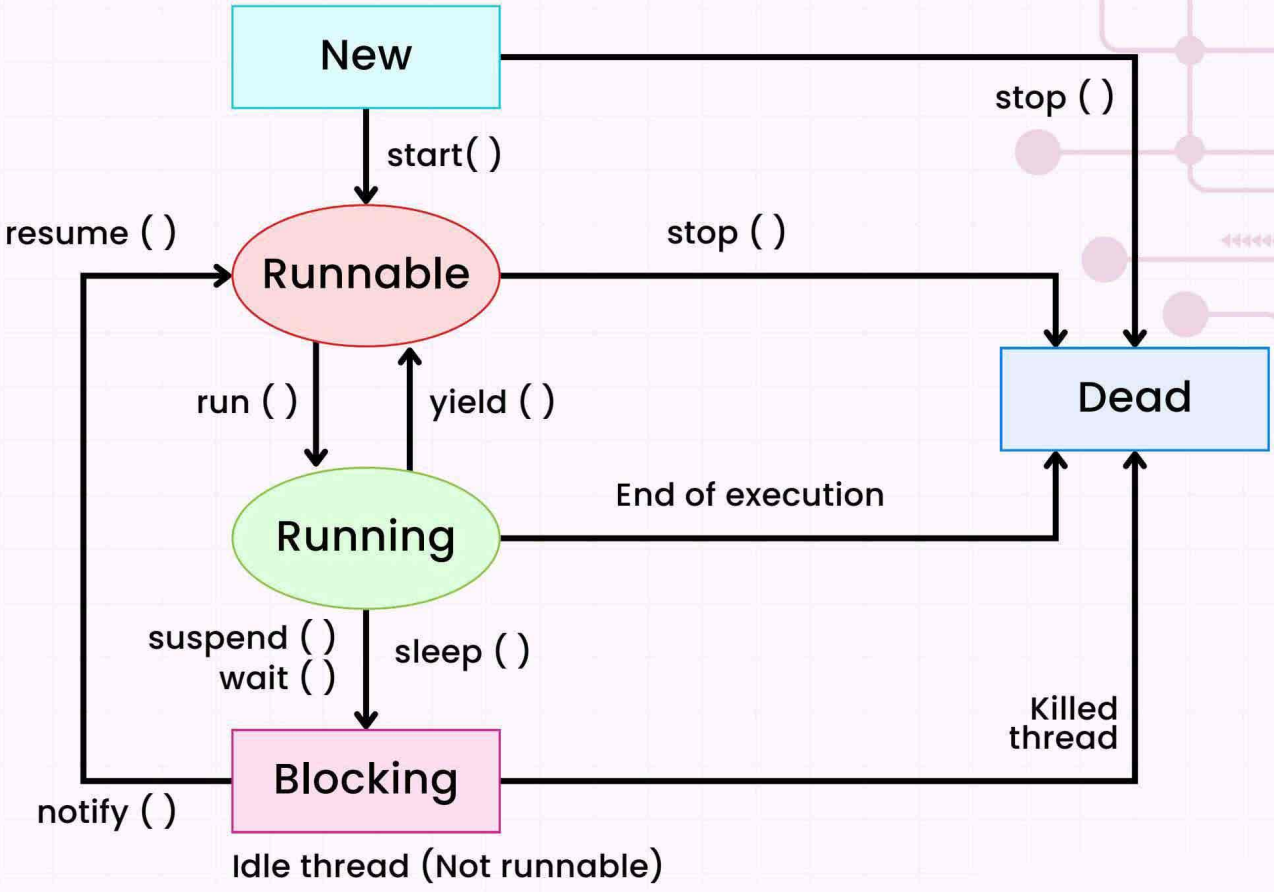
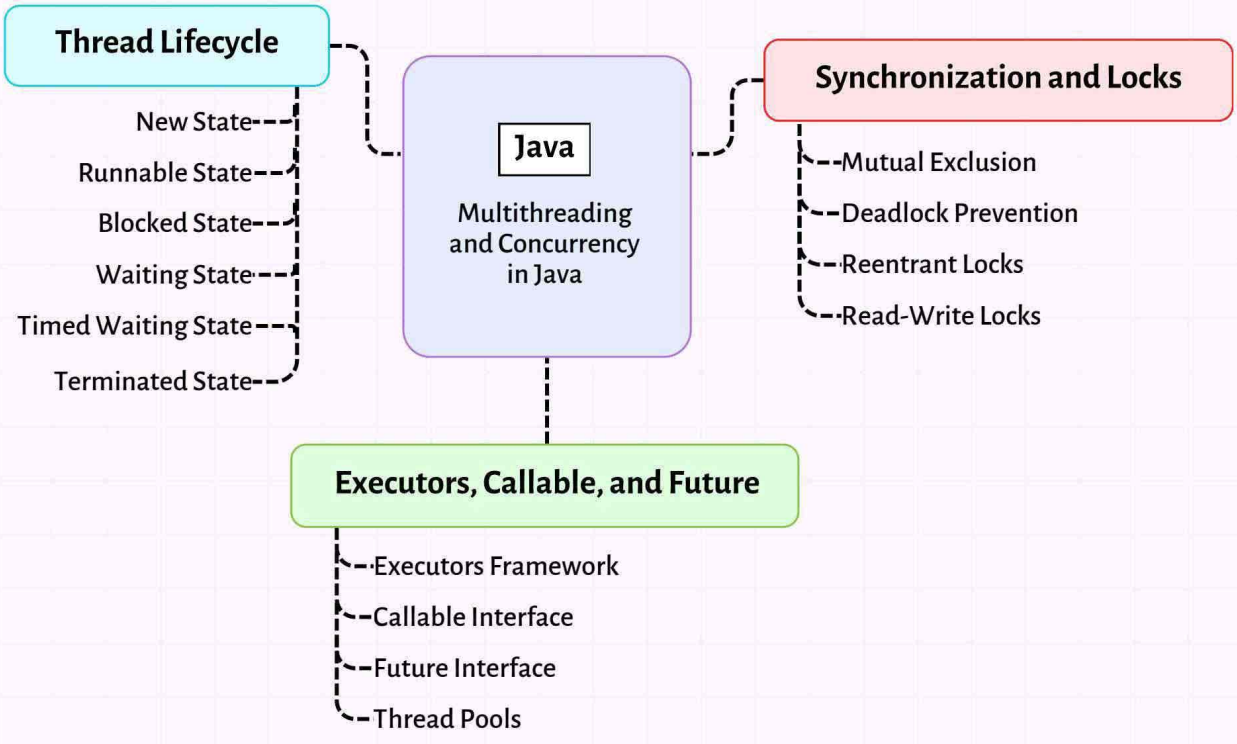
Multithreading and Concurrency in Java



