**Thread (Class):**

A **thread** is the smallest unit of execution in a process. It runs independently but shares resources like memory with other threads in the same process. Threads enable multitasking within a program, improving performance and responsiveness.

**Thread Lifecycle :**

A thread goes through the following states:

1. **Born (New State)**
   1. A thread is created using new Thread().
   2. At this stage, the thread object exists but hasn’t started yet.
2. **Ready (Runnable State)**
   1. The thread is registered with the Thread Scheduler using start().
   2. It is ready to run but may wait for CPU time.
3. **Running (Executing State)**
   1. When the CPU assigns time, the thread executes its run() method.
4. **Dead (Terminated State)**
   1. Once the execution of run() is complete, the thread terminates.
   2. It cannot be restarted.

**Multithreading (Concurrency) :**

**Concurrency** means executing **multiple tasks at the same time**, but **not necessarily in parallel**.

🟢 In Java, concurrency is about handling **multiple threads** that run independently but share resources like memory.

**how to create threads in java**

1. **Extending the Thread Class**
2. **Override** the **run()** method.
3. It is **less reusable**
4. Java supports single inheritance so extending thread class restricts flexibility
5. **Implementing the Runnable Interface (Preferred)**
6. instead of extending from thread class we can implement from runnable interface and **override run().**
7. create thread class object & new thread object and pass this object to thread class.
8. **The Runnable approach is preferred for flexibility.**
9. It allows multiple threads to share same object.
10. We can extend other class
11. **It is less reusable**
12. **Implementing Callable Interface (Java 5)**
13. If we **want to return** something from thread then we can create a thread by **implementing Callable** Interface.
14. It may throw a Checked Exception. Callable<V> is a type of the result returned.
15. **Future** **object** is used to **collect** the **response** from **Callable** thread.
16. **It works asynchronously** means execution of thread is done at the backend and at the same time other tasks can be performed and when the thread execution is done it returns the response.
17. **Lambda Expression :**
18. you can create a thread using a lambda expression **by passing a lambda** to the **Thread** **constructor**, which **takes a Runnable as an argument**.
19. **Modern** and **clean**, no need to extends thread class
20. **great for short and simple task.**

**Thread Methods :**

**Start():**

1. Present inside Thread class that creates and starts a new thread, calling the run() method in that separate thread.
2. **start() is used to register a thread with the thread scheduler** in Java.
3. if we call start method twice -> throw IllegalThreadStateException

**Run():**

1. It is defined in the Thread class and the Runnable interface.
2. The run() method in Java is where the actual logic of the thread is written.
3. It is automatically called when start() is invoked.
4. If called directly (without start()), it runs like a normal method on the **current thread** instead of creating a new thread.

**Thread.currentThread().getName()** to get the name to thread

**ThreadObject.setName("abc")**  to change thread name

**ThreadObject.getId();**  to get thread id

**sleep(ms)** Makes thread sleep for ms milliseconds.

**join()** Makes current thread wait until another thread finishes.

**yield()** Gives CPU control to another thread.

**interrupt()** Interrupts a sleeping or waiting thread.

**isAlive()**  Checks if a thread is running.

**Thread Synchronization (Keyword) :**

1. It is used when multiple thread is sharing the same resource, to avoid thread inconsistency and race condition, and to provide thread safety.
2. Synchronized keyword is used at block level or method level, when we want whole method to be synchronised, we use method level sync and when we want only a particular block of code to be in sync we use block level sync.
3. When the method is static class level lock is created and when the method is non-static object level lock is provided.

**Disadvantages :**

**Avoid synchronized keyword if possible - instead use advance multithreading concepts like lock, thread local**

1. Only one thread can access at a time leads to **low performance**
2. Waiting time is increased.
3. **Fairness Policy** : No guarantee that the waiting queue will be maintained or not.
4. **No Flexibility** : It is only for single method and not across the methods
5. **Fine-grained locking** is not possible.

