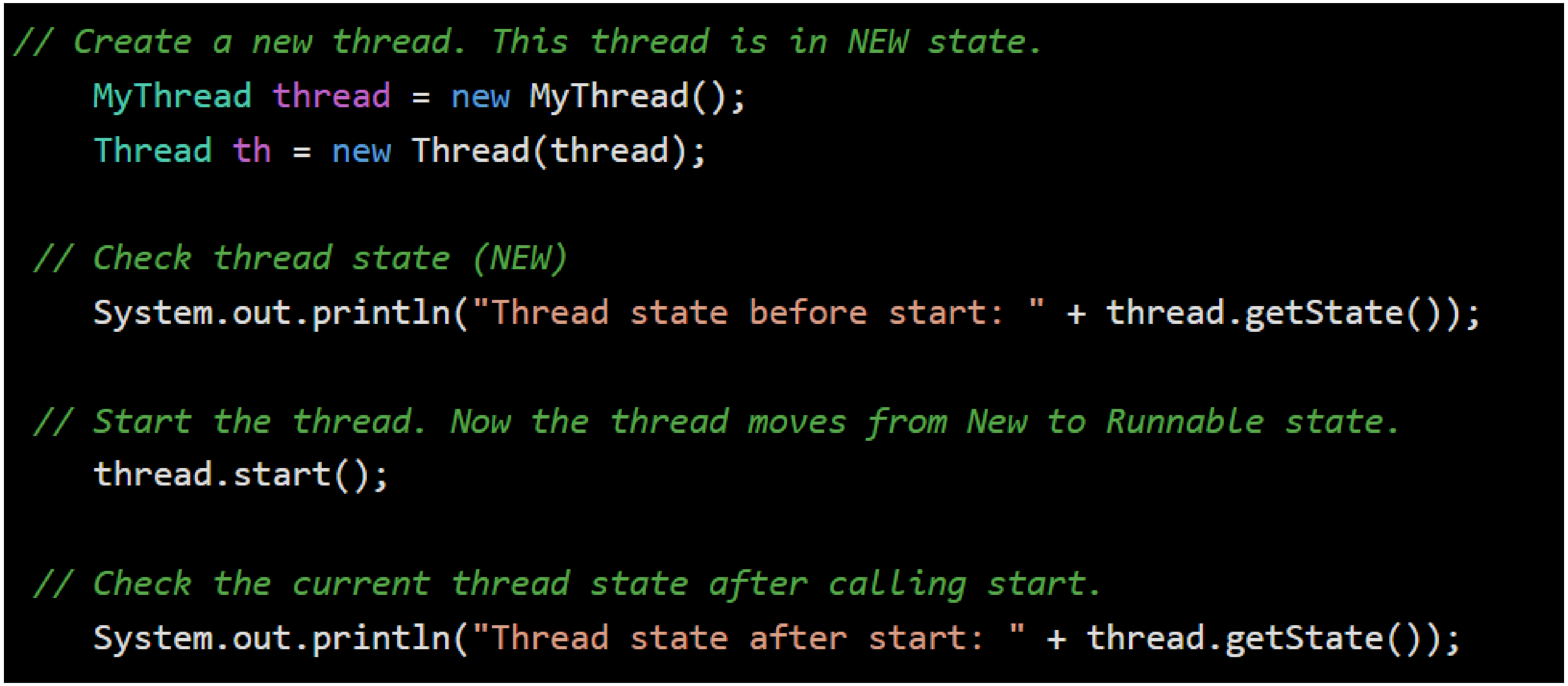
## RUNNABLE

**Runnable:** After start() is called, the thread becomes runnable. It is ready for execution and waiting for the thread scheduler to allocate CPU time.

Thread is ready to run (may be running or waiting for CPU)

```
t.start(); // RUNNABLE
```

# RUNNABLE

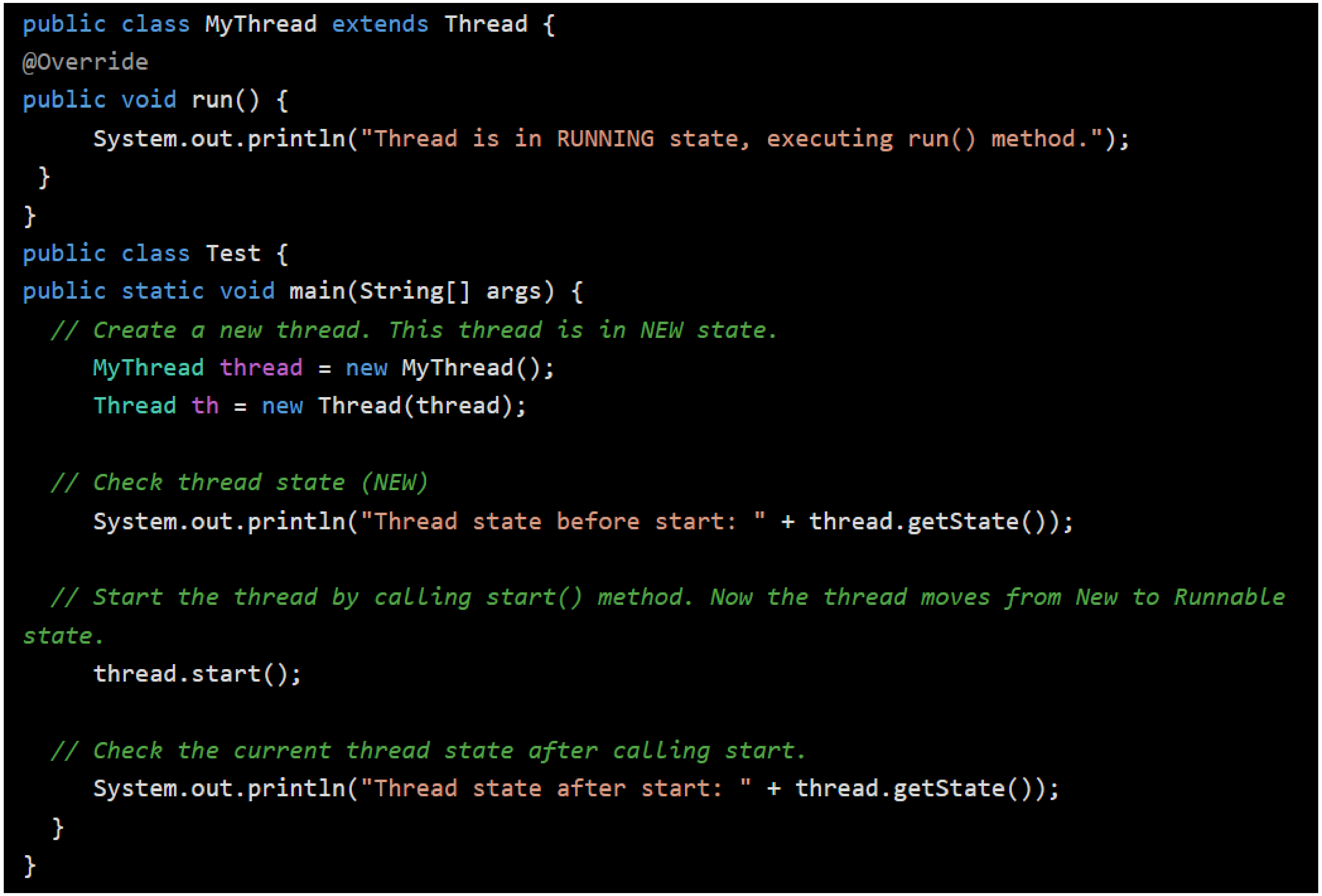


```
// Create a new thread. This thread is in NEW state.
    MyThread thread = new MyThread();
    Thread th = new Thread(thread);

 // Check thread state (NEW)
    System.out.println("Thread state before start: " + thread.getState());

 // Start the thread. Now the thread moves from New to Runnable state.
    thread.start();

 // Check the current thread state after calling start.
    System.out.println("Thread state after start: " + thread.getState());
```