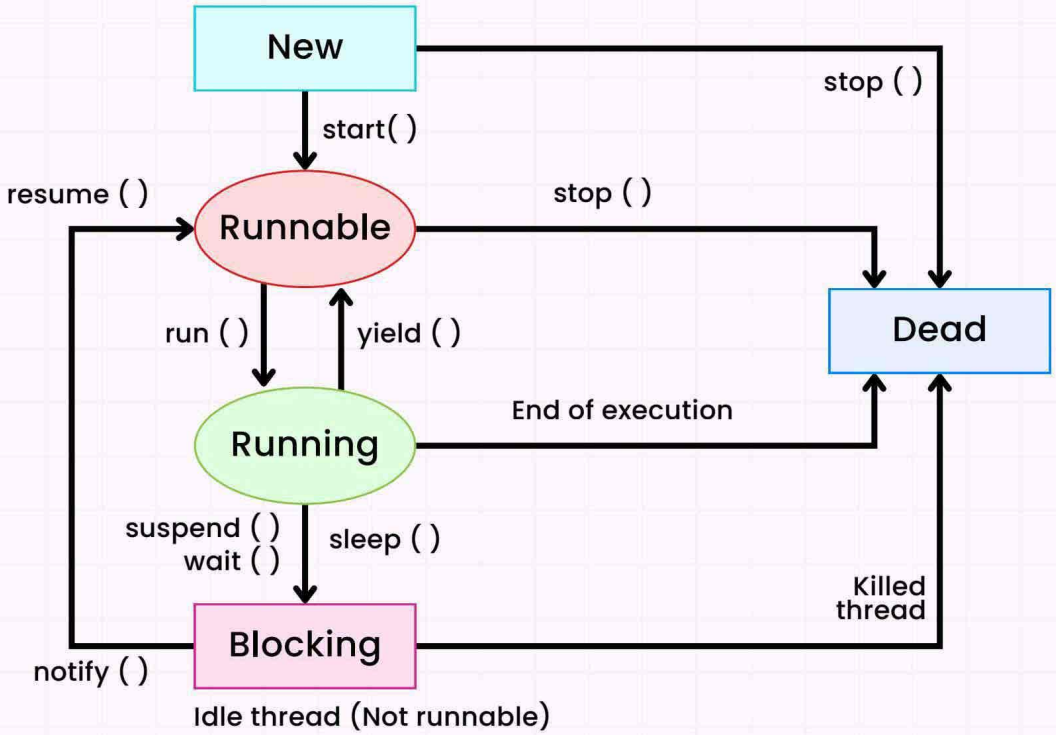
Chapter 5: Multithreading and Concurrency

