Session: 23-09-2025

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).