**Non-Static Method:**

1. **Non-static methods** belong to **objects**, not the class.
2. Synchronization is needed if multiple threads modify shared data(object).
3. The **same object (task) is shared** by t1 and t2.
4. Synchronized(this) { } //this keyword points to current object

**Static method:**

A **static method** belongs to the **class, not an instance**.

1. Both t1 and t2 call the same method at the **class level**.
2. Use synchronized to make static methods thread-safe.
3. Synchronized(className.class) { }.

**These are the interthread communication methods:**

* They are object class methods and are used for inter-thread communication within synchronized block
* We can't call **wait(), notify() and notifyAll()** methods without synchronized block (lock) otherwise it will give **IllegalMonitoStateException.**

**Wait()**

* Causes the current thread to wait until another thread calls notify() or notifyAll() on the same object.
* It releases the lock on the object and moves the thread to the waiting state.
* Can only be called from within a synchronized block or method.

**synchronized(obj) {**

**obj.wait(); // Thread waits and releases the lock on 'obj'**

**}**

**notify()**

* Wakes up one of the threads that are waiting on the object
* The chosen thread goes from waiting to runnable state, but it doesn’t get the lock immediately — it must wait until the current thread releases it.
* Also, must be used inside a synchronized block.

**synchronized(obj) {**

**obj.notify(); // Wakes up one waiting thread**

**}**

**notifyAll()**



* Wakes up all threads waiting on the object.
* Only one thread will acquire the lock when it’s available, the rest will keep waiting.

**synchronized(obj) {**

**obj.notifyAll(); // Wakes up all waiting threads }**

Method to check the number (no.) of cores present inside the system **Runtime.getRuntime().availableProcessors();**



****

**Interthread Communication/Coordination - Producer-Consumer Problem**

**Thread Group (Class) :**

1. **In Java, a ThreadGroup is used to group multiple threads into a single unit for easier management. It helps organize threads in a hierarchy and allows actions like interrupting, suspending, or resuming all threads in the group at once.**
2. ThreadGroup is part of **java.lang package**. It is a class that can be used to group threads and even other thread groups (forming a tree-like hierarchy).
3. Thread groups can **contain child groups**, allowing nested management. This creates a **parent-child relationship** between thread groups.

**Disadvantages :**

1. **Deprecated –** It is considered as outdated
2. No fine-grained control
3. No thread prioritization

**Methods of Thread Group :**

**getName() -** Returns the name of the thread group.

**getParent() -** Returns the parent thread group.

**interrupt() -** Interrupts all threads in the group**.**

**activeCount() -** Returns the number of active threads in the group.

**enumerate(Thread[] list) -** Copies active threads into the given array.

**Advanced Multithreading - Executor Services and Callable Future :**

**Thread Starvation:**

Thread starvation happens when a thread is ready to run but is unable to get CPU time because other threads (usually with higher priority or holding resources) keep executing.

**Multiple reason of thread starvation :**

1. A thread is created but not assigned a task, or it waits too long to get CPU time.
2. High-priority threads continuously take precedence.
3. Locks held for too long or not released properly.
4. Unfair scheduling or resource contention.

**We can go for ExecutorService for better thread management, and use ReentrantLock that provides fairness policy**

**What is Thread Local Cache?**

When a program runs with multiple threads, each thread has its **own memory space** called thread-local cache (or thread-local storage).

It's like each thread has its own copy of a variable stored locally, instead of always accessing the shared main memory (RAM).

This is done to **improve** **performance**, because:

* Accessing local memory is much faster than accessing shared memory.
* But this speed comes at a cost: **data inconsistency**

**Problem It Can Cause**

If thread A updates a variable, and thread B is still reading from its local copy, then B won’t see the updated value. That’s called a visibility problem.

**Thread local (Class):**

1. **ThreadLocal** in Java is a special class that provides **thread-local variables —** that is, **each thread accessing the variable has its own, independently initialized copy** of the variable.
2. It is used when you want to store thread specific values like : User session, database connection, request-specific data in web application.
3. Accessing normal variable by multiple threads can case issue like race condition. With threadlocal each thread will have its own copy of that variable.
4. Internally, ThreadLocal uses a **map** (per thread) to store values, where the key is the ThreadLocal instance. Each thread can **set** and **get** its own value without affecting others.

