**JAVA Memory Management**

**Concurrency in JAVA**

**1. Memory Leak Detector and Fixer (Garbage Collection, JVM Memory Model)**

**Problem Statement:**  
Develop a **Memory Leak Detector** that:

* Simulates a **memory leak scenario** using an unclosed resource (e.g., BufferedReader, InputStream).
* Monitors **heap memory usage** using **Runtime.getRuntime().totalMemory()** and **freeMemory()**.
* Identifies objects that are not garbage-collected due to **strong references** (e.g., via ThreadLocal or static collections).
* Implements a **fix** by:
  + Using WeakReference or SoftReference for cache-like data.
  + Explicitly closing resources in a finally block or using **try-with-resources**.

**2. Multi-threaded Order Processing System (Threads, Synchronization)**

**Problem Statement:**  
Design a **multi-threaded order processing system** where:

* Multiple threads simulate **customers placing orders**.
* An OrderProcessor thread processes **pending orders** in real time.
* Ensure **thread safety** using:
  + **Synchronized methods** to prevent race conditions on shared resources.
  + **BlockingQueue** to handle concurrent order processing.
* Implement a **graceful shutdown** mechanism to stop order processing when the system is terminated.

**3. Flight Ticket Booking System (Concurrency Utilities, ExecutorService, Future)**

**Problem Statement:**  
Develop a **flight ticket booking system** where:

* Users **search for flights and book tickets** concurrently.
* Use **ExecutorService** to handle multiple user requests efficiently.
* Implement **Callable and Future** to:
  + Simulate fetching flight details asynchronously.
  + Return a Future<List<Flight>> containing available flights.
* Implement **synchronization** to prevent **overbooking** of seats for the same flight.

**4. Bank Transaction System (Threads, Synchronization, Deadlock Prevention)**

**Problem Statement:**  
Design a **multi-threaded banking transaction system** where:

* Multiple threads simulate **money transfers** between bank accounts.
* Implement **synchronization** to prevent **race conditions** (e.g., incorrect balances).
* Use **deadlock prevention** techniques:
  + Implement **lock ordering** to prevent circular waits.
  + Use **ReentrantLock** instead of synchronized where necessary.
* Provide a **real-time balance update** mechanism using a shared data structure (ConcurrentHashMap).

**5. Countdown Timer for Parallel Task Execution (CountDownLatch, ExecutorService)**

**Problem Statement:**  
Develop a **multi-threaded countdown timer system** that:

* Uses **CountDownLatch** to **wait** for multiple background tasks to complete before proceeding.
* Spawns **multiple threads** to execute parallel tasks such as:
  + **Downloading files** from multiple sources.
  + **Processing large datasets** in chunks.
* Once all tasks finish, the **main thread resumes execution** and merges results.