- Multithreading and Concurrency are essential aspects of building highly efficient, responsive, and scalable applications in Java.
- Multithreading allows a program to execute multiple threads simultaneously, making full use of the system's processing power.
- In this chapter, we will explore key concepts such as the Thread Lifecycle, Synchronization and Locks, and tools like Executors, Callable, and Future, which make managing multiple threads simpler and more efficient.



1. Thread Lifecycle

- In Java, a thread goes through various states from creation to termination.
- Understanding the thread lifecycle helps in managing threads effectively.



Thread States:

- **New:** The thread is created but not yet started. It's in a new state.
- **Runnable:** After calling the start() method, the thread enters the runnable state, where it's ready to run when the CPU is available.
- **Blocked:** The thread is blocked if it's waiting to acquire a lock or resource that another thread is holding.
- **Waiting/Timed Waiting:** The thread enters this state if it's waiting for another thread to perform a specific action (e.g., calling join() or sleep()).
- **Terminated:** The thread completes execution or is terminated due to an exception. Once terminated, it cannot be restarted.

Real-Life Example

Imagine a worker (thread) at a factory. When hired (new state), the worker waits for the task (runnable state). If the machine they need is occupied (blocked), they must wait. When the machine is free, they work, and finally, when the task is completed, they clock out (terminated).

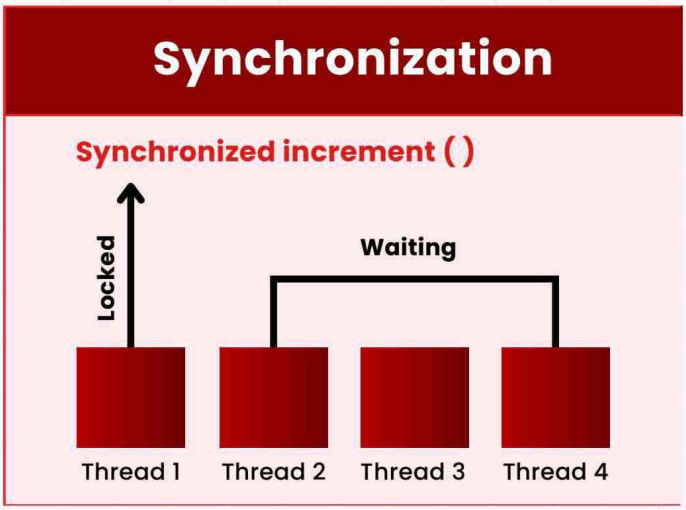
1. **Thread Attempts Access** — A thread tries to access a shared resource.
2. **Synchronized Keyword Applied** — The synchronized keyword is used to lock the resource.
3. **Resource Locked** — The resource is locked, preventing other threads from accessing it.
4. **Thread Accesses Resource** — The thread accesses the resource while it is locked.
5. **Resources Unlocked** — The resource is unlocked after the thread has finished accessing it.

