**Session 1: Memory Management & Garbage Collection (1.5 hrs)**

**Training Goals**

* Understand how Java manages memory internally (Heap, Stack, PermGen/Metaspace).
* Learn GC algorithms and tuning basics.
* Monitor memory in real-time with Runtime and simple profiling tools.

**Theory Points**

* JVM Memory Areas: Stack, Heap, Method Area, Native Memory.
* Object lifecycle and reference types (strong, weak, soft, phantom).
* Common Garbage Collectors (Serial, Parallel, G1, ZGC).
* How and when GC runs.
* Memory leaks in Java (static references, unclosed resources).

**Code Demo 1: Monitoring Memory**

public class MemoryMonitorDemo {

public static void main(String[] args) {

Runtime runtime = Runtime.getRuntime();

System.out.println("Total Memory: " + runtime.totalMemory());

System.out.println("Free Memory: " + runtime.freeMemory());

// Creating objects

for (int i = 0; i < 100000; i++) {

String data = new String("Data-" + i);

}

System.gc(); // Suggest GC

System.out.println("After GC - Free Memory: " + runtime.freeMemory());

}

}

**Exercise:**

* Use VisualVM / IntelliJ Profiler to observe GC events.
* Simulate a memory leak with a List storing objects infinitely.

**🔹 Session 2: Comparable & Comparator (1 hr)**

**Training Goals**

* Understand sorting custom objects.
* Learn difference between natural ordering and custom ordering.

**Theory Points**

* Comparable<T> → defines natural ordering via compareTo().
* Comparator<T> → external sorting logic; multiple sorting rules possible.
* Lambda expressions for Comparators.

**Code Demo 2: Comparable Example**

class Student implements Comparable<Student> {

int id;

String name;

Student(int id, String name) {

this.id = id; this.name = name;

}

public int compareTo(Student s) {

return this.name.compareTo(s.name);

}

public String toString() {

return id + " - " + name;

}

public static void main(String[] args) {

List<Student> list = new ArrayList<>();

list.add(new Student(3, "Ravi"));

list.add(new Student(1, "Amit"));

list.add(new Student(2, "Neha"));

Collections.sort(list);

System.out.println(list);

}

}

**Code Demo 3: Comparator Example**

Collections.sort(list, (s1, s2) -> s1.id - s2.id);

**Exercise:**

* Sort employees by name, age, and salary using different comparators.

**🔹 Session 3: File Handling (1.5 hrs)**

**Training Goals**

* Read/write files in Java.
* Handle exceptions safely using try-with-resources.
* Understand difference between character and byte streams.

**Theory Points**

* Classes: File, FileReader, FileWriter, BufferedReader, BufferedWriter.
* Using NIO.2 (Files, Paths, StandardOpenOption).
* Common pitfalls: encoding issues, closing streams.

**Code Demo 4: Read & Write Example**

import java.io.\*;

public class FileDemo {

public static void main(String[] args) throws IOException {

String filename = "output.txt";

try (BufferedWriter bw = new BufferedWriter(new FileWriter(filename))) {

bw.write("Hello, Java File Handling!");

}

try (BufferedReader br = new BufferedReader(new FileReader(filename))) {

String line;

while ((line = br.readLine()) != null)

System.out.println(line);

}

}

}

**Exercise:**

* Create a file containing employee data.
* Read it, sort by name, and write back to another file.

**🔹 Session 4: Multithreading & Generics (2 hrs)**

**Training Goals**

* Learn basics of thread creation, synchronization, and thread safety.
* Practice using Collections with threads.
* Create generic classes and methods for reusability.

**Theory Points**

* Threads vs Runnable.
* Synchronization blocks and locks.
* ExecutorService for thread pools.
* Generics: Type safety, wildcards, bounded types.
* Collections with Generics.

**Code Demo 5: Simple Thread Example**

class MyThread extends Thread {

public void run() {

for (int i = 0; i < 5; i++)

System.out.println(Thread.currentThread().getName() + " - " + i);

}

public static void main(String[] args) {

MyThread t1 = new MyThread();

MyThread t2 = new MyThread();

t1.start();

t2.start();

}

}

**Code Demo 6: Using Generics**

class Box<T> {

private T value;

public void set(T value) { this.value = value; }

public T get() { return value; }

public static void main(String[] args) {

Box<String> strBox = new Box<>();

strBox.set("Java Rocks!");

System.out.println(strBox.get());

}

}

**Exercise:**

* Create a generic Repository<T> class for CRUD operations.
* Create threads that write to a shared list and monitor performance using Runtime.

**🔹 Session 5: Collections Practical (1 hr)**

**Hands-on Practice**

* Create and manipulate List, Set, Map, and Queue.

**Code Demo 7: Collections Example**

import java.util.\*;

public class CollectionDemo {

public static void main(String[] args) {

List<String> list = new ArrayList<>(List.of("Java", "Python", "C++"));

Set<String> set = new HashSet<>(list);

Map<Integer, String> map = new HashMap<>();

Queue<String> queue = new LinkedList<>(list);

map.put(1, "A");

map.put(2, "B");

list.add("Go");

set.remove("C++");

queue.offer("Rust");

System.out.println("List: " + list);

System.out.println("Set: " + set);

System.out.println("Map: " + map);

System.out.println("Queue: " + queue);

}

}

**Exercise:**

* Iterate through all four data structures using Iterator, forEach, and lambda expressions.
* Compare performance using System.nanoTime().

