Java Multithreading

Presented by Rama Shanker

[**Date**]: 24/01/2025

What is Multithreading?

Definition:

The ability to execute multiple threads (smallest units of a process) concurrently.

Why Multithreading?

Better utilization of CPU resources.

Parallel execution for faster performance.

Allows responsiveness in applications (e.g., GUI).

Threads in Java

Thread: Lightweight process.

Java Support:

java.lang.Thread class.

Java.lang.Runnable interface.

Main Thread: Every Java application starts with a single thread - the main thread.

Creating Threads

Two Ways to Create Threads:

Extend the Thread class.

Implement the Runnable interface.

Extending the Thread Class

Code Example:

```
public class MyThread extends Thread {
    public void run() {
        System.out.println("Thread is running...");
    }

    public static void main(String[] args) {
        MyThread thread = new MyThread();
        thread.start(); // Starts the thread
    }
}
```

Key Methods:

start(): Starts the thread.

run(): Contains the logic for the thread.

Implementing the Runnable Interface

Code Example:

```
public class MyRunnable implements Runnable {
    public void run() {
        System.out.println("Thread is running...");
    }

    public static void main(String[] args) {
        Thread thread = new Thread(new MyRunnable());
        thread.start(); // Starts the thread
    }
}
```

Why Use Runnable?

Better for cases where a class needs to extend another class.

Thread Lifecycle

States of a Thread:

New: Thread is created but not started.

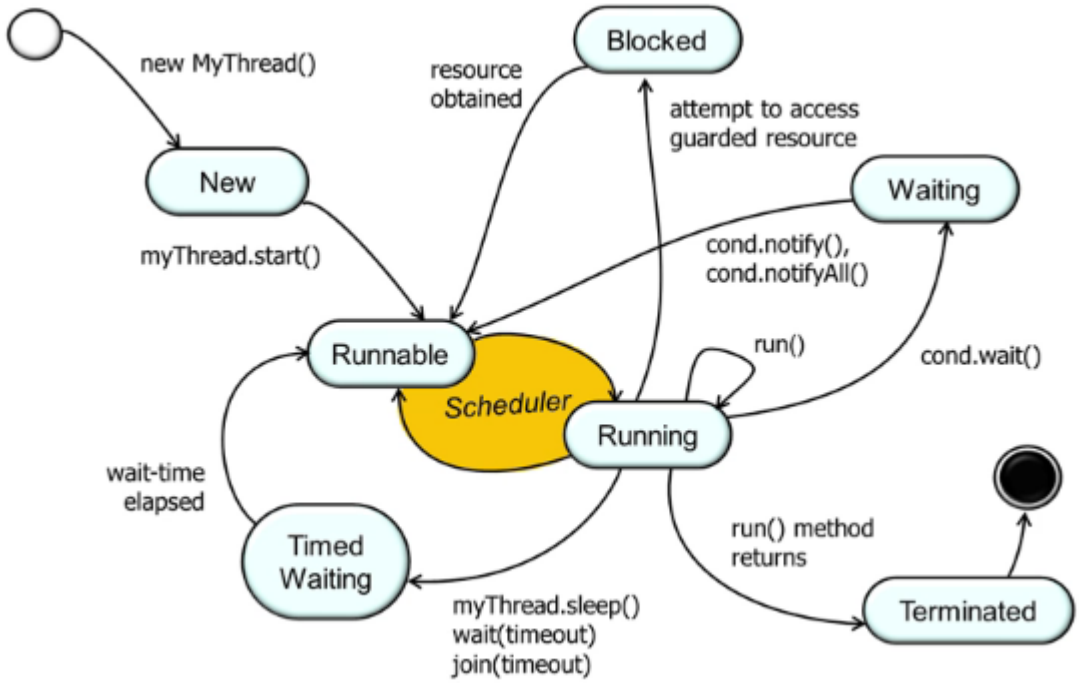
Runnable: Ready to run but waiting for CPU time.

Running: Thread is executing.

Blocked/Waiting: Thread is waiting for resources.

Terminated: Thread has completed execution.

Diagram


new MyThread()
Blocked
resource
obtained
attempt to access
guarded resource
New
Waiting
myThread.start()
cond.notify(),
cond.notifyAll()
Runnable
run()
cond.wait()
Scheduler
Running
wait-time
elapsed
Timed
Waiting
run() method
returns
Terminated
myThread.sleep()
wait(timeout)
join(timeout)

Thread Synchronization

Why Synchronize?

To prevent race conditions when multiple threads access shared resources.

Synchronized Keyword:

Locks the resource so only one thread can access it at a time.

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

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

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

Inter-Thread Communication

Purpose: Allows threads to communicate and coordinate their actions.

Methods:

wait(): Causes the current thread to wait until another thread invokes notify().

notify(): Wakes up a single waiting thread.

notifyAll(): Wakes up all waiting threads.

```
synchronized(obj) {
    obj.wait(); // Wait for notification
    obj.notify(); // Notify one thread
}
```

Thread Pools

Definition: A group of pre-instantiated threads available for use.

Advantages:

Efficient thread management.

Prevents overhead of thread creation/destruction.

Example with Executor Service:

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class ThreadPoolExample {
    public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(3);

        for (int i = 0; i < 5; i++) {
            executor.execute(() -> {
                System.out.println("Thread " + Thread.currentThread().getName() + " is running");
            });
        }

        executor.shutdown();
    }
}
```

Multithreading Challenges

Race Conditions:

Occurs when multiple threads modify shared resources concurrently.

Deadlocks:

Occurs when two threads are waiting on each other to release locks.

Starvation:

A thread is unable to execute because other threads consume all resources.

Best Practices for Multithreading

Minimize use of shared resources.

Use thread-safe collections (e.g., ConcurrentHashMap).

Use thread pools for managing multiple threads.

Avoid deadlocks by acquiring locks in a consistent order.

Summary

Multithreading allows parallel execution and efficient CPU utilization.

Use synchronization and thread pools for better thread management.

Handle challenges like race conditions and deadlocks carefully.