Session: 24-09-2025

Multi-Thread Programming in Core Java

1. Introduction

- **Definition:**
  Multithreading is a process of executing multiple threads simultaneously within a single program.
  A **thread** is the smallest unit of execution.
- **Why Multithreading?**
  - To perform multiple tasks at the same time.
  - Better utilization of CPU.
  - Improves performance of applications.
  - Useful in games, animations, web servers, real-time systems.

---

2. Life Cycle of a Thread

1. **New** – Thread object created using new.
2. **Runnable** – After calling start(), thread is ready to run.
3. **Running** – Thread scheduler picks the thread to execute.
4. **Waiting/Blocked** – Thread is paused temporarily.
5. **Terminated** – Thread finishes execution.

3. Creating Threads in Java

Two common ways:

Method 1: Extending Thread class

```
class MyThread extends Thread {
   public void run() {
      // task of the thread
      for(int i=1; i<=5; i++) {
         System.out.println("Thread is running: " + i);
         try {
            Thread.sleep(1000); // pause for 1 second
         } catch(Exception e) {
            System.out.println(e);
         }
      }
   }
}

public class ThreadExample1 {
   public static void main(String[] args) {
      MyThread t1 = new MyThread(); // create thread
      t1.start();  // start thread
   }
}
```

Here’s the **short line-by-line execution**:

1. class MyThread extends Thread → Create a custom thread class.
2. public void run() → Define the task the thread will perform.
3. for(...) → Loop runs 5 times, printing a message.
4. Thread.sleep(1000) → Pauses thread for 1 second each time.
5. MyThread t1 = new MyThread(); → Create a thread object (NEW state).
6. t1.start(); → Starts a new thread → JVM calls run() in parallel.
7. Output → "Thread is running: 1" to "Thread is running: 5" (with 1 sec delay).
8. After loop ends, thread terminates.

Sample output:

Thread is running: 1

Thread is running: 2

Thread is running: 3

Thread is running: 4

Thread is running: 5

Method 2: Implementing Runnable interface

```
class MyRunnable implements Runnable {
    public void run() {
        for(int i=1; i<=5; i++) {
            System.out.println("Runnable thread: " + i);
            try {
                Thread.sleep(500);
            } catch(Exception e) {
                System.out.println(e);
            }
        }
    }
}

public class ThreadExample2 {
    public static void main(String[] args) {
        MyRunnable obj = new MyRunnable();
        Thread t1 = new Thread(obj);  // create thread object
        t1.start();  // start thread
    }
}
```

Code Execution

1. class MyRunnable implements Runnable
   → Create a class that implements the Runnable interface.
2. public void run()
   → Override run() to define the task for the thread.
3. for(int i=1; i<=5; i++)
   → Loop prints "Runnable thread: i" five times.
4. Thread.sleep(500)
   → Pauses thread for 0.5 seconds in each iteration.
5. MyRunnable obj = new MyRunnable();
   → Create a Runnable object.
6. Thread t1 = new Thread(obj);
   → Create a Thread object and pass obj to it → tells JVM that this thread will execute obj.run().
7. t1.start();
   → Starts a new thread, JVM calls obj.run() in parallel.
8. Output →

Runnable thread: 1

Runnable thread: 2

Runnable thread: 3

Runnable thread: 4

Runnable thread: 5

(with 0.5 sec gap).

}

4. Example: Multiple Threads Running Together

```
class Task1 extends Thread {
    public void run() {
        for(int i=1; i<=5; i++) {
            System.out.println("Task 1 - Count: " + i);
        }
    }
}

class Task2 extends Thread {
    public void run() {
        for(int i=1; i<=5; i++) {
            System.out.println("Task 2 - Count: " + i);
        }
    }
}

public class MultiThreadDemo {
    public static void main(String[] args) {
        Task1 t1 = new Task1();
        Task2 t2 = new Task2();

        t1.start(); // executes Task1
        t2.start(); // executes Task2
    }
}
```

line-by-line execution for your multiple thread program 👇

Code Execution

1. class Task1 extends Thread → Defines a thread class Task1.
2. public void run() → Task for Task1: print "Task 1 - Count: i" five times.
3. class Task2 extends Thread → Defines another thread class Task2.
4. public void run() → Task for Task2: print "Task 2 - Count: i" five times.
5. public class MultiThreadDemo { public static void main... → Entry point of program.
6. Task1 t1 = new Task1(); → Create thread object t1 (NEW state).
7. Task2 t2 = new Task2(); → Create thread object t2 (NEW state).
8. t1.start(); → Starts a new thread → JVM calls t1.run().
9. t2.start(); → Starts another thread → JVM calls t2.run().
10. Both threads now run **concurrently**.
    - Scheduler decides execution order → outputs of Task1 and Task2 **interleave**.

