MODULE 5

Multithreaded Programming

Java OOP — Semester 3

Based on NDU Curriculum DCSA250059

PART A	Introduction to Multithreading
PART B	Synchronization

PART A: INTRODUCTION TO MULTITHREADING

1. What is a Thread?

A thread is the smallest unit of execution within a process. It is a lightweight subprocess that shares the memory space of its parent process.

Process vs Thread

Feature	Process	Thread
Definition	Program in execution	Single execution sequence within a process
Memory	Separate memory space	Shares memory space of process
Communication	Slower (inter-process communication)	Faster (direct memory access)
Creation Time	Longer	Shorter
Termination Time	Longer	Shorter
Context Switch	More overhead	Less overhead
Independence	Independent of other processes	Dependent on other threads
Resource Usage	More resources	Fewer resources

Why Multithreading?

- **Better resource utilization** - CPU doesn't stay idle
- **Improved performance** - Parallel execution
- **Enhanced responsiveness** - UI remains responsive
- **Better scalability** - Utilize multi-core processors
- **Simpler modeling** - Natural fit for many real-world problems

2. Thread Life Cycle (States)

A thread can be in one of the following states during its lifetime:

State	Description	How to Enter
NEW	Thread created but not started	Thread t = new Thread()
RUNNABLE	Ready to run or currently running	t.start()
RUNNING	Thread is executing	Selected by thread scheduler
BLOCKED	Waiting for monitor lock	Trying to access synchronized block
WAITING	Waiting indefinitely for another thread	wait(), join() without timeout
TIMED_WAITING	Waiting for specified time	sleep(ms), join(ms), wait(ms)

State	Description	How to Enter
TERMINATED	Thread execution completed	run() method finishes

Code Example — Thread States

```
public class ThreadStateDemo {
    public static void main(String[] args) throws InterruptedException {
        Thread t = new Thread(() -> {
            try { Thread.sleep(1000); }
            catch (InterruptedException e) { e.printStackTrace(); }
        });
        System.out.println("After creation: " + t.getState());   // NEW
        t.start();
        System.out.println("After start(): " + t.getState());    // RUNNABLE
        Thread.sleep(500);
        System.out.println("During sleep: " + t.getState());     // TIMED_WAITING
        t.join();
        System.out.println("After completion: " + t.getState()); // TERMINATED
    }
}
```

3. Creating Threads

Java provides three main ways to create threads:

Method 1: Extending Thread Class

```
class MyThread extends Thread {
    public MyThread(String name) { super(name); }

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

public class ExtendThreadDemo {
    public static void main(String[] args) {
        MyThread t1 = new MyThread("Thread-1");
        MyThread t2 = new MyThread("Thread-2");
        t1.start();
        t2.start();
    }
}
```

Method 2: Implementing Runnable Interface

```
class MyRunnable implements Runnable {
    private String name;
    public MyRunnable(String name) { this.name = name; }

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

```
        }
    }
}

public class RunnableDemo {
    public static void main(String[] args) {
        Thread t1 = new Thread(new MyRunnable("Runnable-1"));
        Thread t2 = new Thread(new MyRunnable("Runnable-2"));
        t1.start();
        t2.start();
    }
}
```

Method 3: Using Lambda Expressions (Java 8+)

```
public class LambdaThreadDemo {
    public static void main(String[] args) {
        Runnable task = () -> {
            for (int i = 1; i <= 5; i++) {
                System.out.println(Thread.currentThread().getName() + ": " + i);
                try { Thread.sleep(300); }
                catch (InterruptedException e) { e.printStackTrace(); }
            }
        };

        new Thread(task, "Lambda-1").start();
        new Thread(task, "Lambda-2").start();
        new Thread(() -> System.out.println("Anonymous thread")).start();
    }
}
```

Thread vs Runnable — Which to Use?

Thread Class	Runnable Interface
Cannot extend another class	Can extend another class
Simpler for basic usage	More flexible
Each thread has its own object	Multiple threads can share the same object
Method names directly available	Need Thread.currentThread()
Use when not extending any class	Use when extending another class or sharing resources

4. Important Thread Methods

Method	Description
start()	Starts the thread (calls run() internally)
run()	Contains the code executed by the thread
sleep(long ms)	Pauses thread for specified milliseconds
join()	Waits for thread to die
join(long ms)	Waits for thread to die or timeout
yield()	Pauses current thread to allow others
interrupt()	Interrupts a sleeping/waiting thread

Method	Description
isAlive()	Checks if thread is alive
getName()	Returns thread name
setName(String)	Sets thread name
getPriority()	Returns thread priority
setPriority(int)	Sets thread priority (1–10)
currentThread()	Returns currently executing thread object

sleep() Example — Countdown Timer