**Scenario: Web Application Handling User Requests**

Imagine you’re building a web application where each incoming user request is processed by a separate thread. For every request, you need to store user-specific information—like a user ID or session data—that should be available throughout the request-handling process. However, this data must remain isolated to the specific request and not be accidentally shared with other requests (i.e., other threads).

**Problem Without ThreadLocal**

Using a shared variable:

* Thread A sets user ID = "User123"
* Thread B sets user ID = "User456"
* Now Thread A sees "User456" → leads to data corruption and security risks.

**Solution Using ThreadLocal**

With ThreadLocal:

* Thread A stores "User123"
* Thread B stores "User456"
* Both threads access their own values independently, avoiding conflicts.

**How ThreadLocal Works in This Scenario**

**Define** a static ThreadLocal variable

**Set** the value at the start of request

**Get** the value wherever needed during processing

**Remove** the value after the request to avoid memory leaks

**What is volatile?**

volatile is a **keyword** in Java used to declare variables that are **shared between threads**, and it ensures that:

**Any change made by one thread to that variable is immediately visible to all other threads.**

**Why do we need it?**

In multithreading, threads can **cache variables** in their own memory. That means:

* One thread **might not see the updated value** from another thread!
* This can cause **weird bugs** where one thread thinks a variable is still true when another thread already set it to false.

**Syntax :**

volatile boolean running = true;

**Use volatile when:**

* You have a flag like **isRunning, isFinished, shouldStop**
* You want to **stop a thread safely**
* You want to **avoid full locking (synchronized)** but still share a variable safely

**Don’t use it when:**

* You need **atomic operations (like increment/decrement)**
* You’re dealing with **compound actions (read + modify + write)**

**Executor Service (Interface) :**

ExecutorService is an interface that extends Executor and helps manage a group of threads more easily. It simplifies thread management by:

* Handling thread creation, reuse, and termination.
* Allowing you to submit tasks (Runnable or Callable) and manage their execution.
* Offering methods to control the lifecycle of the thread pool and retrieve results.

Think of it as a "thread manager" that takes care of the low-level details, letting you focus on the tasks you want to execute.

1. Part of java.util.concurrent
2. Manages and controls a pool of threads
3. Executes tasks asynchronously
4. Better than manually using Thread
5. Common implementations of ExecutorService include ThreadPoolExecutor and ScheduledThreadPoolExecutor.

**Syntax :**

**ExecutorService executor = Executors.newFixedThreadPool(3);**

It contains key methods like :

1. submit() : Submit a single task (Runnable or Callable). Returns value / Future<T>
2. execute() : To **submit a task** (as a Runnable) for **asynchronous execution**. **Return type**: void
3. invokeAll() : Run a collection of Callables, wait for all
4. invokeAny() : Run a collection of Callables, return first result
5. shutdown() : Gracefully shut down (no new tasks)
6. shutdownNow() : Attempts to stop all actively executing tasks and halts the processing of waiting tasksisShutdown() : Check if shutdown was called
7. isTerminated() : Check if all tasks have finished
8. awaitTermination() : Blocks until all tasks finish after a shutdown request or timeout occurs

**invokeAll(Collection<? extends Callable<T>> tasks)**

* **What it does**:  
  This method takes a **collection of Callable tasks**, runs **all of them**, and **returns a list of Future<T> objects**, one for each task.

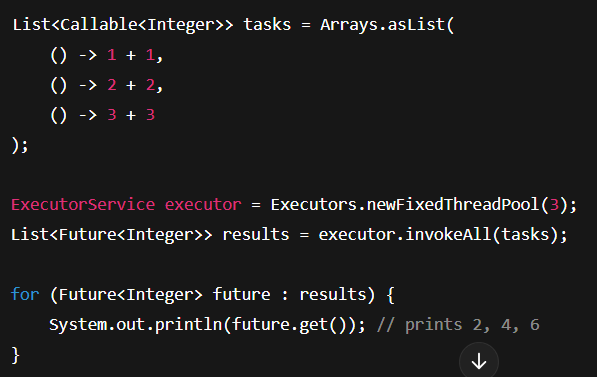
All tasks are submitted at once.

