1. Producer-Consumer (Classic Synchronization)

Goal: Implement the Producer–Consumer pattern using `synchronized`, `wait()`, and `notifyAll()` to safely coordinate multiple threads.

Technical Requirements:

1. MessageQueue

Create a thread-safe `MessageQueue` class that:

- Stores `String` messages using a standard collection (`List` or `Deque`)
- Exposes two synchronized methods:
    - `void put(String message)`
    - `String take()`

2. Producers (MessageSender)

- Implement `MessageSender` as a `Runnable`
- Each sender:
    - Produces **5 unique, time-stamped messages**
    - Adds them to the shared `MessageQueue`

3. Consumers (MessageReceiver)

- Implement `MessageReceiver` as a `Runnable`
- Each receiver:
    - Reads and prints **5 messages** from the shared `MessageQueue`

4. Thread Launch

- Start:
    - 3 MessageSender threads
    - 3 MessageReceiver threads

- All threads must share the *same* *`MessageQueue`* *instance*

Synchronization Rules

- Queue operations must be atomic
- If the queue is empty, a receiver must wait
- When a sender adds a message, it must signal waiting receivers
- Use:
    - `synchronized`
    - `wait()`
    - `notifyAll()`

Result Goal

- No message is lost
- No message is duplicated
- Each message is printed exactly once

---

2. Managed Concurrency and Advanced Locking (Executors & Explicit Locks)

Goal: Refactor the Producer-Consumer solution to use modern Executor Services and replace intrinsic locks with explicit Lock and Condition variables.

Technical Requirements:

1. **Refactor to Executors:** Reimplement the Producer-Consumer application from Question 1 using the Executor Framework.
    - Create a dedicated fixed-size thread pool (e.g., Executors.newFixedThreadPool(3)) for the MessageSender tasks.
    - Create a separate fixed-size thread pool for the MessageReceiver tasks.
2. **Graceful Shutdown:** Implement a clean shutdown sequence for both ExecutorService instances, ensuring the main application waits for all submitted tasks (both senders and receivers) to fully complete before exiting the program.
3. **Advanced Locking:** Modify the internal thread-safety mechanisms within your MessageQueue class. Replace all uses of the synchronized keyword and the wait()/notifyAll() methods with an explicit Lock mechanism (e.g., ReentrantLock from java.util.concurrent.locks) and its associated Condition variables to handle the waiting and signaling logic.

3. Asynchronous Computation and Parallel Processing (Futures & Streams)

Goal: Learn how to use `Callable`, `Future`, and `parallelStream()`.

A. Callable & Future

- Implement a `Callable<Integer>` that computes the sum from `1` to `N`
- Submit it to an `ExecutorService`
- Retrieve the result using `Future.get()`
- Observe blocking behavior
- Shutdown the executor

B. Parallel Streams

- Generate a list of 1,000,000 integers
- Compute the sum using:
    - Sequential stream
    - Parallel stream
- Print execution time for both
- Observe differences