```
Thread timer = new Thread(() -> {
    for (int i = 10; i >= 0; i--) {
        System.out.println("Countdown: " + i);
        try { Thread.sleep(1000); }
        catch (InterruptedException e) {
            System.out.println("Timer interrupted!");
            return;
        }
    }
    System.out.println("TIME'S UP!");
});
timer.start();
Thread.sleep(5000);
timer.interrupt();  // Stop after 5 seconds
```

join() Example — Waiting for Workers

```
Thread t1 = new Thread(() -> {
    System.out.println("Worker 1: Starting...");
    try { Thread.sleep(2000); } catch (InterruptedException e) {}
    System.out.println("Worker 1: Completed");
});
t1.start();
t1.join();  // Main waits for t1 to finish
System.out.println("Main: Worker done. Proceeding...");
```

5. Thread Priority

Thread priority (1–10) is a hint to the scheduler about thread importance.

- Thread.MIN_PRIORITY = 1
- Thread.NORM_PRIORITY = 5 (default)
- Thread.MAX_PRIORITY = 10

```
Thread low  = new Thread(task, "Low-Priority");
Thread high = new Thread(task, "High-Priority");

low.setPriority(Thread.MIN_PRIORITY);   // 1
high.setPriority(Thread.MAX_PRIORITY);  // 10

low.start();
high.start();
```

⚠️ **Note:** Higher priority does not guarantee execution order. This is platform-dependent. Use priority

for performance tuning only, not program correctness.

6. Stopping and Blocking Threads

⚠️ **Note:** stop(), suspend(), and resume() are deprecated due to safety issues. Never use them.

Correct Way — Using a volatile Flag

```
class StoppableTask implements Runnable {
    private volatile boolean running = true;

    public void stop() { running = false; }

    @Override
    public void run() {
        while (running) {
            System.out.println("Working...");
            try { Thread.sleep(500); }
            catch (InterruptedException e) { break; }
        }
        System.out.println("Thread stopped gracefully");
    }
}
```

Correct Way — Using interrupt()

```
class InterruptibleTask implements Runnable {
    @Override
    public void run() {
        while (!Thread.currentThread().isInterrupted()) {
            System.out.println("Processing...");
            try { Thread.sleep(500); }
            catch (InterruptedException e) {
                Thread.currentThread().interrupt(); // Preserve status
                break;
            }
        }
    }
}

Thread t = new Thread(new InterruptibleTask());
t.start();
Thread.sleep(2000);
t.interrupt();  // Signal the thread to stop
```

PART B: SYNCHRONIZATION

7. The Need for Synchronization

When multiple threads access shared resources, race conditions can occur.

Race Condition Example (Without Synchronization)

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

// 1000 threads each incrementing 1000 times
// Expected: 1,000,000 | Actual: Usually less (e.g., 998,745)
```

The count++ operation is actually three steps:

- Read current value of count
- Increment the value
- Write the new value back

When multiple threads interleave, updates can be lost.

8. Synchronized Methods

Synchronized Instance Method

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

    public synchronized void increment() {
        count++;  // Now atomic!
    }

    public synchronized int getCount() {
        return count;
    }
}
// Result: always 1,000,000
```

Synchronized Static Method

```
class SharedResource {
    private static int sharedCounter = 0;

    // Locks on the Class object, not instance
    public static synchronized void incrementStatic() {
        sharedCounter++;
    }
}
```

9. Synchronized Blocks

For finer control, use synchronized blocks instead of synchronizing entire methods.

```
class BankAccount {
    private double balance;
    private Object lock = new Object();

    public void deposit(double amount) {
        synchronized(lock) {
            balance += amount;
        }
    }
```

```
    public void withdraw(double amount) {
        synchronized(this) {
            if (balance >= amount) balance -= amount;
        }
    }
}
```

Synchronized Method	Synchronized Block
Locks entire method	Locks only specific code section
Simpler syntax	More complex but flexible
Lock on 'this' or Class	Can lock any object
Less efficient for large methods	More efficient for critical sections only

10. The volatile Keyword

volatile ensures visibility of changes across threads, but does NOT guarantee atomicity.

```
class VolatileDemo {
    private static volatile boolean running = true;

    public static void main(String[] args) throws InterruptedException {
        Thread worker = new Thread(() -> {
            System.out.println("Worker started");
            while (running) { /* Do work */ }
            System.out.println("Worker stopped");
        });
        worker.start();
        Thread.sleep(2000);
        running = false;  // Worker sees this immediately due to volatile
    }
}
```

volatile	synchronized
Only ensures visibility	Ensures visibility AND atomicity
No locking overhead	Locking overhead
Cannot handle compound operations	Can handle compound operations
Lighter weight	Heavier

11. Inter-thread Communication

wait(), notify(), notifyAll()

These methods are defined in the Object class and must be called from a synchronized context.

Method	Description
wait()	Causes thread to wait until notified
wait(long timeout)	Waits with timeout

Method	Description
notify()	Wakes up one waiting thread
notifyAll()	Wakes up all waiting threads

Producer-Consumer Problem