Possible Output (varies each run)

Task 1 - Count: 1

Task 2 - Count: 1

Task 1 - Count: 2

Task 2 - Count: 2

Task 1 - Count: 3

Task 2 - Count: 3

Task 1 - Count: 4

Task 2 - Count: 4

Task 1 - Count: 5

Task 2 - Count: 5

👉 Sometimes Task1 may finish first, sometimes Task2, depending on **thread scheduling** by JVM.

5. Important Thread Methods

- start() → starts a thread.
- run() → code executed by the thread.
- sleep(ms) → pauses thread for given milliseconds.
- join() → waits for one thread to finish before continuing.
- isAlive() → checks if thread is still running.
- setPriority() → sets thread priority (1–10).

---

6. Use Cases of Multithreading

- **Web servers** – handle multiple requests at same time.
- **Gaming** – animations, background music, controls.
- **Online downloads** – downloading and playing simultaneously.
- **Data processing** – parallel execution for faster results.

✅ Summary:

Multithreading in Java allows concurrent execution of two or more threads, making programs faster and more efficient. It can be achieved by extending Thread or implementing Runnable.

Thread Priority in Java

- Each thread in Java has a **priority** (an integer from **1 to 10**).
- Default priority = **5** (NORM_PRIORITY).
- Higher priority thread is **more likely** to be scheduled by JVM, but **not guaranteed** (depends on OS & JVM).

Constants in Thread class

- Thread.MIN_PRIORITY → **1** (lowest)
- Thread.NORM_PRIORITY → **5** (default)
- Thread.MAX_PRIORITY → **10** (highest)

Setting Priority

t1.setPriority(Thread.MAX_PRIORITY); // 10

t2.setPriority(Thread.MIN_PRIORITY); // 1

👉 **Key Point:** Priority only gives a **hint** to the scheduler. It doesn't ensure strict order of execution.

Thread Synchronization

- When multiple threads access **shared resources** (like variables, files, or databases) at the same time, it can cause **data inconsistency**.
- **Synchronization** ensures that **only one thread** can access the shared resource at a time.

How it is done in Java

1. **synchronized keyword**
    - Used with methods or blocks.
    - Example:
    - synchronized void display() {
    -     // only one thread can execute here at a time
    - }
2. **Other tools:** Locks, Semaphores, Atomic variables (from java.util.concurrent).

Thread communication (wait / notify / notifyAll)

- wait(), notify(), notifyAll() are methods on Object used for thread coordination.
- They **must** be called inside a synchronized block/method.
- wait() releases the monitor and suspends the thread until notified.
- notify() wakes one waiting thread; notifyAll() wakes all waiting threads.
- Always check conditions with while (to handle spurious wakeups).
-

Stepwise working of Producer – Consumer

1. **If buffer is empty → consumer waits.**
2. **Producer produces an item → sets buffer as full → calls notifyAll() to wake consumer.**
3. **Consumer consumes the item → sets buffer as empty → calls notifyAll() to wake producer.**
4. **This cycle continues until all items are produced and consumed.**

Producer–Consumer example (single shared slot)

```
class SharedResource {
  private int data;
  private boolean available = false;

  public synchronized void produce(int value) {
    while (available) {
      try { wait(); } catch (InterruptedException e) { Thread.currentThread().interrupt(); }
    }
    data = value;
    available = true;
    System.out.println("Produced: " + data);
    notifyAll(); // wake waiting consumers (safer than notify in multi-thread cases)
```

```java
    }

    public synchronized void consume() {
        while (!available) {
            try { wait(); } catch (InterruptedException e) { Thread.currentThread().interrupt(); }
        }
        System.out.println("Consumed: " + data);
        available = false;
        notifyAll(); // wake waiting producers
    }
}

class Producer extends Thread {
    SharedResource r;
    Producer(SharedResource r) { this.r = r; }
    public void run() {
        for (int i = 1; i <= 5; i++) r.produce(i);
    }
}

class Consumer extends Thread {
    SharedResource r;
    Consumer(SharedResource r) { this.r = r; }
    public void run() {
        for (int i = 1; i <= 5; i++) r.consume();
    }
}
```

```
public class ThreadCommunicationExample {
    public static void main(String[] args) {
        SharedResource r = new SharedResource();
        new Producer(r).start();
        new Consumer(r).start();
    }
}
```

Possible output (order may vary):

Produced: 1

Consumed: 1

Produced: 2

Consumed: 2

Produced: 3

Consumed: 3

Produced: 4

Consumed: 4

Produced: 5

Consumed: 5

Step-by-step execution (short)

1. main() creates SharedResource r, starts Producer and Consumer threads — scheduling is non-deterministic.
2. **If Producer runs first:** Producer enters produce() (acquires r's monitor). available is false, so it sets data, sets available = true, prints Produced: 1.
3. Producer calls notifyAll() (wakes any waiting thread(s) but does **not** release the lock immediately), then exits produce() and **releases** the monitor.