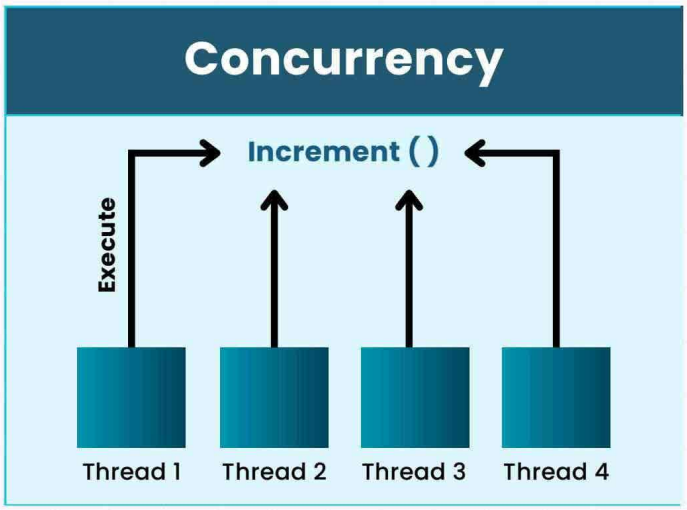
Tip

Threads cycle through these states, and understanding these transitions can help you manage multithreading more efficiently.

2. Synchronization, Locks, Deadlocks

- Concurrency in Java can lead to issues when multiple threads try to access shared resources simultaneously.
- Synchronization and locks are used to prevent such issues.