## RUNNING

- The thread is actively executing its run() method.

# RUNNING



```java
public class MyThread extends Thread {
@Override
public void run() {
     System.out.println("Thread is in RUNNING state, executing run() method.");
 }
}
public class Test {
public static void main(String[] args) {
  // Create a new thread. This thread is in NEW state.
     MyThread thread = new MyThread();
     Thread th = new Thread(thread);

  // Check thread state (NEW)
     System.out.println("Thread state before start: " + thread.getState());

  // Start the thread by calling start() method. Now the thread moves from New to Runnable
state.
     thread.start();

  // Check the current thread state after calling start.
     System.out.println("Thread state after start: " + thread.getState());
  }
}
```

## BLOCKED

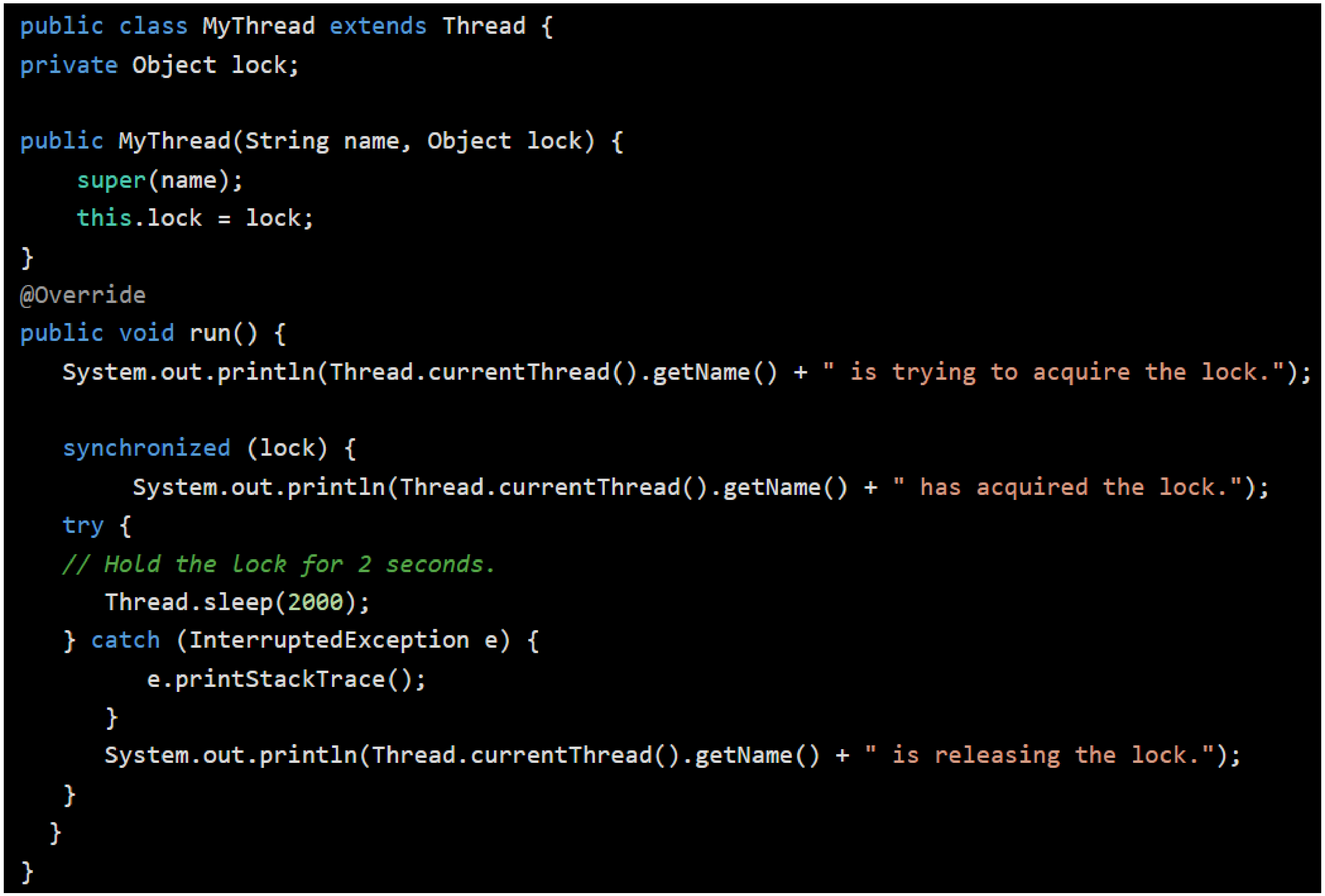
Thread is waiting to acquire a lock

A thread is blocked when it is trying to access a synchronized block or method that is currently held by another thread.

