Multithreading & Concurrency

Advanced Java Programming

Masoud Hamad

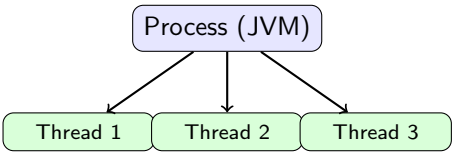
State University of Zanzibar (SUZA)

2025/2026 Academic Year

Outline

Why Multithreading?

- **Responsiveness** — keep UI responsive while performing background tasks
- **Performance** — utilize multi-core CPUs effectively
- **Resource sharing** — threads share memory within a process
- **Scalability** — handle multiple clients simultaneously (e.g., web servers)
- **Asynchronous I/O** — continue processing while waiting for file or network operations



Thread vs Process

A **process** has its own memory space. **Threads** within a process share the same memory (heap) but have their own stack.

Creating Threads — Extending Thread

```
class MyThread extends Thread {
    private String taskName;

    public MyThread(String taskName) {
        this.taskName = taskName;
    }

    @Override
    public void run() {
        for (int i = 1; i <= 5; i++) {
            System.out.println(taskName + " - Count: " + i);
            try {
                Thread.sleep(500);  // pause 500ms
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
    }
}

// Usage
MyThread t1 = new MyThread("Task A");
MyThread t2 = new MyThread("Task B");
```

Creating Threads — Implementing `Runnable`

```
class MyRunnable implements Runnable {
    private String taskName;

    public MyRunnable(String taskName) {
        this.taskName = taskName;
    }

    @Override
    public void run() {
        for (int i = 1; i <= 5; i++) {
            System.out.println(taskName + " - Count: " + i);
            try { Thread.sleep(500); }
            catch (InterruptedException e) { Thread.currentThread().interrupt(); }
        }
    }
}

// Usage
Thread t1 = new Thread(new MyRunnable("Task A"));
Thread t2 = new Thread(new MyRunnable("Task B"));
t1.start();
t2.start();
```

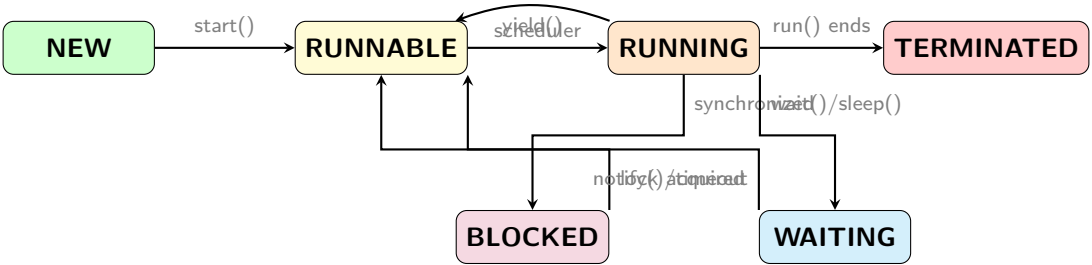
Thread vs Runnable — Comparison

Aspect	**extends Thread**	**implements Runnable**
Inheritance	Cannot extend another class	Can extend another class
Reusability	Tightly coupled	Loosely coupled, reusable
Resource sharing	Harder to share	Easier to share state
Thread pool	Not compatible	Works with ExecutorService
Lambda	Not applicable	Supported (functional interface)
Best practice	Rarely used	**Preferred approach**

Recommendation

Always prefer `Runnable` (or `Callable`) over extending `Thread`. It follows the principle of *composition over inheritance* and allows use with thread pools.

Thread Lifecycle States



Essential Thread Methods

```
Thread t = new Thread(() -> {
    System.out.println("Running in: " + Thread.currentThread().getName());
});

t.setName("Worker-1");          // set thread name
t.setPriority(Thread.MAX_PRIORITY);  // 1 (MIN) to 10 (MAX), default 5
t.setDaemon(true);              // daemon thread dies when main thread exits
t.start();                      // start thread execution
```

```
// join() - wait for thread to finish
Thread worker = new Thread(() -> {
    try { Thread.sleep(2000); }
    catch (InterruptedException e) { Thread.currentThread().interrupt(); }
    System.out.println("Work done!");
});
worker.start();
System.out.println("Waiting for worker...");
worker.join();  // blocks until worker completes
System.out.println("Worker finished, continuing main thread.");
```

sleep, yield, and interrupt

```
// sleep() - pause current thread for specified milliseconds
Thread.sleep(1000);  // pauses for 1 second

// yield() - hint to scheduler to give other threads a chance
Thread.yield();  // may or may not be honored by the scheduler

// interrupt() - request a thread to stop
Thread worker = new Thread(() -> {
    while (!Thread.currentThread().isInterrupted()) {
        System.out.println("Working...");
        try {
            Thread.sleep(500);
        } catch (InterruptedException e) {
            System.out.println("Interrupted! Cleaning up...");
            Thread.currentThread().interrupt(); // re-set flag
            break;
        }
    }
    System.out.println("Thread exiting gracefully.");
});

worker.start();
Thread.sleep(2000);
```