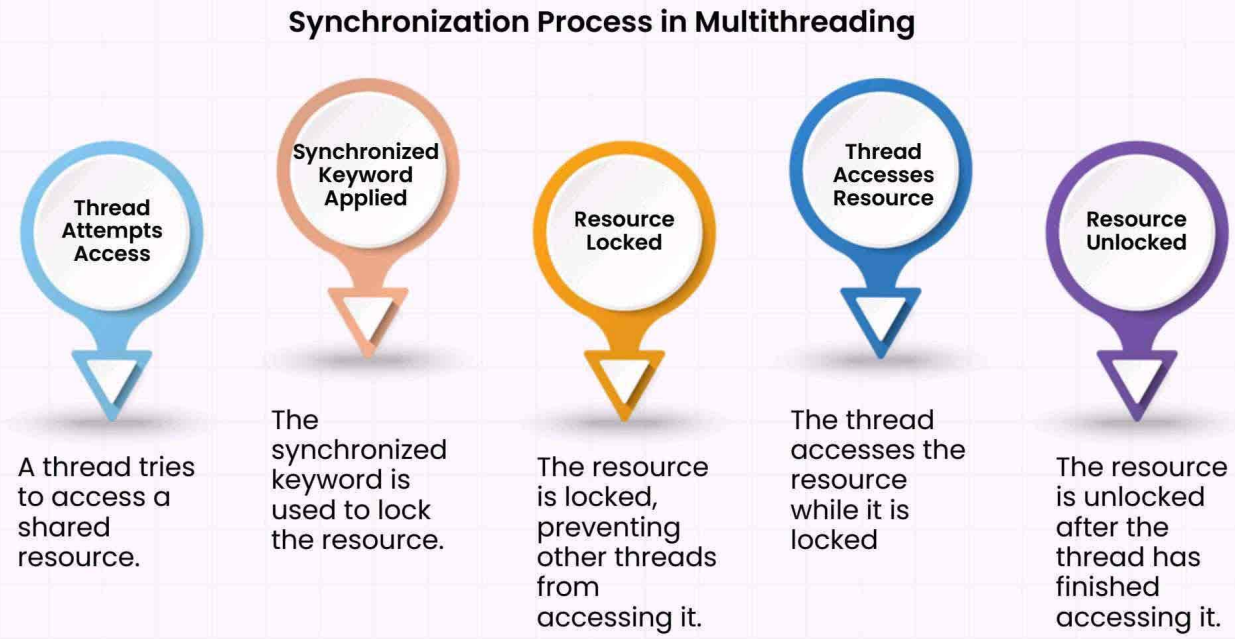




Synchronization:

- Synchronization ensures that only one thread can access a shared resource at a time, avoiding data inconsistency.
- It can be achieved using the synchronized keyword, which locks the resource while a thread is accessing it.

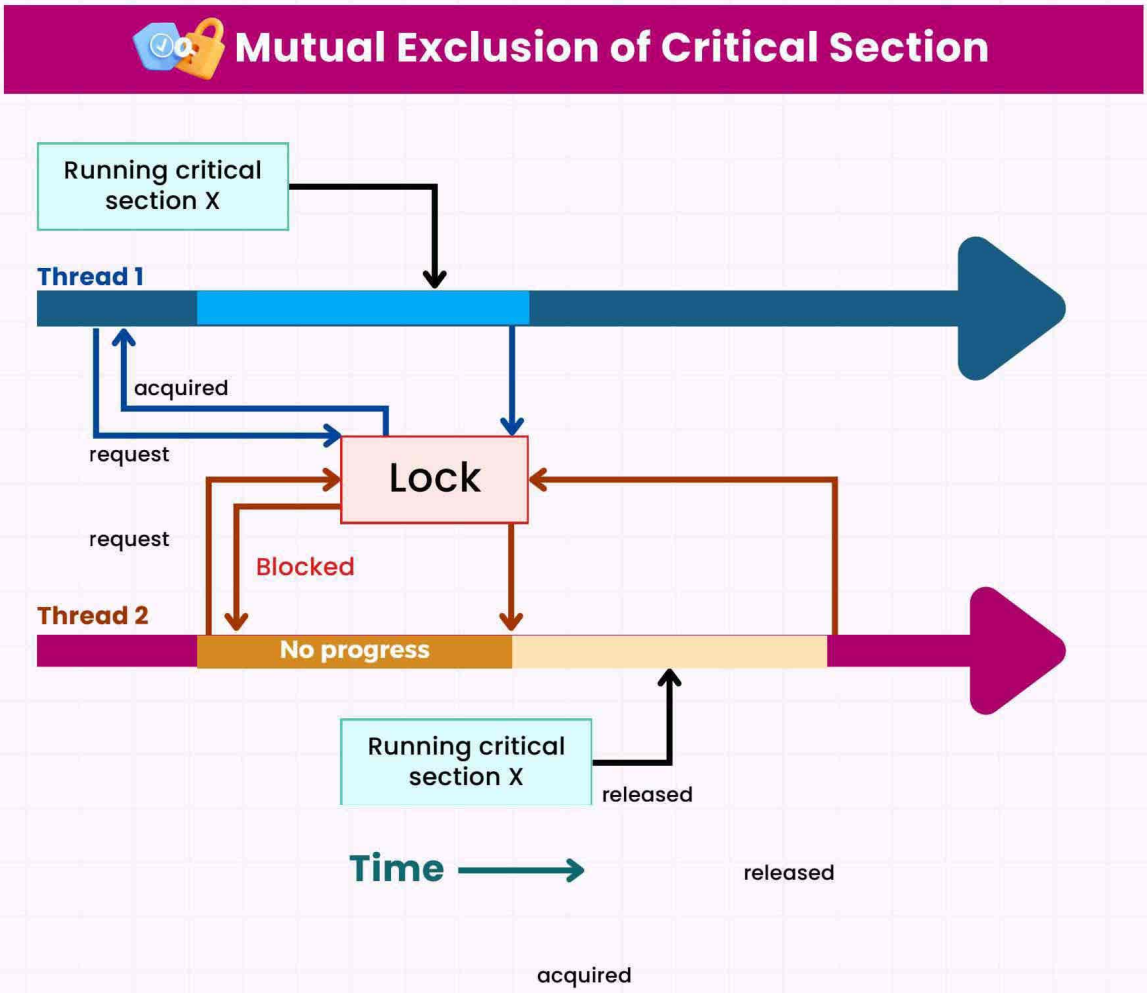
```
public synchronized void increment() {
    // Critical section of code
}
```



