# **Threads in Java**

## **Meaning**

A **Java thread** is the smallest unit of execution within a program. It acts as a lightweight subprocess, running independently but sharing memory with other threads of the same process, supporting concurrent task execution.

## **Features of Threads**

* **Concurrency**: Enables multiple tasks to run simultaneously, making applications more efficient and responsive.
* **Shared Memory**: Threads share the address space of their parent process, allowing fast communication but requiring thread-safe programming.
* **Lightweight**: Creating threads incurs less overhead than creating processes.
* **Responsiveness**: Java threads help maintain responsiveness in applications (e.g., user interface, servers) even when some tasks are blocked or executing lengthy operations.
* **Multitasking**: Allows parallel operations, such as input handling and data processing.

## **Life Cycle of a Thread**

Java threads transition through distinct states during their lifetime:

* **New**: Thread object created, not started.
* **Active**: After start(), the thread is either:
  + **Runnable**: Ready to run, awaiting CPU.
  + **Running**: Actively executing.
* **Waiting/Blocked**: Temporarily inactive, waiting for resources or conditions.
* **Timed Waiting**: Paused for a specific period (via sleep() or timed waits).
* **Terminated**: After completing or being stopped; cannot be restarted.

Threads move between states based on method calls (e.g., start, join, sleep), resource availability, and scheduler actions.

## **Creating Threads**

There are two primary ways to create threads in Java:

## **1. Extending the Thread Class**

Create a class that extends Thread, override the run() method, and invoke start() to run the thread independently.

Java

class MyThread extends Thread {  
 public void run() {  
 System.out.println("Thread Started Running...");  
 }  
}  
  
public class Geeks {  
 public static void main(String[] args) {  
 MyThread t1 = new MyThread();  
 t1.start();  
 }  
}

## **2. Implementing Runnable Interface**

Implement the Runnable interface, override run(), and pass your object to a Thread instance, calling start() to execute.

Java

class MyThread implements Runnable {  
 public void run() {  
 System.out.println("Thread is Running Successfully");  
 }  
}  
public class Geeks {  
 public static void main(String[] args) {  
 MyThread g1 = new MyThread();  
 Thread t1 = new Thread(g1);  
 t1.start();  
 }  
}

**Recommendation**: Extend Thread only if the class doesn't need to extend another; prefer Runnable for better modularity and design.

## **Thread Class API**

## **Constructors**

|  |  |
| --- | --- |
| **Constructor** | **Action Performed** |
| Thread() | New thread object |
| Thread(Runnable target) | With a Runnable |
| Thread(Runnable target, String name) | Runnable & named |
| Thread(String name) | Named thread |
| Thread(ThreadGroup group, Runnable target) | In thread group |
| ... | Other variants |

## **Important Methods**

|  |  |
| --- | --- |
| **Method** | **Action** |
| start() | Starts the thread |
| run() | Thread entry point |
| sleep(long ms) | Pauses thread |
| join() | Waits for the thread to finish |
| interrupt() | Interrupts the thread |
| setPriority(int) | Sets priority |
| isAlive() | Checks if thread is running |
| setDaemon(boolean) | Sets daemon/user mode |
| getState() | Gets thread state |
| yield() | Scheduler hint to pause and let others execute |

## **Types of Threads in Java**

* **User Threads**: Main threads running client code.
* **Daemon Threads**: Background service threads (e.g., garbage collection). They die when all user threads finish.

## **Thread Priorities**

Each thread has a priority from 1 (MIN\_PRIORITY) to 10 (MAX\_PRIORITY), default is 5 (NORM\_PRIORITY). Threads with higher priority may get CPU preference, but exact scheduling is JVM-dependent.

## **Concurrency Issues**

Common multithreading problems include:

* **Race Conditions**: Multiple threads modifying shared data, causing unpredictable results.
* **Deadlocks**: Threads locking each other out waiting for resources.
* **Livelocks**: Threads actively yielding, but making no progress.
* **Starvation**: Threads never get CPU time due to scheduler choices.
* **Priority Inversion**: Low-priority thread holds lock needed by a high-priority thread.

## **Synchronization & Thread Safety**

Use synchronization (synchronized keyword), locks, atomic variables, thread-safe collections, and pools to handle concurrency safely.

**Example: Thread-Safe Counter**

Java

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

## **Thread States Example**

Java

Thread t1 = new MyThread();  
System.out.println(t1.getState()); *// NEW*  
t1.start();  
System.out.println(t1.getState()); *// RUNNABLE*  
t1.join();  
System.out.println(t1.getState()); *// TERMINATED*

## **Best Practices**

* Always use start() to run threads concurrently, not run() directly.
* Design for thread safety (especially with shared resources).
* Use thread pools for managing many threads efficiently.
* Prefer implementing Runnable for modular, maintainable code.

This documentation provides a reference guide for understanding, designing, and working with threads in Java, suitable for both learners and advanced programmers.