It **waits until all tasks are completed** (either successfully or with exceptions).

The returned list contains Future objects in the **same order** as the tasks in the collection.

You can call .get() on each Future to retrieve the result or check for exceptions.

Used When you want to run a bunch of tasks and need the result from **each one** after all of them have finished.



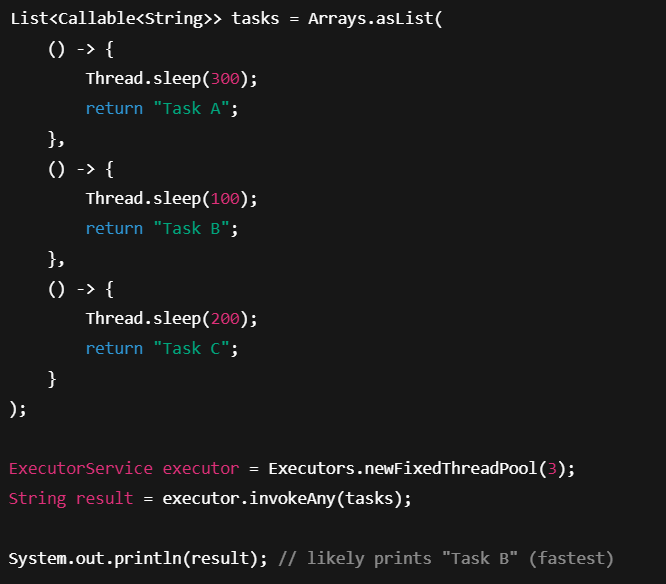
**invokeAny(Collection<? extends Callable<T>> tasks)**

* **What it does**:  
  This method takes a **collection of Callable tasks**, runs them, and **returns the result of the first task that completes successfully** (without throwing an exception).

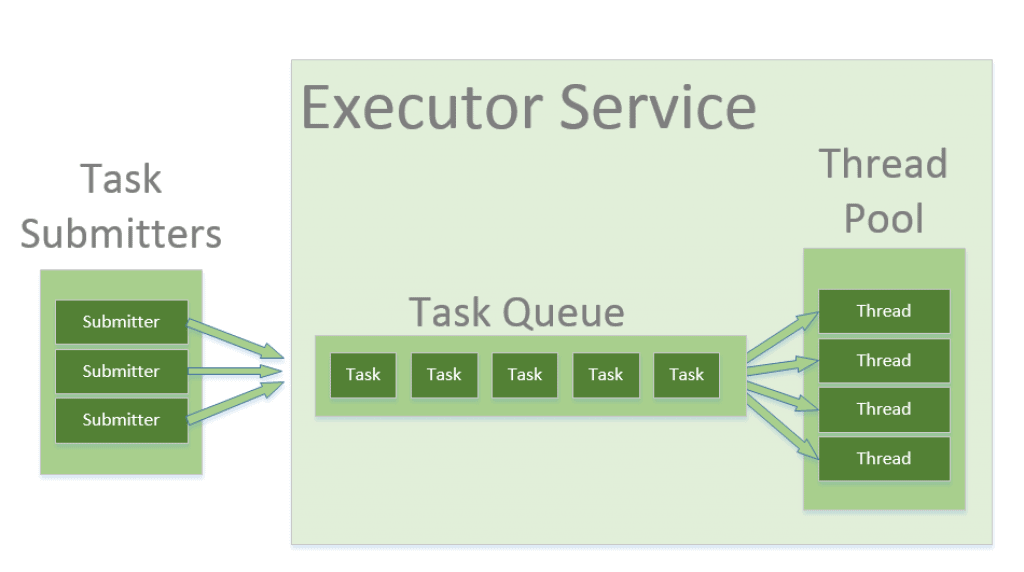
It **executes all tasks** but only returns **one result**.