Locks:

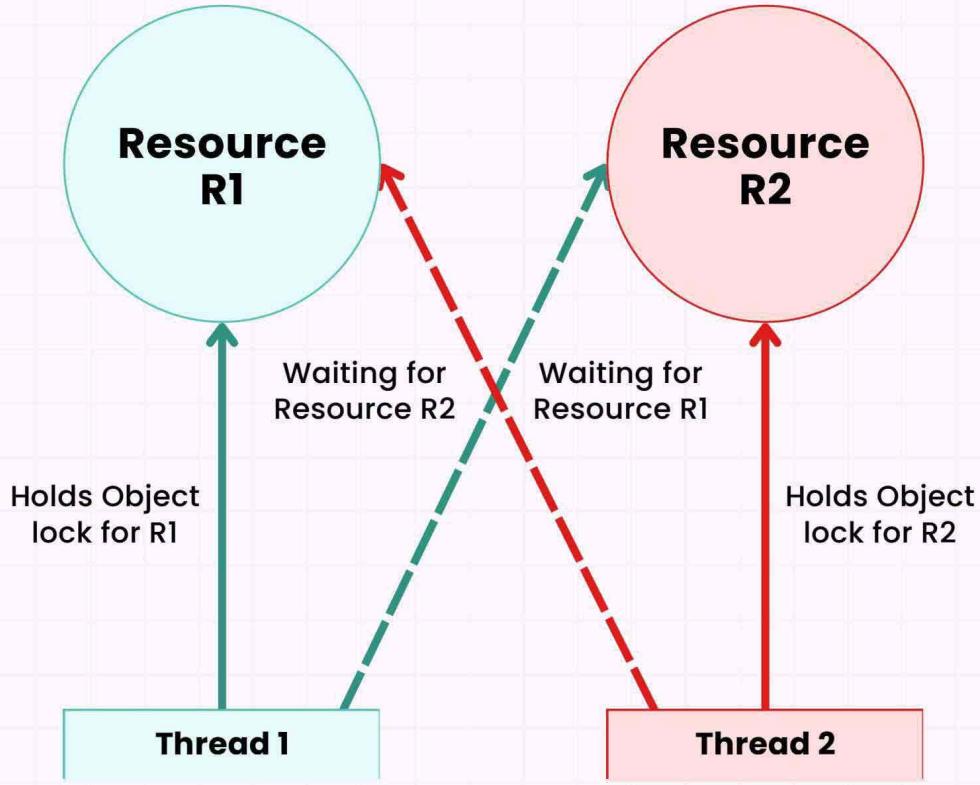
- A more flexible and advanced form of synchronization. Java's Lock interface (in java.util.concurrent.locks) allows more control over thread synchronization than the synchronized keyword.
- You can use ReentrantLock to lock and unlock explicitly.

```
Lock lock = new ReentrantLock();
lock.lock();
try {
    // Critical section of code
} finally {
    lock.unlock();
}
```



Deadlocks:

- Deadlocks occur when two or more threads are blocked forever, each waiting for the other to release a resource.
- Avoid deadlocks by following best practices such as using a consistent order to acquire locks or using timeouts.



