Concurrency in Java explained

give me the deep and detailed explaination of Concurrency in java so that i can able to understand it well and do not have any type of doubt in this.

so give me the deep and detailed explaination

Concurrency in Java — deep, detailed, and (I hope) crystal clear

I'll explain core concepts, Java primitives/APIs, the Java Memory Model (visibility & ordering), common problems (deadlock, livelock, starvation), high-level utilities, patterns/best practices, and show small, clear code examples you can run. I'll keep each idea simple and then show why it matters.

---

1) What is concurrency vs parallelism

- **Concurrency**: multiple tasks *in progress* at the same time (conceptual). They may interleave on one CPU or run simultaneously on many CPUs. Focus: structure & correctness when tasks interact.
- **Parallelism**: actually running at the same time (multiple cores). It's a subset of concurrency.

Java provides tools to create/coordinate concurrent tasks safely.

---

2) Threads: the basic unit

- A **Thread** is an OS/VM schedulable unit of execution with its own call stack.
- Two common ways to create a thread:

```java
// 1. Extend Thread
class MyThread extends Thread {
    public void run() { System.out.println("hello"); }
}
new MyThread().start();

// 2. Implement Runnable (preferred)
Runnable r = () -> System.out.println("hello");
new Thread(r).start();
```

- **`start()`** creates the thread and invokes **`run()`** on that new thread; calling **`run()`** directly runs on current thread (common mistake).

3) Thread lifecycle (states)

- NEW → RUNNABLE → RUNNING (VM/jvm threads are RUNNABLE)
- WAITING / TIMED_WAITING (when waiting or sleeping)
- BLOCKED (waiting to acquire a monitor/synchronized lock)
- TERMINATED (finished)

Understanding states helps debug blocking and deadlocks.

---

4) Race conditions, visibility & atomicity (why synchronization is needed)

- **Race**: two threads access shared data and at least one writes -> outcome depends on scheduling.
- **Atomicity**: an operation is indivisible (e.g., `int++` is *not* atomic).
- **Visibility**: one thread's write may not be immediately visible to another thread unless rules (happens-before) are satisfied.
- **Ordering**: compilers/CPU reorder instructions; the Java Memory Model (JMM) defines rules so programmers can reason about reordering.

---

5) `synchronized` and intrinsic locks (monitors)

- `synchronized` gives mutual exclusion + memory visibility.

  ```java
  class Counter {
      private int count = 0;
      public synchronized void increment() { count++; }
      public synchronized int get() { return count; } // read with lock
  }
  ```

- Rules:
  - Only one thread can hold a given object's monitor at a time.
  - `synchronized` establishes **happens-before**: unlock → later lock sees writes.
- Use `synchronized(this)` or `synchronized(method)` or `synchronized(ClassName.class)` for static shared state.

Pros: simple, clear.
Cons: can be coarse-grained (block other threads), possible deadlocks if multiple locks acquired in inconsistent order.

---

6) `volatile`

- `volatile` ensures **visibility** and prevents certain reorderings but **does not** provide atomicity for compound actions.

```java
volatile boolean stopped = false;
// Thread A sets stopped = true;
// Thread B will eventually see stopped == true
```

- Use when:
  - The variable is read/written atomically (primitive or reference).
  - You need visibility guarantees without locking (e.g., a stop flag).
- `volatile` alone cannot replace a lock for `count++` (not atomic).

---

7) java.util.concurrent.atomic (Atomic classes)

- Atomic classes (e.g., `AtomicInteger`, `AtomicReference`) provide lock-free atomic operations using CPU CAS (compare-and-swap).

```java
AtomicInteger ai = new AtomicInteger(0);
ai.incrementAndGet(); // atomic
```

- Use for counters, simple state updates with retry semantics. Avoid overusing them for complex invariants (locks or transactions are better).

---

8) ReentrantLock and advanced locks

- `ReentrantLock` from `java.util.concurrent.locks` is explicit lock control:

```java
ReentrantLock lock = new ReentrantLock();
lock.lock();
try {
  // critical section
} finally {
  lock.unlock();
}
```

- Features vs `synchronized`:
  - tryLock(timeout), interruptible lock acquisition, fairness policy, ability to get waiting thread info.
  - `ReadWriteLock` (separate read & write locks) for many readers, few writers.
  - `StampedLock` for optimistic reads (Java 8+).
- Use when you need advanced features not available with `synchronized`.