Happens in synchronized code

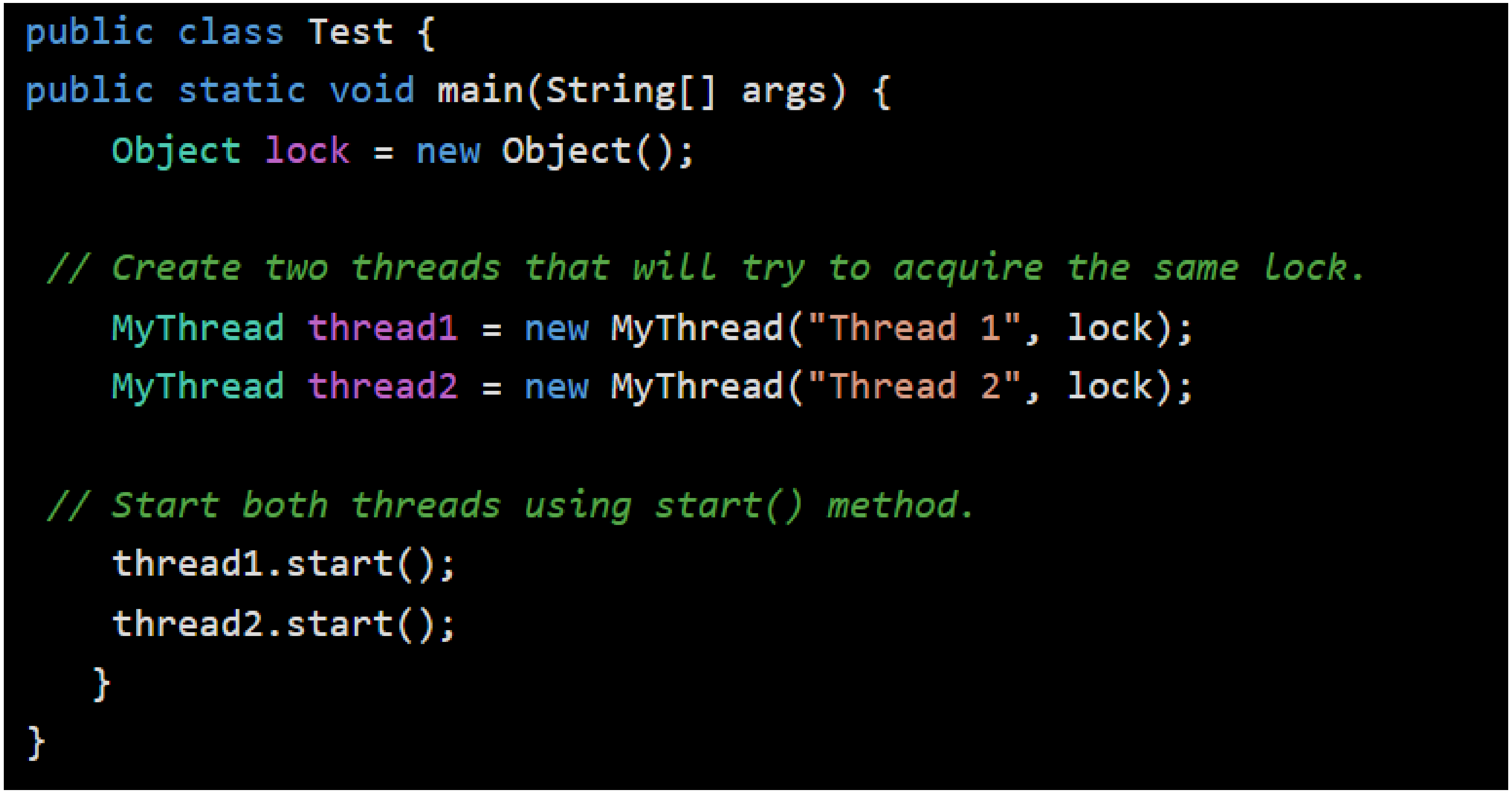
```
synchronized(obj) {
// thread may be BLOCKED
}
```

# Blocked



```java
public class MyThread extends Thread {
private Object lock;

public MyThread(String name, Object lock) {
    super(name);
    this.lock = lock;
}
@Override
public void run() {
   System.out.println(Thread.currentThread().getName() + " is trying to acquire the lock.");

   synchronized (lock) {
        System.out.println(Thread.currentThread().getName() + " has acquired the lock.");
   try {
   // Hold the lock for 2 seconds.
      Thread.sleep(2000);
   } catch (InterruptedException e) {
         e.printStackTrace();
      }
      System.out.println(Thread.currentThread().getName() + " is releasing the lock.");
   }
  }
}
```

## BLOCKED



```java
public class Test {
public static void main(String[] args) {
    Object lock = new Object();

 // Create two threads that will try to acquire the same lock.
    MyThread thread1 = new MyThread("Thread 1", lock);
    MyThread thread2 = new MyThread("Thread 2", lock);

 // Start both threads using start() method.
    thread1.start();
    thread2.start();
   }
}
```

## WAITING

- The thread is waiting indefinitely for another thread to perform a specific action (e.g., waiting for notify() or join()).
- Uses wait(), join()
- obj.wait();    // WAITING
- t.join();      // WAITING

## TIMED_ WAITING

The thread is waiting for another thread to perform an action for a specified waiting time (e.g., sleep(ms), wait(ms), join(ms)).

Thread waits for a fixed time

Uses sleep(), wait(timeout), join(timeout)

