Project 1: Creating Threads by Extending `Thread`

Goal
Start five threads where each prints numbers 1 to 10.

Steps

1. Create a class (e.g., `NumberPrinter`) extending `Thread`.
2. In its `run()` method, loop from 1 to 10 and print each number.
3. In `main`, create five instances of this thread class (each with a unique name).
4. Call `start()` on each thread and observe concurrent output.

Project 2: Implementing `Runnable` + Thread Priority

Goal
Create multiple threads using `Runnable`, set different thread priorities, and each thread prints numbers that are multiples of its own priority (e.g., from 1 to 50).

Steps

1. Create a class (e.g., `PriorityPrinter`) implementing `Runnable`.
2. In `run()`, loop through a range (e.g., 1–50). For each number, check if it's divisible by `Thread.currentThread().getPriority()`. If so, print it.
3. In `main`, create five `Thread` objects, each wrapping the same `Runnable`.
4. Assign distinct priorities (e.g., min, normal, max, etc.) to each thread.
5. Call `start()` on each and observe which multiples are printed.

Project 3: Synchronization (Shared Counter)

Goal

Learn to synchronize access to a shared resource (a counter) to avoid race conditions.

Steps

1. Create a `Counter` class with an integer field `count` and a synchronized `increment()` method.
2. Create a `Runnable` (e.g., `IncrementTask`) that repeatedly calls `increment()` in a loop.
3. In `main`, instantiate a single `Counter`.
4. Launch multiple threads using `IncrementTask` sharing the same `Counter`.
5. Wait (join) for all threads to finish. Check that `count` matches the total number of increments across all threads.

Project 4: Thread Pool (ExecutorService)

Goal

Use an `ExecutorService` to manage a group of threads, submitting multiple tasks for parallel execution.

Steps

1. Create a simple `Runnable` that performs a short task (e.g., prints a message).
2. In `main`, create an `ExecutorService` (e.g., a fixed or cached thread pool).
3. Submit multiple instances of the `Runnable` in a loop.
4. Call `shutdown()` when done to stop accepting new tasks.
5. Optionally use `awaitTermination()` to wait for all tasks to complete.

Project 5: Managing Multiple Shared Resources with ReentrantLock

Goal

Take the basic idea of using `ReentrantLock` and apply it to a slightly more realistic scenario with multiple shared resources (e.g., simulating transfers between bank accounts). You'll learn how to lock each resource safely and handle potential waits or conflicts.

Scenario Description (Example: Bank Accounts)

1. **Bank Accounts**
    - Suppose you have 2 or 3 different accounts, each with its own balance.
    - Each account also has a dedicated `ReentrantLock` to protect its balance.
2. **Threads (Transfers)**
    - Multiple threads are created, each representing a transfer operation.
    - A transfer involves:
        1. Locking the **source** account.
        2. Locking the **destination** account.
        3. Withdrawing from source, depositing into destination.
        4. Unlocking both accounts in the correct order (usually the same order you locked them).
3. **Avoid Deadlock**
    - If two threads each lock different accounts first, then attempt to lock the other's account, a deadlock can occur.
    - One approach is to **always** lock accounts in the same order (e.g., by account ID).
    - Alternatively, you can use methods like `tryLock()` with a timeout to detect and handle lock unavailability.
4. **Observations**
    - You'll see how `ReentrantLock` provides more control than `synchronized`, especially for scenarios needing multiple locks.
    - You can track successful vs. failed transfers if locks can't be acquired in time.

Suggested Steps

1. **Create an `Account` Class**

    - Fields: `balance` (e.g., `int` or `double`), a `ReentrantLock` object, and an `id` or name for clarity.
    - Constructor initializes the balance, sets `id`, and instantiates the lock.
2. **Define a `transfer()` Method**
    - Accepts two `Account` objects (source, destination) and an amount.
    - Locks both accounts in a consistent order (e.g., by comparing their IDs).
    - Withdraw from the source account, deposit into the destination.
    - Unlock both accounts in a `finally` block.
3. **Create a `TransferTask` (Runnable)**
    - Has references to source and destination `Account` objects, plus a transfer amount.
    - In `run()`, calls `transfer()` repeatedly or just once (your choice).
4. **In `main`** (or any driver class):
    - Instantiate 2–3 `Account` objects with different balances.
    - Create multiple `Thread` objects using `TransferTask`, each trying to move money between random pairs of accounts.
    - Start them and wait for them to finish (using `join()` or a thread pool's shutdown).
    - Print final balances to confirm correctness and see that no money was "lost" or "magically created."
5. **Optional Enhancement**
    - Use `tryLock(long time, TimeUnit unit)` to handle scenarios where a lock can't be acquired quickly.
    - If you fail to acquire a lock, skip or retry the transfer to avoid deadlocks.