As soon as **one task finishes successfully**, it **cancels the others**.

If all tasks fail, it throws an ExecutionException.  
Used When you want the **first successful result** from a group of tasks and don’t care about the rest.



**Thread Pool:**



A Thread Pool is a pool of pre-created worker threads that efficiently manage task execution. Instead of creating a new thread for every task, tasks are submitted to the pool, and available threads execute them.

**Why thread pool?**

1. Reuses threads instead of creating/destroying them frequently.
2. Limits the number of concurrent threads to avoid excessive CPU/memory usage.
3. Efficiently schedules and executes tasks using an internal queue

**How threadPool works :**

ExecutorService uses a thread pool to manage thread execution. For example:

* When you submit tasks to an ExecutorService, it internally uses a thread pool (like ThreadPoolExecutor) to execute those tasks.
* The thread pool maintains a number of threads and assigns tasks to them. If there are more tasks than threads available, some tasks may wait in a queue until a thread becomes free.
* Once a task is completed, the thread is returned to the pool for reuse, reducing the overhead of creating new threads.

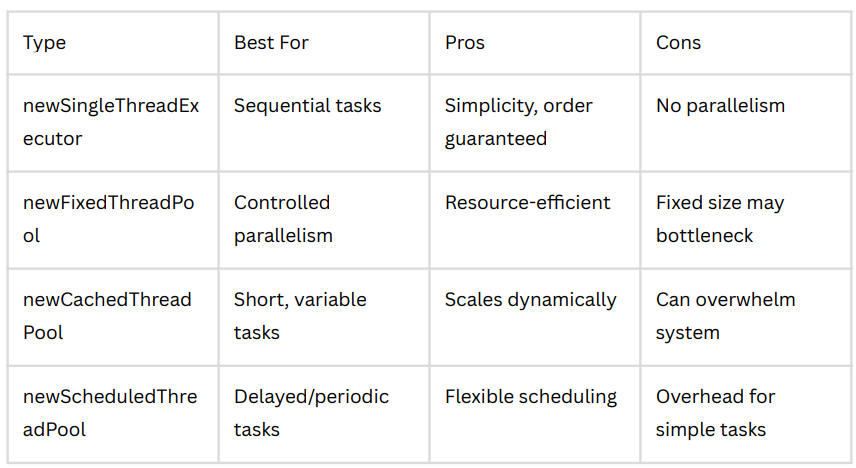
**Types of Thread Pools in Java:**

Java provides different types of thread pools via **Executors** class:

1. **newFixedThreadPool(n)** -> Fixed number of threads (n). Extra tasks wait in a queue.
2. **newCachedThreadPool()** -> Creates new threads as needed but reuses idle threads. When we don’t know how many threads will be needed and task is very small. And thread Is created dynamically.
3. **newSingleThreadExecutor()** -> When we want only single thread in thread pool.

A single thread processes tasks sequentially.

1. **newScheduledThreadPool(n)** -> Schedules tasks with delay or periodically.
2. **ForkJoinPool()** : It splits a large task into smaller subtasks (fork), executes them in parallel, and combines the results (join).

****

**Java Concurrency :**

**Lock (Interface) :**

1. it is an interface present inside the concurrent package which has implementation like reinterned.
2. avoids deadlock.
3. provides flexibility - takes the locks across the methods
4. can specify the waiting time - avoids the deadlock

**ReentrantLock (from java.util.concurrent.locks) :**

ReentrantLock is a class that offers **explicit locking** with **more advanced features** compared to the synchronized keyword.

**Key Features of ReentrantLock:**

1. Re-entrant : A thread can **acquire the same lock multiple times** without blocking itself.
2. Fairness : Use new ReentrantLock(true) to enable **FIFO (First-In-First-Out)** lock granting.
3. Interruptible Locking : A thread waiting to acquire the lock can be **interrupted**
4. Try-Lock : Use tryLock() to attempt to acquire the lock **without blocking**.