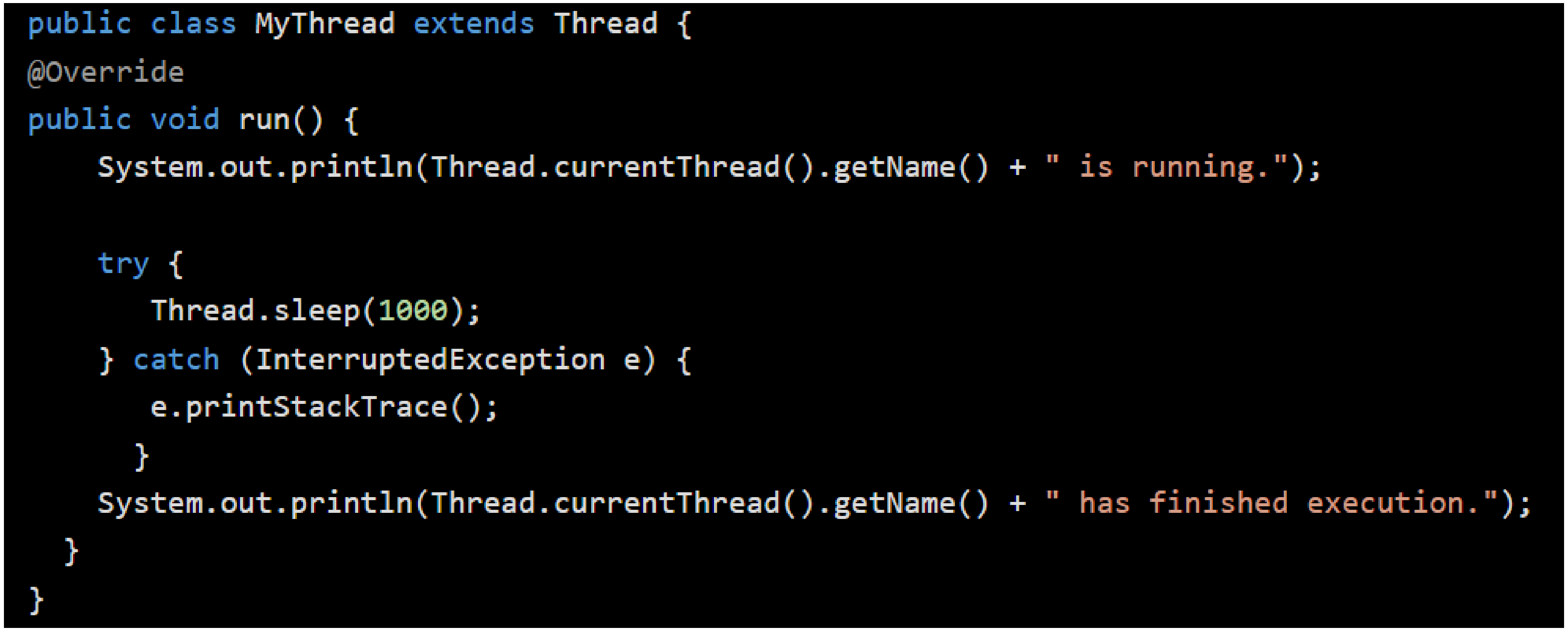
Thread.sleep(1000); // TIMED_WAITING

## TERMINATED (Dead)

The thread has finished execution or was terminated due to an exception.

```
// run() finished
```

## TERMINATED (Dead)

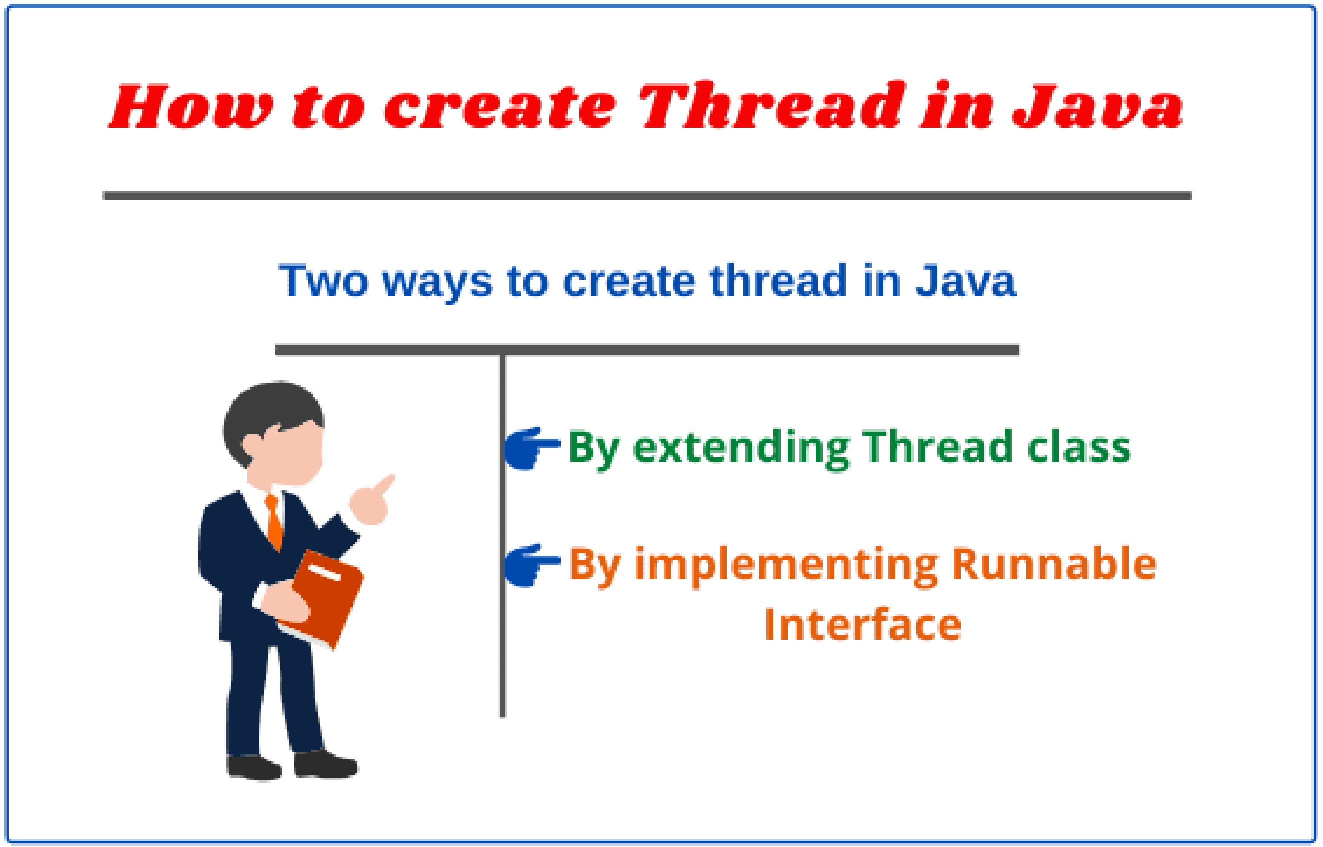


```
public class MyThread extends Thread {
@Override
public void run() {
    System.out.println(Thread.currentThread().getName() + " is running.");

    try {
       Thread.sleep(1000);
    } catch (InterruptedException e) {
       e.printStackTrace();
      }
    System.out.println(Thread.currentThread().getName() + " has finished execution.");
  }
}
```

## State Transition Methods

- **start()**: Moves a thread from New to Runnable.
- **sleep(long millis)**: Moves a thread to Timed Waiting.
- **wait()**: Moves a thread to Waiting.
- **join()**: Causes the current thread to wait until another thread finishes.
- **notify() / notifyAll()**: Wakes up threads in a Waiting state.