## **Module: Multithreading & Concurrency in Java**

### 🔹 ****1. What is a Thread?****

**Concept Simplified:**

* A **thread** is the smallest unit of execution within a process.
* Every Java program runs on at least one thread — the **main thread**.
* Threads allow multiple parts of a program to run **concurrently** (not necessarily simultaneously, depending on CPU cores).

**Why Threads?**

* Improve performance on multi-core processors.
* Handle independent tasks (like I/O, computation, UI updates) in parallel.
* Example: Downloading a file and updating progress bar at the same time.

**Code Example: Creating a Thread**

// Approach 1: Extending Thread class

class MyThread extends Thread {

public void run() {

System.out.println("Thread running: " + Thread.currentThread().getName());

}

public static void main(String[] args) {

MyThread t1 = new MyThread();

t1.start(); // start() triggers JVM to call run() on a separate thread

}

}

// Approach 2: Implementing Runnable interface

class MyRunnable implements Runnable {

public void run() {

System.out.println("Thread running: " + Thread.currentThread().getName());

}

public static void main(String[] args) {

Thread t = new Thread(new MyRunnable());

t.start();

}

}

**Key Methods:**

* start() → launches new thread.
* run() → thread logic.
* sleep(ms), join(), yield() → control thread behavior.

### 🔹 ****2. Thread Life Cycle****

**Stages:**

1. **New** → Thread created but not started.
2. **Runnable** → Ready to run, waiting for CPU.
3. **Running** → CPU executing the thread’s run() method.
4. **Blocked / Waiting** → Waiting for resource or another thread’s signal.
5. **Terminated** → Finished execution.

**Visual Flow (describe this in class):**

NEW → RUNNABLE → RUNNING → WAITING/BLOCKED → TERMINATED

**Code Demo:**

class ThreadLifeCycle extends Thread {

public void run() {

try {

System.out.println(getName() + " is Running...");

Thread.sleep(1000);

System.out.println(getName() + " is Done.");

} catch (InterruptedException e) {

e.printStackTrace();

}

}

public static void main(String[] args) {

ThreadLifeCycle t1 = new ThreadLifeCycle();

System.out.println("State before start: " + t1.getState());

t1.start();

System.out.println("State after start: " + t1.getState());

}

}

### 🔹 ****3. Executor Framework****

**Why Executors?**

* Managing threads manually (create/start/join) is messy.
* The **Executor Framework** provides a high-level API for thread pooling and task scheduling.

**Core Interfaces:**

* Executor → basic interface.
* ExecutorService → advanced version with shutdown, submit, and future support.
* ScheduledExecutorService → run tasks at fixed delay or interval.

**Thread Pools:**

* Pre-created threads reused for multiple tasks — saves time and memory.

**Code Example: Using Executors**

import java.util.concurrent.\*;

public class ExecutorDemo {

public static void main(String[] args) {

ExecutorService executor = Executors.newFixedThreadPool(3); // 3 threads

for (int i = 1; i <= 5; i++) {

int taskId = i;

executor.submit(() -> {

System.out.println("Task " + taskId + " running on " + Thread.currentThread().getName());

});

}

executor.shutdown(); // graceful shutdown

}

}

**Key Executor Types:**

| **Method** | **Description** |
| --- | --- |
| newSingleThreadExecutor() | One worker thread |
| newFixedThreadPool(n) | Pool of n threads |
| newCachedThreadPool() | Reuses threads dynamically |
| newScheduledThreadPool(n) | Executes tasks at fixed intervals |

### 🔹 ****4. Synchronization****

**The Problem:**

* When multiple threads access shared data, results can become inconsistent.  
  Example: two threads updating the same counter at the same time → race condition.

**Solution: Synchronization**

* Allows only one thread to access a shared block at a time.

**Mechanisms:**

1. synchronized keyword (method or block level).
2. Lock interface (more control).
3. volatile keyword (visibility guarantee).

**Code Example: Synchronized Block**

class Counter {

private int count = 0;

public synchronized void increment() {

count++;

}

public int getCount() {

return count;

}

}

public class SyncDemo {

public static void main(String[] args) throws InterruptedException {

Counter c = new Counter();

Thread t1 = new Thread(() -> {

for (int i = 0; i < 1000; i++) c.increment();

});

Thread t2 = new Thread(() -> {

for (int i = 0; i < 1000; i++) c.increment();

});

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

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

System.out.println("Final count: " + c.getCount());

}

}

**Without synchronization**, final count might be less than 2000.  
**With synchronization**, it will always be 2000.

### 🔹 ****5. Bonus: Deadlocks & Best Practices****

**Deadlock:**  
Two or more threads waiting for each other’s locks — all stuck forever.

**Avoid it by:**

* Always acquiring locks in a consistent order.
* Minimizing synchronized blocks.
* Using concurrent collections (ConcurrentHashMap, CopyOnWriteArrayList).

### 🧩 ****Session Exercises****

1. Create a program with 3 threads printing numbers 1–10 alternately.
2. Use ExecutorService to submit 10 tasks and print the executing thread name.
3. Write a synchronized bank account example for deposit/withdraw.
4. Use Thread.getState() to observe lifecycle transitions.