Daemon Threads

```
public class DaemonExample {
    public static void main(String[] args) throws InterruptedException {
        Thread daemon = new Thread(() -> {
            while (true) {
                System.out.println("Daemon: background cleanup...");
                try { Thread.sleep(1000); }
                catch (InterruptedException e) { break; }
            }
        });
        daemon.setDaemon(true);  // must set before start()
        daemon.start();

        // Main thread does some work
        Thread.sleep(3000);
        System.out.println("Main thread ending...");
        // Daemon thread is automatically terminated when main exits
    }
}
```

Daemon vs User Threads

User threads keep the JVM alive. **Daemon threads** (e.g., garbage collector) run in the

Race Condition — The Problem

```
class Counter {
    private int count = 0;
    public void increment() { count++; }  // NOT thread-safe!
    public int getCount()   { return count; }
}

public class RaceConditionDemo {
    public static void main(String[] args) throws InterruptedException {
        Counter counter = new Counter();

        Thread t1 = new Thread(() -> {
            for (int i = 0; i < 100000; i++) counter.increment();
        });
        Thread t2 = new Thread(() -> {
            for (int i = 0; i < 100000; i++) counter.increment();
        });

        t1.start(); t2.start();
        t1.join();  t2.join();

        // Expected: 200000, but often less due to race condition!
        System.out.println("Count: " + counter.getCount());
    }
```

Synchronized Keyword

```
class SynchronizedCounter {
    private int count = 0;

    // Synchronized method: locks on 'this' object
    public synchronized void increment() {
        count++;
    }

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

```
class Counter {
    private int count = 0;
    private final Object lock = new Object();

    public void increment() {
        // Synchronized block: more fine-grained control
        synchronized (lock) {
            count++;
        }
    }
```

wait(), notify(), and notifyAll()

These methods enable **inter-thread communication** and must be called within a `synchronized` block:

```
class SharedBuffer {
    private int data;
    private boolean hasData = false;

    public synchronized void produce(int value) throws InterruptedException {
        while (hasData) {
            wait();  // release lock and wait until consumed
        }
        data = value;
        hasData = true;
        System.out.println("Produced: " + value);
        notify();    // wake up a waiting consumer
    }

    public synchronized int consume() throws InterruptedException {
        while (!hasData) {
            wait();  // release lock and wait until produced
        }
        hasData = false;
        System.out.println("Consumed: " + data);
```

wait/notify — Producer and Consumer

```
public class WaitNotifyDemo {
    public static void main(String[] args) {
        SharedBuffer buffer = new SharedBuffer();

        Thread producer = new Thread(() -> {
            try {
                for (int i = 1; i <= 5; i++) {
                    buffer.produce(i);
                    Thread.sleep(500);
                }
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        });

        Thread consumer = new Thread(() -> {
            try {
                for (int i = 1; i <= 5; i++) {
                    buffer.consume();
                    Thread.sleep(800);
                }
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
```

ExecutorService — Thread Pools

```
import java.util.concurrent.*;

public class ExecutorDemo {
    public static void main(String[] args) {
        // Create a fixed thread pool with 3 threads
        ExecutorService executor = Executors.newFixedThreadPool(3);

        for (int i = 1; i <= 6; i++) {
            final int taskId = i;
            executor.submit(() -> {
                String threadName = Thread.currentThread().getName();
                System.out.println("Task " + taskId + " on " + threadName);
                try { Thread.sleep(1000); }
                catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                }
            });
        }

        executor.shutdown();  // no new tasks; finish existing ones
        // executor.shutdownNow();  // attempt to cancel running tasks
    }
}
```

Types of Thread Pools

Factory Method	Description
`newFixedThreadPool(n)`	Pool with exactly `n` threads; queues excess tasks
`newCachedThreadPool()`	Creates threads as needed, reuses idle ones (60s timeout)
`newSingleThreadExecutor()`	Single thread; tasks execute sequentially
`newScheduledThreadPool(n)`	For delayed or periodic task execution

Best Practices

- Always call `shutdown()` when done with the executor
- Use `try-finally` or try-with-resources (Java 19+) to ensure shutdown
- Choose pool size based on workload: CPU-bound $\approx$ number of cores; I/O-bound $\approx$ higher

Callable and Future

Unlike `Runnable`, `Callable<V>` can return a result and throw checked exceptions:

```
import java.util.concurrent.*;

public class CallableDemo {
    public static void main(String[] args) throws Exception {
        ExecutorService executor = Executors.newFixedThreadPool(2);

        Callable<Integer> factorial = () -> {
            int result = 1;
            for (int i = 1; i <= 10; i++) {
                result *= i;
                Thread.sleep(100);
            }
            return result;
        };

        Future<Integer> future = executor.submit(factorial);

        System.out.println("Doing other work...");

        // get() blocks until result is available
        Integer result = future.get(5, TimeUnit.SECONDS);
        System.out.println("10! = " + result);  // 3628800
```

CountDownLatch

A synchronization aid that allows one or more threads to wait until a set of operations completes:

```
import java.util.concurrent.CountDownLatch;

public class CountDownLatchDemo {
    public static void main(String[] args) throws InterruptedException {
        int numWorkers = 3;
        CountDownLatch latch = new CountDownLatch(numWorkers);

        for (int i = 1; i <= numWorkers; i++) {
            final int id = i;
            new Thread(() -> {
                System.out.println("Worker " + id + " starting...");
                try { Thread.sleep((long)(Math.random() * 2000)); }
                catch (InterruptedException e) { Thread.currentThread().interrupt();
                     }
                System.out.println("Worker " + id + " done.");
                latch.countDown();  // decrement count
            }).start();
        }

        System.out.println("Main: waiting for all workers...");
```

ConcurrentHashMap

Thread-safe alternative to `HashMap` without the overhead of `Collections.synchronizedMap`:

```
import java.util.concurrent.ConcurrentHashMap;

public class ConcurrentMapDemo {
    public static void main(String[] args) throws InterruptedException {
        ConcurrentHashMap<String, Integer> wordCount = new ConcurrentHashMap<>();

        Runnable counter = () -> {
            String[] words = {"java", "thread", "java", "stream", "thread"};
            for (String word : words) {
                wordCount.merge(word, 1, Integer::sum);
                // atomically: if key exists, apply function; else insert
            }
        };

        Thread t1 = new Thread(counter);
        Thread t2 = new Thread(counter);
        t1.start(); t2.start();
        t1.join();  t2.join();

        wordCount.forEach((k, v) -> System.out.println(k + ": " + v));
```

AtomicInteger and Atomic Classes

Lock-free thread-safe operations on single variables:

```
import java.util.concurrent.atomic.AtomicInteger;

public class AtomicDemo {
    public static void main(String[] args) throws InterruptedException {
        AtomicInteger counter = new AtomicInteger(0);

        Runnable task = () -> {
            for (int i = 0; i < 100000; i++) {
                counter.incrementAndGet();  // atomic operation
            }
        };

        Thread t1 = new Thread(task);
        Thread t2 = new Thread(task);
        t1.start(); t2.start();
        t1.join();  t2.join();

        // Always prints 200000 (no race condition!)
        System.out.println("Count: " + counter.get());
    }
}
```

BlockingQueue — Producer-Consumer

`BlockingQueue` handles synchronization automatically:

```
import java.util.concurrent.*;

public class ProducerConsumerDemo {
    public static void main(String[] args) {
        BlockingQueue<Integer> queue = new ArrayBlockingQueue<>(5);

        // Producer
        Thread producer = new Thread(() -> {
            try {
                for (int i = 1; i <= 10; i++) {
                    queue.put(i);  // blocks if queue is full
                    System.out.println("Produced: " + i);
                    Thread.sleep(200);
                }
                queue.put(-1);  // poison pill to signal completion
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        });
```

BlockingQueue — Consumer Side

```
        // Consumer
        Thread consumer = new Thread(() -> {
            try {
                while (true) {
                    int value = queue.take();  // blocks if queue is empty
                    if (value == -1) break;    // poison pill received
                    System.out.println("Consumed: " + value);
                    Thread.sleep(500);
                }
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        });

        producer.start();
        consumer.start();
    }
}
```

BlockingQueue Implementations

- `ArrayBlockingQueue` — bounded, backed by array

Multiple Producers and Consumers with ExecutorService

```
public class MultiProducerConsumer {
    public static void main(String[] args) {
        BlockingQueue<String> queue = new LinkedBlockingQueue<>(10);
        ExecutorService executor = Executors.newFixedThreadPool(5);

        // 2 producers
        for (int p = 1; p <= 2; p++) {
            final int id = p;
            executor.submit(() -> {
                try {
                    for (int i = 1; i <= 5; i++) {
                        String item = "P" + id + "-Item" + i;
                        queue.put(item);
                        System.out.println("Produced: " + item);
                    }
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                }
            });
        }

        // 3 consumers
        for (int c = 1; c <= 3; c++) {
```

Common Concurrency Pitfalls

Deadlock

- Two threads each hold a lock the other needs
- Both wait forever
- Prevention: always acquire locks in the same order

Starvation

- A thread never gets CPU time
- Caused by priority inversion or greedy threads

Race Condition

- Outcome depends on thread scheduling
- Fix with synchronization or atomic operations

Livelock

- Threads keep responding to each other but make no progress
- Similar to two people in a hallway stepping aside in the same direction

Summary

Core Threading

- Thread creation: `Runnable` preferred over `Thread`
- Lifecycle: NEW $\rightarrow$ RUNNABLE $\rightarrow$ RUNNING $\rightarrow$ TERMINATED
- Key methods: start, join, sleep, interrupt
- Synchronization with `synchronized`
- Inter-thread communication: wait/notify

java.util.concurrent

- `ExecutorService` for thread pools
- `Callable`/`Future` for returning results
- `CountDownLatch` for coordination
- `ConcurrentHashMap` for thread-safe maps
- `AtomicInteger` for lock-free counters
- `BlockingQueue` for producer-consumer

Best Practices

Prefer high-level `java.util.concurrent` utilities over low-level `synchronized`/`wait`/`notify`. Use thread pools instead of creating threads manually. Always plan for graceful shutdown.