## Creating Threads



**How to create Thread in Java**

Two ways to create thread in Java

- By extending Thread class
- By implementing Runnable Interface

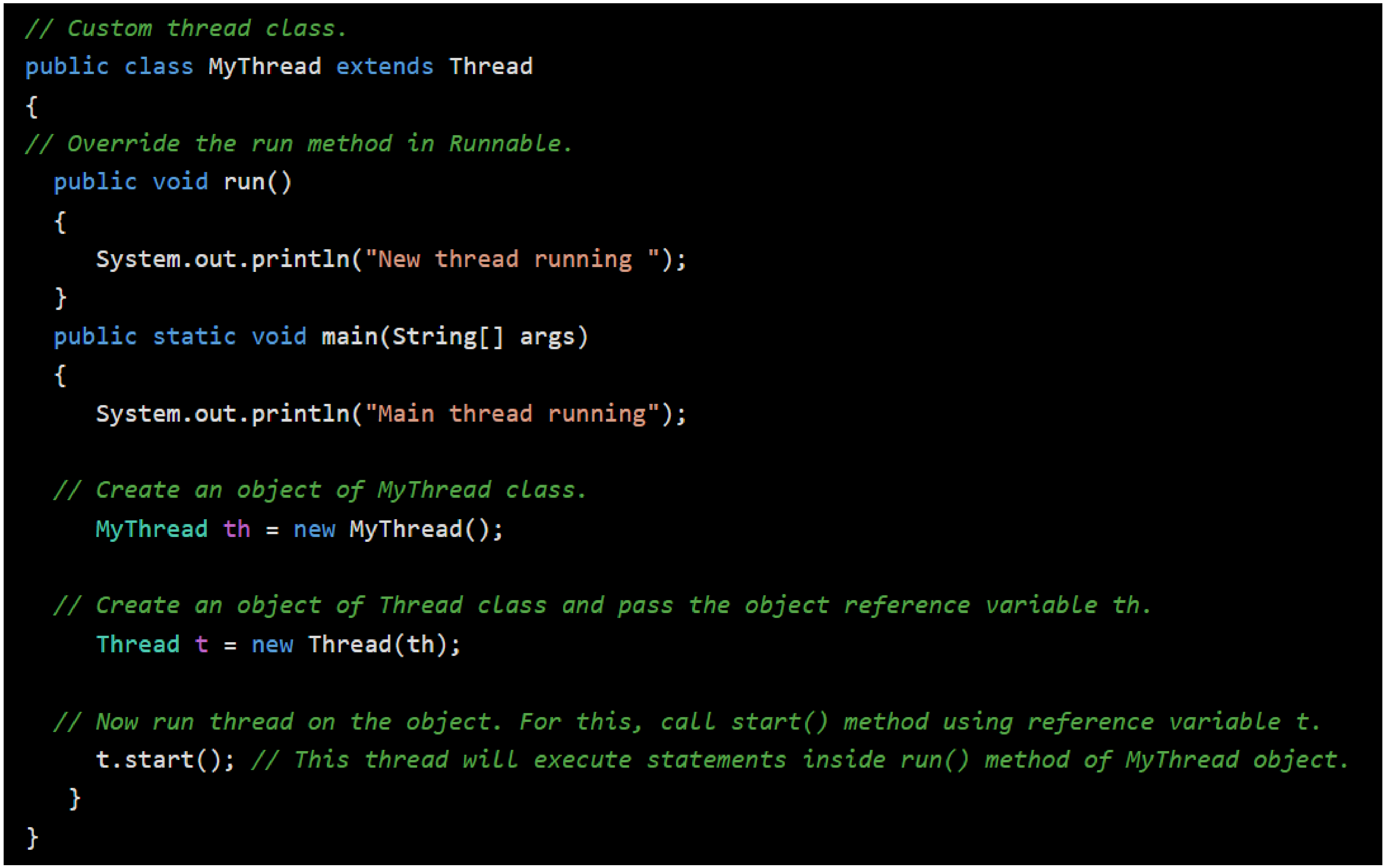
## Creating Threads

Extending Thread class.

Implementing Runnable interface.

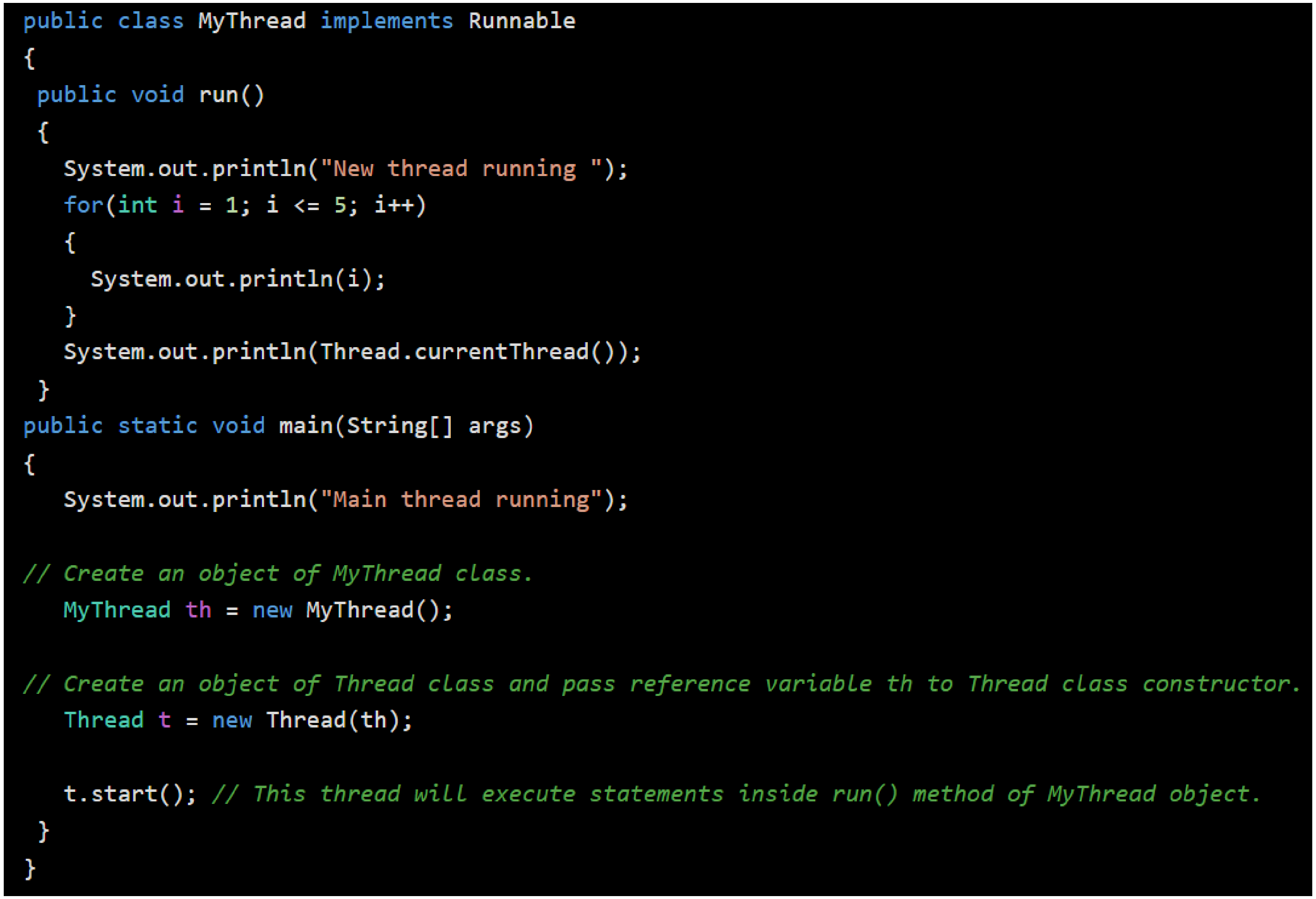
Start with .start(), not .run().