Here, both threads are waiting for each other to unlock resources R1 and R2, but thread2 cannot release lock for resource R2 until it gets hold of resource R1.

Example of Deadlock Condition

Real-Life Example

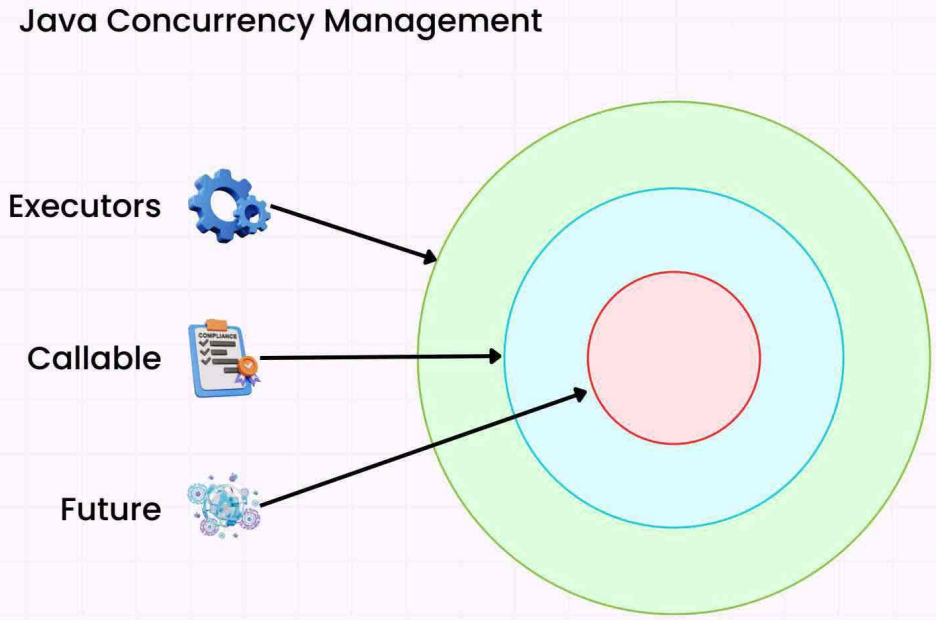
Synchronization is like a key to a room. Only one person (thread) can enter the room at a time. A deadlock is like two people (threads) waiting for each other's keys—they both need each other's key but neither can proceed, resulting in a standstill.

Tip

Always ensure that locks are released properly, and avoid circular dependencies to prevent deadlocks.

3. Executors, Callable, and Future

Java provides higher-level constructs to manage threads more effectively than manually creating and managing individual threads.



Executors:

- The Executor Framework provides a way to manage thread pools and handle tasks concurrently.
- Instead of manually creating threads, you can submit tasks to an executor for execution.
- The ExecutorService interface represents a thread pool and allows submitting tasks using methods like submit().

```
ExecutorService executor = Executors.newFixedThreadPool(3);
executor.submit(() -> {
    // Task to be executed
});
executor.shutdown();
```

Callable and Future:

- While Runnable tasks do not return a result, the Callable interface allows tasks to return a value or throw an exception.
- The Future interface is used to represent the result of a Callable task, allowing you to retrieve the result once the task is complete.

```
Callable<Integer> task = () -> {
    return 10 + 20;
};

Future<Integer> future = executor.submit(task);
Integer result = future.get();  // Retrieves the result
```

Real-Life Example

Think of an executor as a task manager who assigns tasks to workers (threads) in a factory. Workers can either work on tasks that don't require feedback (Runnable) or tasks where the result is needed (Callable). The Future is like a promise that a result will be available when the worker finishes.

