Sure Gyanenjay! Let's go through each of your questions step by step in a simple and practical way, especially from an interview and real-world Java backend perspective:

**✅ 1) Can you explain synchronization in Java?**

**Synchronization** in Java ensures that **only one thread at a time** can access a **critical section** (shared resource like variable, object, or block of code).  
This prevents **race conditions** and maintains **data consistency** in a multithreaded environment.

**🔹 How it works:**

* Java provides synchronized **keyword** to make a method or block thread-safe.

public synchronized void increment() {

count++;

}

Or:

public void increment() {

synchronized(this) {

count++;

}

}

**🔹 Where it’s used:**

* Updating shared counters
* Reading/writing shared files or DB
* Singleton implementation (lazy loading)

**🔸 Problem:**

It may **block threads**, even if no actual conflict, causing performance issues.

**✅ 2) How can we handle deadlocks in Java?**

A **deadlock** happens when **two or more threads** are **waiting for each other to release locks**, and none can proceed.

**🔹 Example:**

Thread 1 locks A → waits for B

Thread 2 locks B → waits for A

**🔸 Prevention Techniques:**

1. **Lock ordering** – Always acquire locks in a fixed global order.
2. **TryLock with timeout** (from ReentrantLock) – Avoid waiting forever.

if (lock1.tryLock(1, TimeUnit.SECONDS)) {

if (lock2.tryLock(1, TimeUnit.SECONDS)) {

// do work

}

}

1. **Avoid nested locking** – Try not to lock inside another lock.
2. **Deadlock detection algorithms** – Useful in complex systems.

**✅ 3) Explain lock starvation in Java**

**Lock starvation** occurs when a thread **never gets CPU time or lock** access because other threads are **constantly preempting it**.

**🔹 Cause:**

* High-priority threads always getting scheduled.
* Thread waiting for a lock, but not getting chance (due to unfair lock access).

**🔹 Example:**

ReentrantLock lock = new ReentrantLock(true); // fair lock avoids starvation

**🔸 Prevention:**

* Use **fair locks** (like ReentrantLock(true))
* Avoid long-running tasks holding locks
* Design thread access policies fairly

**✅ 4) Alternatives of synchronized keyword in Java**

Java provides more advanced, flexible, and performant alternatives:

| **🔧 Feature** | **💡 Alternative** |
| --- | --- |
| Fine-grained locking | ReentrantLock |
| Wait-notify per condition | Condition |
| Atomic operations | AtomicInteger, AtomicReference (CAS-based) |
| Read-heavy data structures | ReadWriteLock |
| Concurrent collections | ConcurrentHashMap, CopyOnWriteArrayList |
| Semaphore | Semaphore for controlling access to a resource pool |
| Barriers/Latches | CountDownLatch, CyclicBarrier, Phaser |
| Asynchronous flow | CompletableFuture |

**🔹 Example of ReentrantLock:**

Lock lock = new ReentrantLock();

lock.lock();

try {

// critical section

} finally {

lock.unlock();

}

✅ Use ReentrantLock if you want:

* Timeout-based lock acquiring
* Fair lock (FIFO)
* Interruptible lock
* Multiple condition variables

Would you like code examples for any of these like starvation, deadlock with tryLock, or fair locking?

Also want a short interview version of all 4 answers for quick revision?