## Creating Threads - Extending Thread Class



```java
// Custom thread class.
public class MyThread extends Thread
{
// Override the run method in Runnable.
  public void run()
  {
     System.out.println("New thread running ");
  }
  public static void main(String[] args)
  {
     System.out.println("Main thread running");

  // Create an object of MyThread class.
     MyThread th = new MyThread();

  // Create an object of Thread class and pass the object reference variable th.
     Thread t = new Thread(th);

  // Now run thread on the object. For this, call start() method using reference variable t.
     t.start(); // This thread will execute statements inside run() method of MyThread object.
   }
}
```

## Creating Threads - Runnable Interface



```java
public class MyThread implements Runnable
{
 public void run()
 {
   System.out.println("New thread running ");
   for(int i = 1; i <= 5; i++)
   {
     System.out.println(i);
   }
   System.out.println(Thread.currentThread());
 }
public static void main(String[] args)
{
   System.out.println("Main thread running");

// Create an object of MyThread class.
   MyThread th = new MyThread();

// Create an object of Thread class and pass reference variable th to Thread class constructor.
   Thread t = new Thread(th);

   t.start(); // This thread will execute statements inside run() method of MyThread object.
 }
}
```

## Thread Sleep

Sometimes we need to make a thread sleep for a particular period of time. For this, we use the sleep() method in Java program. The sleep() method is a static method provided by the Java *Thread class* so it can be called by its class name.

The Thread.sleep() is used to sleep a current thread for a specified amount of time. It always pauses the current thread for a given period of time. The sleep() method controls the behavior of thread and transition of thread from one state to another.

When the sleep() method is called on Thread object, the thread is become out of the queue and enters into blocked (or non-runnable state) for a specified amount of time.

When the specified amount of time is elapsed, the thread does not go into the running state (execution state) immediately. It goes into the runnable state (ready state) until it is called by *Thread scheduler* of JVM. Look at the below figure.

## Thread Sleep

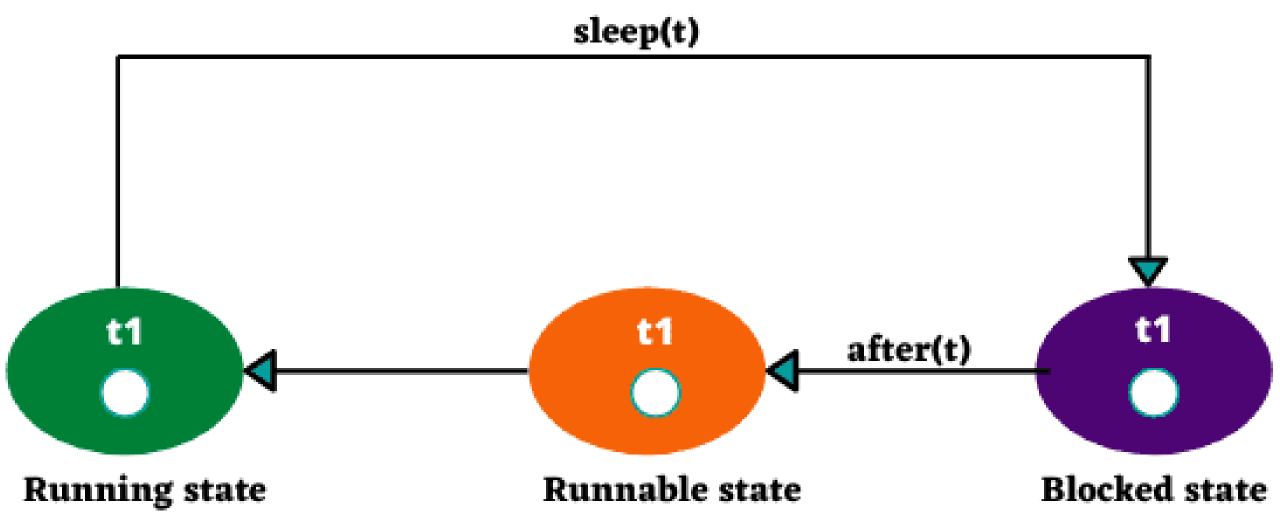
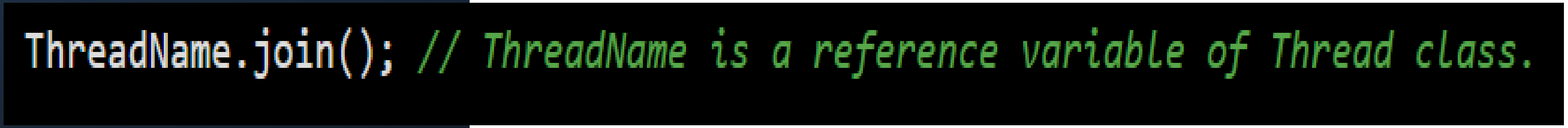