```
class SharedBuffer {
    private int[] buffer;
    private int count = 0, in = 0, out = 0;

    public SharedBuffer(int size) { buffer = new int[size]; }

    public synchronized void produce(int item) throws InterruptedException {
        while (count == buffer.length) wait();  // Buffer full
        buffer[in] = item;
        in = (in + 1) % buffer.length;
        count++;
        notify();  // Notify consumer
    }

    public synchronized int consume() throws InterruptedException {
        while (count == 0) wait();  // Buffer empty
        int item = buffer[out];
        out = (out + 1) % buffer.length;
        count--;
        notify();  // Notify producer
        return item;
    }
}
```

wait()	sleep()
Method of Object class	Method of Thread class
Releases the lock	Does NOT release lock
Must be in synchronized context	Can be called from anywhere
Requires notify() to wake up	Wakes up after specified time
Used for inter-thread communication	Used for pausing execution

12. Deadlock

A deadlock occurs when two or more threads are blocked forever, each waiting for a resource held by the other.

Deadlock Example

```
Thread t1 = new Thread(() -> {
    synchronized (RESOURCE1) {
        Thread.sleep(100);
        synchronized (RESOURCE2) { /* ... */ }  // Waiting for R2
    }
});

Thread t2 = new Thread(() -> {
    synchronized (RESOURCE2) {
        Thread.sleep(100);
        synchronized (RESOURCE1) { /* ... */ }  // Waiting for R1 — DEADLOCK!
    }
});
```

Preventing Deadlock

1. Lock Ordering — always acquire locks in the same order:

```
// Both threads acquire locks in the same order: R1 then R2
synchronized (RESOURCE1) {
    synchronized (RESOURCE2) { /* safe */ }
}
```

- Use tryLock() with timeout (java.util.concurrent.locks.ReentrantLock)
- Minimize nested synchronization
- Avoid holding multiple locks simultaneously when possible

13. Advanced Threading Concepts

ThreadLocal

Provides thread-local variables — each thread has its own independent copy.

```
private static ThreadLocal<Integer> threadLocal = ThreadLocal.withInitial(() -> 0);
private static ThreadLocal<String> userContext = new ThreadLocal<>();

// Each thread sets and reads its own value independently
userContext.set("Alice");      // Thread 1
userContext.set("Bob");        // Thread 2
// No interference between threads
userContext.remove();          // Clean up when done
```

Daemon Threads

Daemon threads run in the background and do not prevent the JVM from exiting.

```
Thread daemon = new Thread(() -> {
    while (true) {
        System.out.println("Background monitoring...");
        try { Thread.sleep(1000); } catch (InterruptedException e) { break; }
    }
});

daemon.setDaemon(true);  // Must be set BEFORE start()
daemon.start();
// JVM exits when all non-daemon threads finish
```

14. Complete Banking System Example

Demonstrates ReentrantLock for advanced synchronization in a multi-threaded scenario.

```
import java.util.concurrent.locks.ReentrantLock;

class Bank {
    private double[] accounts;
    private ReentrantLock lock = new ReentrantLock();

    public void transfer(int from, int to, double amount) throws InterruptedException {
```

```
        lock.lock();
        try {
            if (accounts[from] < amount) return;
            accounts[from] -= amount;
            Thread.sleep(10);  // Simulate processing
            accounts[to]   += amount;
            System.out.printf("Transferred %.2f | Total: %.2f%n",
                amount, getTotalBalance());
        } finally {
            lock.unlock();  // ALWAYS unlock in finally block
        }
    }

    public double getTotalBalance() {
        lock.lock();
        try {
            double sum = 0;
            for (double a : accounts) sum += a;
            return sum;
        } finally { lock.unlock(); }
    }
}
```

15. Quick Reference

Synchronization Options

Option	Use Case
synchronized method	Entire method needs synchronization
synchronized block	Only part of method needs synchronization
volatile	Simple flag/status variables
ReentrantLock	Advanced features (tryLock, fairness)

Common Pitfalls

- **Race Condition** - Multiple threads modifying shared data without synchronization
- **Deadlock** - Circular waiting for locks
- **Starvation** - Thread never gets CPU time
- **Livelock** - Threads keep changing state but make no progress

Best Practices

- Always use synchronized or volatile for shared data
- Keep synchronized blocks as small as possible
- Acquire locks in consistent order to prevent deadlock
- Prefer Runnable over Thread for flexibility
- Use ExecutorService for managing thread pools
- Handle InterruptedException properly — never silently swallow it
- Never ignore an interrupted exception

Important Exam Questions

- What is the difference between process and thread?
- How do you create a thread in Java?
- What are the different states in the thread life cycle?
- What is synchronization and why is it needed?
- What is the difference between wait() and sleep()?
- What is a deadlock? How do you prevent it?
- What is the difference between synchronized method and synchronized block?
- What is the volatile keyword?
- What is the difference between notify() and notifyAll()?
- What are daemon threads?

— End of Module 5: Multithreaded Programming —