Checking Priority: int priority = thread.getPriority();

Setting Priority: thread.setPriority(10); // Range: 1 (MIN) to 10 (MAX), default is 5

**Important Notes on Thread Priority:**

* Priorities are **hints to the thread scheduler**, not guarantees.
* The **OS and JVM implementation** determine how priorities are handled.
* **Higher priority does not ensure earlier or more frequent execution**.
* In most real-world cases, output is **unpredictable** and **non-deterministic**.

**Difference between Synchronised and Asynchronized?**

**Synchronized (or *Synchronous*) :**

Operations happen one after another, in a specific order, and often with locks to ensure only one thread can access a resource at a time.

Characteristics:

* Only one thread executes the block/method at a time.
* Often used to prevent race conditions in multi-threaded environments.
* Ensures data consistency and thread safety.

Downside:

* Can slow things down (less parallelism).
* Can lead to thread blocking or even deadlocks if not handled carefully.

**Asynchronous (or *Asynchronized*) :**

Operations can happen independently or in parallel, without waiting for others to complete.

Characteristics:

* Threads don’t block — they can continue doing other work.
* Often used in event-driven, non-blocking, or callback-based programming.
* Improves performance and responsiveness (like in UIs, server requests, etc.)

**Default Nature of any method (Sync.. or Async..)**

1. Methods are synchronous by default.
2. They are not thread-safe unless you make them synchronized.
3. They are not asynchronous unless you specifically run them in a new thread or use async frameworks (e.g., CompletableFuture, ExecutorService).

**Questions :**

**1. What is a thread in Java?**

**2. How do you create a thread in Java?**

**3. What is the difference between start() and run()?**

**4. What are the thread states in Java?**

**5. how we can apply manual lock to thread?**

**6. What is the difference between a process and a thread?**

* **Process** : A process is an independent program in execution. Each process has its own memory space, resources (like file handles), and data.
* **Thread** : A thread is a smallest unit of execution within a process. Multiple threads can exist inside one process, and they share the same memory and resources of that process.
* **E**.g. Process : chrome open multiple tabs; each tab can be a separate process so that if one crashes, others remain unaffected. Thread: Within one tab, multiple threads handle different tasks – one for rendering, one for loading content. They all share the same memory but do different jobs at the same time.

**7. What are daemon threads?**

* Daemon threads are **background threads** that provide support services to user threads (main or worker threads). They run in the background and do not prevent the program from exiting.
* They run in background have **low priority** and ends automatically and are created like normal threads
* The daemon thread runs for a short time, but **as soon as the main (user) thread ends**, the daemon thread is **terminated automatically**.
* Must set .setDaemon(true) **before calling** .start().
* You can check if a thread is daemon using .isDaemon().
* Daemon threads are **not used for critical tasks** (like saving data), because they may be terminated at any time.

**7. What is thread synchronization?**

* In multithreading, multiple threads can run at the same time, and if they share data, they might try to read/write at the same time, causing race conditions or incorrect results.
* If two threads call increment() simultaneously, they may both read the same value and overwrite each other’s result, leading to incorrect counting.
* By using synchronized, only one thread can execute this method at a time, ensuring data consistency.

**8. What is a deadlock in multithreading?**

A deadlock occurs when two threads hold locks and wait indefinitely for each other and for the resource to be accessible.

**9. What is thread starvation?**

Thread starvation happens when low-priority threads never get CPU time due to higher- priority threads continuously running.

**10. What is the difference between wait(), notify(), and notifyAll()?**

**11. What is the difference between ExecutorService and ThreadPool?**

ExecutorService manages a pool of threads for better performance.

ThreadPool avoids frequent thread creation and improves efficiency.

**17. What is thread-local storage?**

Each thread gets its own independent copy of a variable.

**12. What is Callable and Future in Java?**

Callable allows a thread to return a result, and Future retrieves it.

**13. What is a ForkJoinPool?**

A ForkJoinPool splits a large task into smaller subtasks (fork), executes them in parallel, and combines the results (join).