4. Consumer acquires r's monitor, checks while (!available) → now available == true, so it proceeds: prints Consumed: 1, sets available = false, calls notifyAll(), and exits releasing the monitor.
5. **If Consumer ran first:** Consumer enters consume(), sees !available, calls wait() → this **releases** the monitor and the consumer blocks. Producer later runs, produces a value and notifyAll(). The waiting consumer is awakened but must reacquire the monitor before continuing; once it reacquires the monitor it re-checks the while condition, then consumes.
6. Repeat until loop finishes. Using while ensures correctness on spurious wakeups; notifyAll() avoids missed signals in multiple-producer/consumer scenarios.

Teaching tips (short)

- Emphasize: wait() releases lock; notify()/notifyAll() do not release lock — lock is released only when synchronized block/method exits.
- Prefer notifyAll() if multiple threads may be waiting.
- For production code, consider BlockingQueue from java.util.concurrent (it handles waiting/notification for you).

Deadlock

- Deadlock: two or more threads are blocked forever, each waiting for a lock held by another.
- Required conditions: Mutual exclusion, Hold-and-wait, No preemption, Circular wait.

Deadlock example

```
class Resource1 {}
class Resource2 {}

class Thread1 extends Thread {
    private final Resource1 r1;
    private final Resource2 r2;
    Thread1(Resource1 r1, Resource2 r2) { this.r1 = r1; this.r2 = r2; }

    public void run() {
        synchronized (r1) {
            System.out.println("Thread1 locked Resource1");
            try { Thread.sleep(100); } catch (InterruptedException e) {
Thread.currentThread().interrupt(); }
            synchronized (r2) {
                System.out.println("Thread1 locked Resource2");
            }
        }
    }
}

class Thread2 extends Thread {
    private final Resource1 r1;
    private final Resource2 r2;
```

```
    Thread2(Resource1 r1, Resource2 r2) { this.r1 = r1; this.r2 = r2; }

    public void run() {
        synchronized (r2) {
            System.out.println("Thread2 locked Resource2");
            try { Thread.sleep(100); } catch (InterruptedException e) {
Thread.currentThread().interrupt(); }
            synchronized (r1) {
                System.out.println("Thread2 locked Resource1");
            }
        }
    }
}

public class DeadlockExample {
    public static void main(String[] args) {
        Resource1 r1 = new Resource1();
        Resource2 r2 = new Resource2();
        new Thread1(r1, r2).start();
        new Thread2(r1, r2).start();
    }
}
```

Likely output (then program hangs):

Thread1 locked Resource1

Thread2 locked Resource2

After those two lines the program typically **hangs** (deadlock).

Step-by-step execution (short)

1. main() creates two resources r1 and r2, starts Thread1 and Thread2.
2. Suppose Thread1 runs first: it enters synchronized(r1) and prints "Thread1 locked Resource1". It then sleeps for 100ms *while still holding r1*.
3. Scheduler runs Thread2: it enters synchronized(r2) and prints "Thread2 locked Resource2". It then sleeps for 100ms *while still holding r2*.
4. After sleeping, Thread1 tries synchronized(r2) but cannot acquire r2 (held by Thread2) — so Thread1 blocks waiting for r2.
5. Thread2 then tries synchronized(r1) but cannot acquire r1 (held by Thread1) — so Thread2 blocks waiting for r1.
6. Now Thread1 is waiting for r2 and Thread2 is waiting for r1 → **circular wait**: neither can proceed → deadlock.

Quick ways to avoid deadlock (short)

- **Consistent lock ordering:** always acquire locks in the same global order (e.g., always lock r1 then r2).
- **Try-lock with timeout:** use Lock.tryLock(timeout, TimeUnit) so a thread can back off and retry.
- **Reduce lock scope:** keep synchronized sections as short as possible.
- **Use higher-level concurrency utilities:** e.g., java.util.concurrent classes that avoid explicit multiple locks.
- **Detect & recover:** in complex systems, detect deadlock (thread dump / jstack) and restart or roll back tasks.