---

9) High-level concurrency utilities (java.util.concurrent)

These reduce complexity and are preferred over manual thread management.

Executors and thread pools

- `ExecutorService` decouples task submission from task execution.

```java
ExecutorService ex = Executors.newFixedThreadPool(4);
ex.submit(() -> doWork());
ex.shutdown();
```

- Types: fixed pool, cached pool, single thread, scheduled pool.
- Use thread pools to avoid unbounded thread creation and to control concurrency.

Callable and Future

- `Callable<V>` returns a value and can throw exceptions; `Future` gives result or cancellation.

```java
Future<Integer> f = ex.submit(() -> 42);
int result = f.get(); // blocks or use isDone()
```

CompletionService & invokeAll

- `CompletionService` helps collect results as tasks finish.
- `invokeAll` executes a collection of tasks and waits for completion.

Blocking queues

- `BlockingQueue` implementations (e.g., `LinkedBlockingQueue`, `ArrayBlockingQueue`, `SynchronousQueue`) are used to implement producer-consumer patterns safely.

```java
BlockingQueue<Runnable> q = new LinkedBlockingQueue<>();
ThreadPoolExecutor tpe = new ThreadPoolExecutor(..., q, ...);
```

Other synchronizers

- `CountDownLatch` — one-time countdown latch (like waiting for N threads to finish).
- `CyclicBarrier` — barrier for N threads that can be reused.
- `Semaphore` — control number of permits (e.g., connection pool).
- `Phaser` — flexible barrier supporting dynamic parties.

- **`Exchanger`** — exchange objects between two threads.

---

10) Fork/Join and parallelism

- **`ForkJoinPool`** and **`RecursiveTask`** are for divide-and-conquer parallelism; useful with many small tasks (work-stealing).

  ```java
  class SumTask extends RecursiveTask<Long> { ... }
  ForkJoinPool pool = new ForkJoinPool();
  pool.invoke(new SumTask(...));
  ```

- Use for CPU-bound tasks that can be recursively split.

---

11) CompletableFuture (async pipelines)

- **`CompletableFuture`** gives non-blocking composition and chaining:

  ```java
  CompletableFuture.supplyAsync(() -> fetch())
    .thenApplyAsync(result -> transform(result))
    .thenAcceptAsync(x -> System.out.println(x));
  ```

- Supports combining, timeouts, exception handling without blocking threads unnecessarily.

---

12) Concurrent collections

- **`ConcurrentHashMap`**, **`CopyOnWriteArrayList`**, **`ConcurrentLinkedQueue`**, **`ConcurrentSkipListMap`** — designed for concurrent access without external synchronization.
- Use them instead of **`Collections.synchronizedMap()`** for better scalability (especially **`ConcurrentHashMap`**).

---

13) Java Memory Model (JMM) — core ideas (visibility & happens-before)

- **Happens-before**: a relation that guarantees memory visibility and ordering. If A happens-before B, writes by A are visible to B.
- Examples that create happens-before edges:
  - Program order within a thread.
  - Unlock (monitor) → subsequent lock (monitor) on same monitor.
  - Write to a **`volatile`** variable → subsequent read of that **`volatile`**.
  - Thread.start() happens-before actions in started thread.

  - Thread.join() happens-after the terminated thread's actions.
- Without happens-before, threads may see stale values due to CPU caches or instruction reordering.

---

14) Common concurrency bugs and how to avoid them

- **Deadlock**: two or more threads waiting forever for locks held by each other.
  - Avoid by acquiring locks in a fixed global order, minimize lock scope, or use `tryLock` with timeout.
- **Livelock**: threads keep changing state in response to other threads but make no progress.
  - Back off, use randomized retries, or design different retry logic.
- **Starvation**: a thread never gets CPU or lock (e.g., unfair locks). Use fairness only when needed.
- **Race conditions / data corruption**: fix with locks, atomic variables, or confinement.
- **Priority inversion**: lower-priority thread holds resource needed by higher-priority thread. Usually rare in typical JVM apps.

---

15) Design patterns & techniques

- **Thread confinement**: keep data local to a thread (no sharing) — simplest way to avoid concurrency issues.
- **Immutability**: immutable objects are thread-safe by default.
- **Guarded by lock**: document which lock protects which field (`@GuardedBy`).
- **Use higher-level abstractions** (`ExecutorService`, `CompletableFuture`, `BlockingQueue`) over raw `Thread`.
- **Avoid shared mutable state** if possible. Prefer message passing (queues) or immutable snapshots.
- **Fail-fast / defensive copies**: when sharing collections, consider copying or using concurrent collections.