**14. What is a CompletableFuture?**

**CompletableFuture** is a class in Java (introduced in **Java 8**) used for **asynchronous** programming. It allows you to run tasks in the background without blocking the main thread and lets you **chain multiple tasks**, handle results, and manage exceptions easily.

**Key Points:**

* Runs code **asynchronously** using supplyAsync() or runAsync().
* Supports **chaining** with methods like thenApply(), thenAccept(), and thenCompose().
* Helps in **combining** multiple futures using thenCombine(), allOf(), and anyOf().
* Provides **exception handling** using exceptionally() and handle().

**15. What is a ReentrantLock and why use it?**

**16. What is the difference between invokeAny() and invokeAll()?**

**17. What is concurrency ?**

**Concurrency** in Java means the ability to **run multiple tasks at the same time**, or **manage multiple tasks efficiently** so they appear to run simultaneously. It helps improve performance, especially when dealing with I/O operations, multiple users, or large workloads.

**Key Points:**

* Achieved using **threads** or **executors**.
* Java provides tools like **Thread**, **Runnable**, **ExecutorService**, **ForkJoinPool**, and **CompletableFuture**.
* Concurrency helps in **better CPU utilization** and **responsive applications**.
* Requires handling issues like **race conditions**, **deadlocks**, and **thread safety**.

**Example use case**: A web server handling multiple user requests at the same time using threads.

**What is Parallelism?**

**Parallelism** means **executing multiple tasks at the same exact time**, typically by using **multiple CPU cores**. It’s a form of concurrency, but with true simultaneous execution.

**Key Points:**

* Focuses on **splitting a big task** into smaller subtasks and running them in **parallel**, often using **multi-core processors**.
* Improves performance for **CPU-bound** operations (e.g., data processing, calculations).
* Achieved in Java using tools like **Fork/Join framework**, **parallel streams**, or **CompletableFuture**.

**locking methods in multithreading**

**1. synchronized keyword**

1. Most basic locking mechanism.
2. Allows only one thread at a time to execute the synchronized block/method.
3. Works at method or block level.
4. Easy to use, but can lead to performance issues if overused.

**2. ReentrantLock (from java.util.concurrent.locks)**

1. More advanced and flexible than synchronized.
2. You need to manually lock and unlock the code.

Supports features like:

1. Try-lock (try to get the lock without waiting)
2. Fairness (first-come-first-served)
3. Interruptible lock (can stop waiting if interrupted)

**3. ReadWriteLock (for read-heavy operations)**

1. Allows multiple threads to read simultaneously.
2. But only one thread can write (and no reads allowed during writing).
3. Improves performance when there are **more reads than writes**.

**4. StampedLock (Java 8+)**

1. Advanced lock supporting optimistic reading.
2. Faster than ReadWriteLock in some scenarios.
3. **Use when you need high concurrency with frequent reads.**

**5. synchronized + volatile combo**

1. volatile ensures visibility, not locking.
2. Sometimes used with synchronized to keep code thread-safe and performant.

**Explain how Java handles multithreading and concurrency. What are the risks?**

Java provides built-in support for multithreading and concurrency using the java.lang.Thread class and java.util.concurrent package.

**Key Concepts:**

**1. Threads**

* **A thread is a lightweight unit of execution.**
* **Java allows multiple threads to run in parallel, sharing the same memory space.**

**2. Thread Creation**

* **Extend Thread class or implement Runnable / Callable interface.**

**3. Concurrency API**

* **Java 5+ provides ExecutorService, Future, CompletableFuture (Java 8), Locks, Semaphores, etc., via java.util.concurrent.**

**4. Synchronization**

* **Java uses synchronized, volatile, and locks (ReentrantLock) to manage shared resource access.**

**5. Thread Communication**

* **wait(), notify(), notifyAll() help threads communicate inside critical sections**

**How you perform multithreading?**

I can create threads by extending Thread or implementing Runnable, but in real-world applications I prefer using ExecutorService or CompletableFuture for better thread management, reusability, and performance. If I need a return value, I use Callable with Future. I also ensure thread-safety using synchronization or concurrent utilities like AtomicInteger

**Multithreading and multitasking?**