Multi-threading:

1. Implement a multithreaded program to perform matrix multiplication.
   - Approach: Use Java threads and synchronize access to shared resources.
2. Write a Java program to demonstrate deadlock and livelock scenarios.
   - Approach: Use synchronized blocks and methods to create deadlock and livelock situations.
3. Create a thread-safe singleton class.
   - Approach: Use synchronized methods and double-checked locking.
4. Implement a producer-consumer problem using threads and semaphores.
   - Approach: Use Java threads, semaphores, and a shared queue.
5. Write a Java program to demonstrate the dining philosophers problem.
   - Approach: Use threads, semaphores, and a shared resource (chopsticks).

Threading:

1. Implement a thread pool to manage multiple threads.
   - Approach: Use Java's Executor Framework and ThreadPoolExecutor.
2. Write a Java program to demonstrate thread interruption and cancellation.
   - Approach: Use Thread.interrupt() and Thread.isInterrupted().
3. Create a thread-safe queue implementation.
   - Approach: Use synchronized methods and a shared queue.
4. Implement a thread-safe dictionary (HashMap).
   - Approach: Use ConcurrentHashMap or synchronized methods.
5. Write a Java program to demonstrate thread-local variables.
   - Approach: Use ThreadLocal and ThreadLocalVariable.

Synchronous Operations:

1. Implement a synchronous queue (BlockingQueue).
   - Approach: Use Java's BlockingQueue interface and implementations like ArrayBlockingQueue.
2. Write a Java program to demonstrate the use of CountDownLatch.
   - Approach: Use CountDownLatch and await() method.
3. Create a synchronous map (ConcurrentHashMap).
   - Approach: Use ConcurrentHashMap and synchronized methods.
4. Implement a synchronous list (CopyOnWriteArrayList).
   - Approach: Use CopyOnWriteArrayList and synchronized methods.
5. Write a Java program to demonstrate the use of CyclicBarrier.
   - Approach: Use CyclicBarrier and await() method.

Garbage Collection:

Problem Statement 1: Understanding Basic Garbage Collection

Objective: Observe the behavior of garbage collection with a focus on object creation and finalization.

Tasks:

1. Create a Class:
   - Define a class `GCExample` with a constructor that prints a message and a `finalize()` method that prints when an object is garbage collected.
2. Instantiate and Nullify:
   - In the `main` method, create multiple instances of `GCExample`, set their references to `null`, and call `System.gc()`.
3. Observe Output:
   - Check the console output to verify when the `finalize()` method is invoked.

Approach:

1. Define a class `GCExample` with a constructor and a `finalize()` method.
2. In the `main` method, create instances of `GCExample` and then nullify references.
3. Explicitly request garbage collection using `System.gc()` and observe when `finalize()` is called.

Problem Statement 2: Memory Usage with Collections

Objective: Analyze how memory usage changes when using collections and how garbage collection affects it.

Tasks:

1. **Create a Class with Large Data:**
   - Define a class `LargeData` with a large array.
2. **Use Collections:**
   - Create a list of `LargeData` objects and periodically clear the list.
3. **Monitor Memory:**
   - Print memory usage before and after clearing the list and invoking garbage collection.

Approach:

1. Define the `LargeData` class with a large data structure.
2. Create a `List` to hold instances of `LargeData` and periodically clear it.
3. Call `System.gc()` and print memory usage before and after clearing the list to observe the effect.

Problem Statement 3: Custom Finalizers and `try-with-resources`

Objective: Implement a custom finalizer and integrate it with `try-with-resources` to understand resource management.

Tasks:

1. **Create a Resource Class with Finalizer:**
   - Define a class `ResourceWithFinalizer` with a `finalize()` method that simulates resource cleanup.
2. **Use `try-with-resources`:**
   - Implement a method that uses `try-with-resources` to manage instances of `ResourceWithFinalizer`.
3. **Analyze Finalizer Behavior:**
   - Observe how finalizers are invoked and compare it with the use of `try-with-resources`.

Problem Statement 4: Producer-Consumer problem

Objective: Implement a Producer-Consumer problem using Java threads where a producer produces items and adds them to a shared buffer, while a consumer takes items from the buffer.

Components:

1. **Shared Buffer (Queue):**
   - A data structure to hold the produced items and be accessed by both producer and consumer threads.
2. **Producer:**
   - A thread that generates items and adds them to the buffer.
3. **Consumer:**
   - A thread that consumes items from the buffer.
4. **Synchronization:**

    - Ensure that the producer and consumer do not encounter issues like race conditions or deadlocks when accessing the shared buffer.

Approach

1. **Define a Shared Buffer:**
    - Use a blocking queue such as `ArrayBlockingQueue` to handle synchronization and avoid manual locking.
2. **Create Producer and Consumer Classes:**
    - Implement the `Runnable` interface for both producer and consumer to define their tasks.
3. **Manage Concurrency:**
    - Use `ArrayBlockingQueue` which internally manages synchronization.
4. **Run the Threads:**
    - Start producer and consumer threads and observe their interaction with the shared buffer.