Fig: Life cycle of Thread when sleep() method is invoked

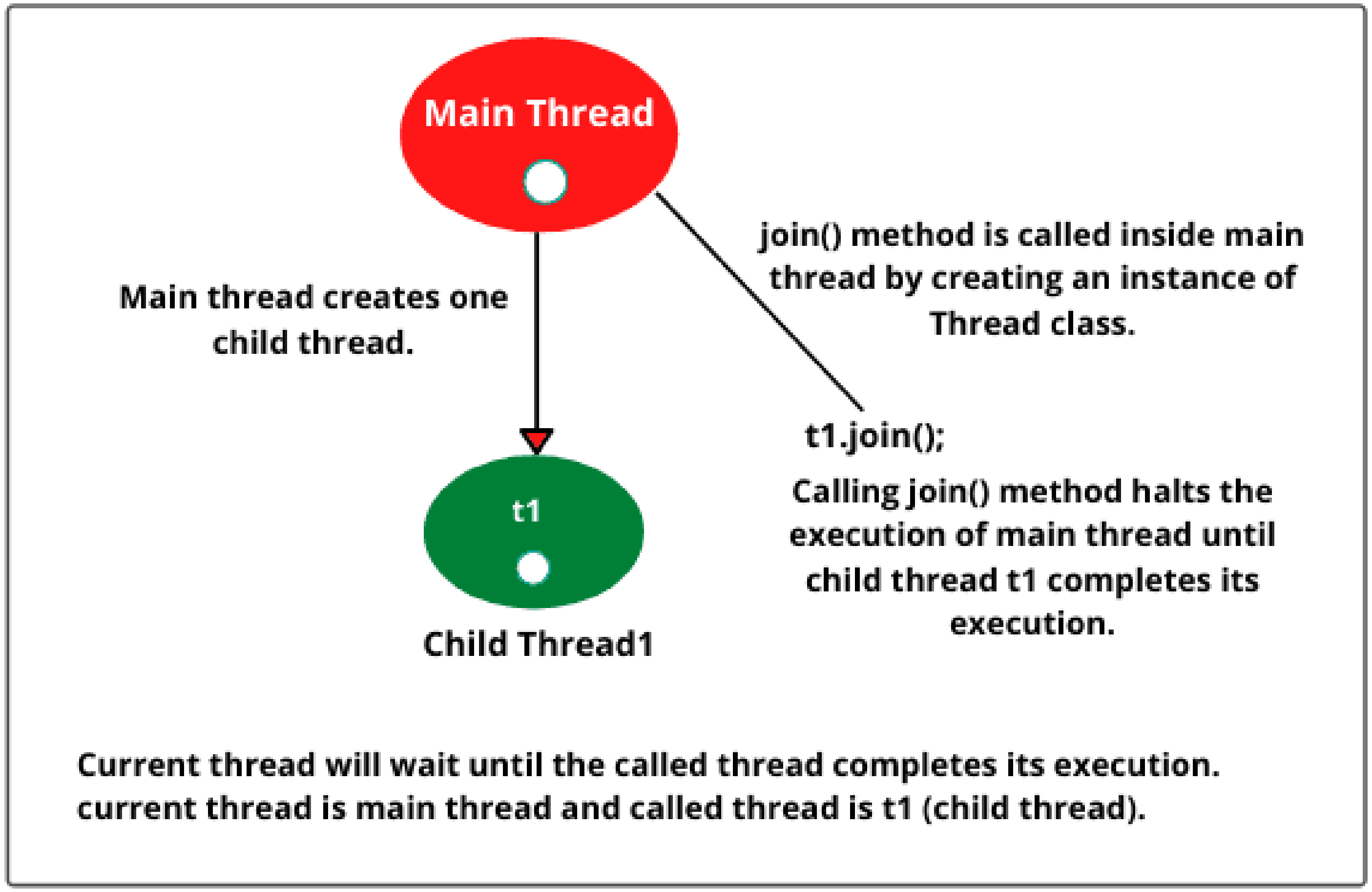
## Thread Join

- The **join() method in Java** is used when a thread needs to wait for another thread to finish its execution.
- In other words, this method is used to wait for the current thread until the thread that has called the join() method completes its execution.
- The join() method is an instance method, and it is always called by an object of *Thread class*.



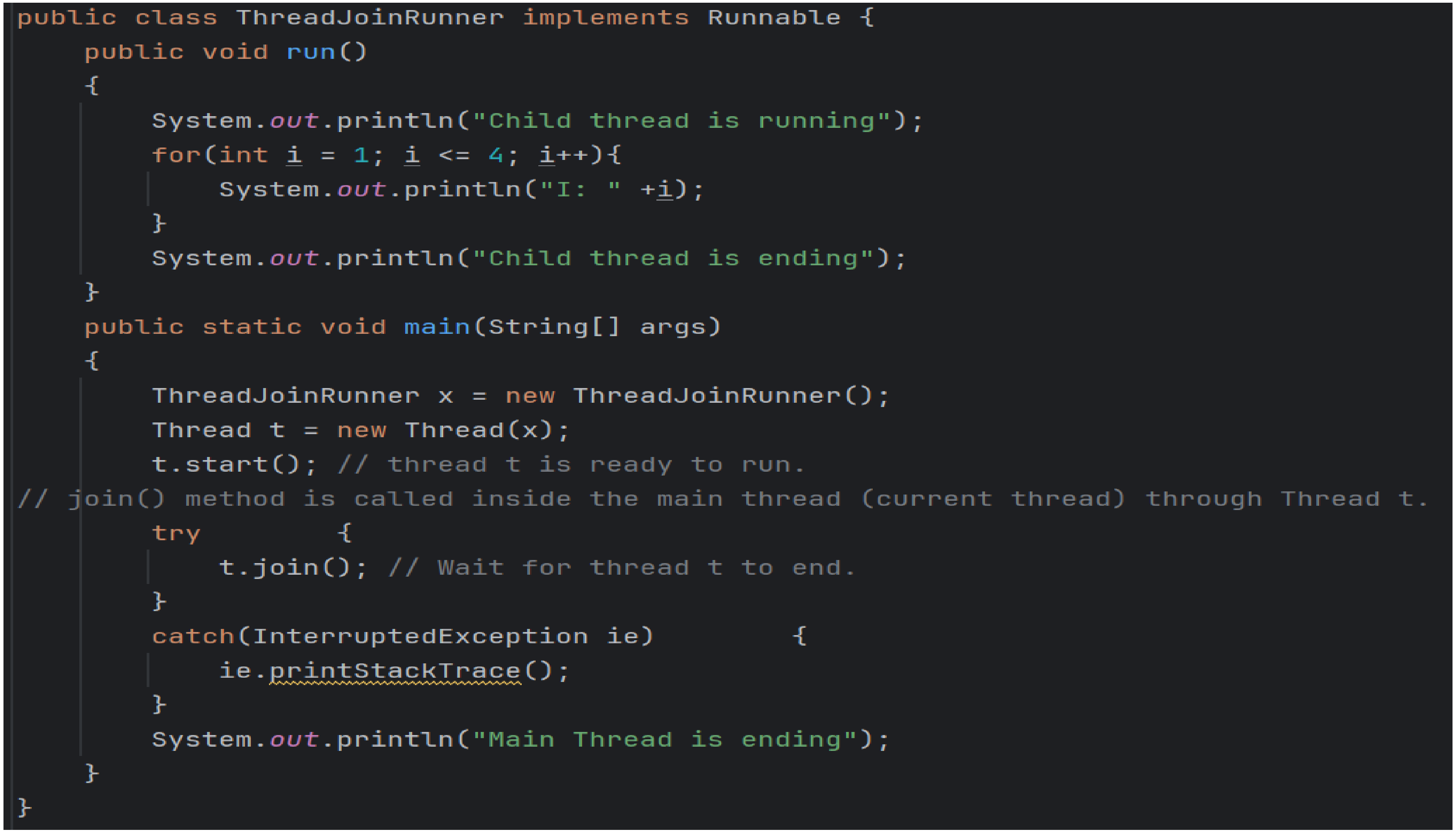
```
ThreadName.join(); // ThreadName is a reference variable of Thread class.
```

## Thread Join


Main Thread
Main thread creates one child thread.
join() method is called inside main thread by creating an instance of Thread class.
t1.join();
Calling join() method halts the execution of main thread until child thread t1 completes its execution.
t1
Child Thread1

Current thread will wait until the called thread completes its execution.
current thread is main thread and called thread is t1 (child thread).

