1. What issues will you face with multithreading, and how can we protect against them?

Common Issues in Multithreading

1. **Race Conditions**  
   Multiple threads access shared data concurrently and at least one thread modifies it, leading to unpredictable behavior.  
   **Example**: Two threads updating the same bank account balance.
2. **Deadlocks**  
   Two or more threads wait forever for each other to release locks.  
   **Example**: Thread A holds Lock 1 and waits for Lock 2, while Thread B holds Lock 2 and waits for Lock 1.
3. **Starvation**  
   A thread waits indefinitely to access resources because other threads are constantly prioritized.
4. **Thread Interference**  
   Occurs when threads interleave operations on shared data, breaking logic.

How to Prevent These Issues

1. **Use Synchronization Mechanisms**

 **Avoid Nested Locks**  
Keep locking order consistent to avoid deadlocks.

1. **Prefer Concurrency Utilities**  
   Java's java.util.concurrent package is your best friend:

 **Immutable Objects**  
Design objects that don’t change state to eliminate shared data problems.

 **Thread Pools Over Manual Threads**  
Avoid creating new threads for every task—use Executors.newFixedThreadPool() instead for better performance and control.

1. Create a class implementing a thread-safe program.

Runnable task = () -> {

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

counter.increment();

}

};

4. Synchronized in method level vs. block level. What are the differences and advantages of each?

 **Use method-level** synchronization if your entire method logic is critical and must be executed by only one thread at a time.

* **Use block-level** if only part of the method needs protection, especially when performance and concurrency matter.

In Java, ThreadLocal is a class that provides **thread-confined variables**, meaning each thread accessing such a variable has its own, independently initialized copy. This is incredibly useful in multithreaded environments where you want to avoid shared state without using synchronization.

ThreadLocal ही Java मधील एक क्लास आहे जी प्रत्येक **थ्रेडसाठी स्वतंत्र डेटा साठवते**. म्हणजेच, जर दोन थ्रेड्स एकाच ThreadLocal व्हेरिएबलकडे पाहत असतील, तरी त्यांना **एकमेकांचा डेटा दिसत नाही**. प्रत्येक थ्रेडला त्याचा स्वतःचा वेगळा डेटा मिळतो

How thread pool works internationally

How It Works Internally (Java Perspective)

1. **Initialization**:
   * A thread pool is created using classes like Executors.newFixedThreadPool() or ThreadPoolExecutor.
   * It sets up:
   * A **core pool size** (minimum number of threads).
   * A **maximum pool size** (upper limit).
   * A **task queue** to hold waiting tasks.
2. **Submitting Tasks**:
   * When you call executor.submit(task):
   * If the number of running threads is **less than the core size**, a new thread is created to handle the task.
   * If core threads are busy, the task is **queued**.
   * If the queue is full and the number of threads is **less than the max**, a new thread is created.
   * If all threads are busy and the queue is full, the task is **rejected** (based on the rejection policy).

**Worker Threads**:

* Each thread in the pool picks tasks from the queue and runs them.
* After finishing a task, the thread checks the queue again.
* If idle for too long (beyond keepAliveTime), extra threads (beyond core size) may be terminated.

**Shutdown**:

* When shutdown() is called, the pool stops accepting new tasks.
* It waits for current tasks to finish before terminating threads.