---

16) Practical examples

a) Thread pool + tasks

```
java

ExecutorService ex = Executors.newFixedThreadPool(4);
List<Future<String>> futures = new ArrayList<>();
for (int i = 0; i < 10; i++) {
    int id = i;
    futures.add(ex.submit(() -> {
        Thread.sleep(100); // simulate
        return "done " + id;
    }));
}
for (Future<String> f : futures) {
    System.out.println(f.get()); // blocking wait
}
ex.shutdown();
```

b) Atomic counter vs synchronized

```java
// atomic
AtomicInteger ai = new AtomicInteger();
ai.incrementAndGet();

// synchronized
class C {
  private int count;
  public synchronized void inc() { count++; }
  public synchronized int get() { return count; }
}
```

c) Producer-consumer with BlockingQueue

```java
BlockingQueue<Integer> q = new LinkedBlockingQueue<>();
ExecutorService ex = Executors.newFixedThreadPool(2);

// producer
ex.submit(() -> {
  for (int i=0;i<100;i++) q.put(i);
  q.put(-1); // poison pill
  return null;
});

// consumer
ex.submit(() -> {
  int v;
  while ((v = q.take()) != -1) {
    process(v);
  }
  return null;
});
```

d) Deadlock example and fix

Deadlock (dangerous):

```java
synchronized(a) { synchronized(b) { ... } }
synchronized(b) { synchronized(a) { ... } } // if two threads run these, deadlock possible
```

Fix: acquire locks in the same order everywhere:

```java
synchronized(first) { synchronized(second) { ... } }
```

Or use **`tryLock`** with timeout.

e) CompletableFuture chaining

```java
CompletableFuture.supplyAsync(() -> fetchFromDb())
  .thenApply(result -> compute(result))
  .thenAccept(finalResult -> System.out.println(finalResult));
```

17) Debugging & profiling concurrent apps

- Tools/techniques:
  - Thread dumps (`jstack`) to see threads & locks.
  - VisualVM / Mission Control for profiling and monitoring.
  - Logging with thread id, stack traces.
  - Deadlock detection (some JVM flags and tools).
  - Use `-Djava.util.concurrent.ForkJoinPool.common.threadFactory=...` to instrument ForkJoin.

(Exact tool commands vary; check your JDK docs or IDE for specifics.)

18) Performance, scalability & pitfalls

- Coarse-grained locking limits concurrency (long critical sections).
- False sharing: multiple threads updating adjacent fields on same cache line cause performance degradation—structure data to avoid it.
- Over-threading: creating far more threads than cores/context-switching costs.
- Use profiling to find actual bottlenecks before optimizing.

19) Practical checklist (when writing concurrent code)

1. Can I avoid shared mutable state? Prefer immutability or confinement.
2. If sharing, which lock/strategy protects each piece of data? Document it.
3. Prefer higher-level constructs (`ExecutorService`, `BlockingQueue`, `CompletableFuture`).
4. Keep critical sections small.
5. Consider `ConcurrentHashMap` / atomics when appropriate.
6. Handle thread creation with pools and graceful shutdown (`shutdown()/shutdownNow()`).
7. Add timeouts when waiting on locks or external resources.
8. Write tests that try to expose concurrency issues (multi-threaded unit tests).
9. Log thread ids in important traces for easier debugging.

20) Short summary — when to use what

- **Use `synchronized`** for simple mutual exclusion.
- **Use `volatile`** for simple flags/visibility.
- **Use atomics** for atomic counters & simple state transitions.
- **Use `ReentrantLock`** for advanced locking features (timeouts, fairness).
- **Use `ExecutorService`** and thread pools for managing threads.
- **Use `CompletableFuture`** for non-blocking async composition.
- **Use `ForkJoinPool`** for divide-and-conquer parallelism.
- **Use concurrent collections & blocking queues** for safe shared data structures.

If you want, I can now:

- Convert any of the above examples into a complete, runnable IntelliJ project setup.
- Walk through a concrete debugging scenario (e.g., you're seeing a deadlock — paste thread dump and I'll analyze).
- Give targeted practice problems and solutions (e.g., implement a thread-safe LRU cache from scratch, or build a producer-consumer with multiple producers/consumers and a bounded queue).

Which of those would you like next?