# Thread Join



```java
public class ThreadJoinRunner implements Runnable {
    public void run()
    {
        System.out.println("Child thread is running");
        for(int i = 1; i <= 4; i++){
            System.out.println("I: " +i);
        }
        System.out.println("Child thread is ending");
    }
    public static void main(String[] args)
    {
        ThreadJoinRunner x = new ThreadJoinRunner();
        Thread t = new Thread(x);
        t.start(); // thread t is ready to run.
// join() method is called inside the main thread (current thread) through Thread t.
        try        {
            t.join(); // Wait for thread t to end.
        }
        catch(InterruptedException ie)        {
            ie.printStackTrace();
        }
        System.out.println("Main Thread is ending");
    }
}
```

## Thread Wait

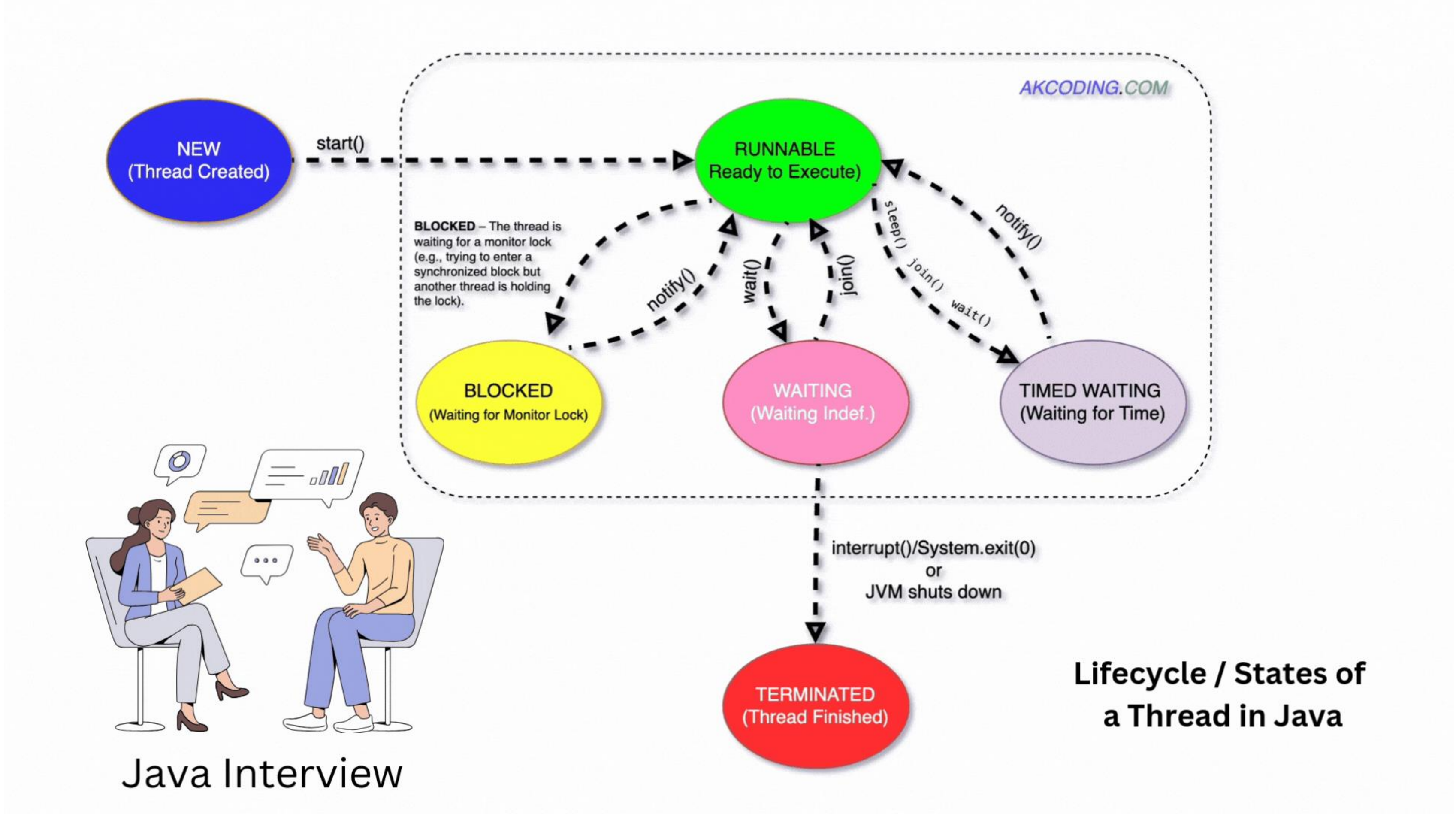
wait() is used to make the **current thread pause execution** until **another thread notifies it.**

It is mainly used for **inter-thread communication.**

## Key points about wait()

- Defined in **Object class** (not Thread)
- Must be called **inside a synchronized block/method**
- **Releases the lock** of the object
- Thread goes into **WAITING** state
- Thread wakes up only when:
- notify() or notifyAll() is called
- or thread is interrupted

# States of a Thread


AKCODING.COM
NEW (Thread Created)
start()
RUNNABLE Ready to Execute)
BLOCKED – The thread is waiting for a monitor lock (e.g., trying to enter a synchronized block but another thread is holding the lock).
notify()
wait()
join()
sleep()
join()
wait()
notify()
BLOCKED (Waiting for Monitor Lock)
WAITING (Waiting Indef.)
TIMED WAITING (Waiting for Time)
interrupt()/System.exit(0) or JVM shuts down
TERMINATED (Thread Finished)
Lifecycle / States of a Thread in Java
Java Interview

## The Executor Framework

- Thread Pools manage worker threads.
- Fixed vs. Cached pools.
- Avoids the overhead of manual creation.

## The Executor Framework

Race Condition: Data corruption from simultaneous access.

Synchronized keyword: The 'Lock'.

Intrinsic Locks (Monitors).

## Synchronization & Race Conditions

- Race Condition: Data corruption from simultaneous access.
- Synchronized keyword: The 'Lock'.
- Intrinsic Locks (Monitors).

## Locks & Atomic Variables

- ReentrantLock: Flexible locking.
- Atomic Variables: Lock-free safety.
- CAS (Compare-and-Swap).

## CompletableFuture (Async)

Non-blocking pipelines.

supplyAsync, thenApply, thenAccept.

Handling errors with .exceptionally().

## Fork/Join Framework

- Divide and Conquer.
- Fork: Split tasks.
- Join: Combine results.

## Fork/Join: Action vs Task

RecursiveAction: Returns nothing.

RecursiveTask: Returns a result.

ForkJoinPool: The specialized engine.

# Best Practices

| | |
|---|---|
| Shut down | Always shut down Executors. |
| Minimize | Minimize synchronized block size. |
| Use | Use Concurrent collections. |