Quick Notes

Using the Executor Framework helps manage thread pools and handle multiple tasks more efficiently, improving performance in multithreaded applications.

Summary

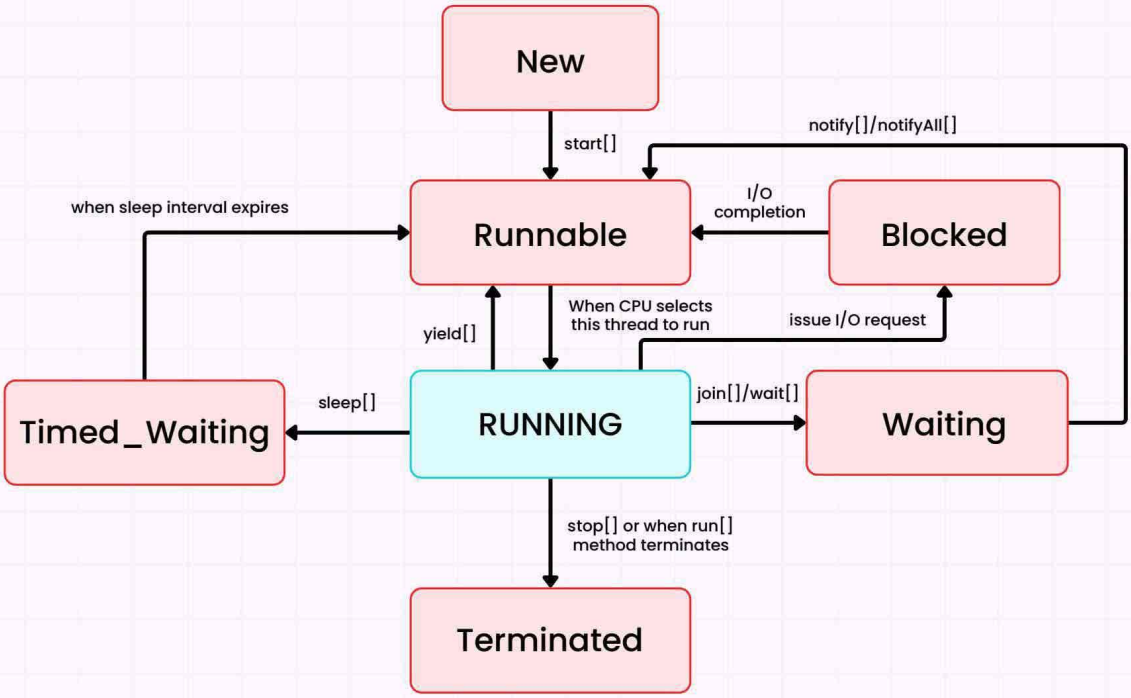
- Multithreading and concurrency in Java allow applications to perform multiple tasks simultaneously, improving performance and responsiveness.
- By mastering these concepts, you'll be well-prepared to handle multithreading challenges in real-world projects.

Interview Q & A

1. What are the main stages of the thread lifecycle in Java?

The main stages of the Java thread lifecycle are:

- **New:** Thread is created but not started.
- **Runnable:** Thread is ready to run but waiting for CPU.
- **Blocked:** Thread is blocked and waiting to acquire a lock.
- **Waiting:** Thread is waiting indefinitely until notified.
- **Timed Waiting:** Thread waits for a specified time (e.g., sleep()).
- **Terminated:** Thread has finished executing.



2. How do you create and start a thread in Java?

You can create a thread by:

1. Extending the Thread class.
2. Implementing the Runnable interface.

```java
class MyThread extends Thread {
    public void run() {
        System.out.println("Thread running...");
    }
}

MyThread thread = new MyThread();
thread.start();
```

Tip:

Always use start() to begin the thread instead of run().

3. What is the difference between the start() and run() methods in Java threading?

- start(): Creates a new thread and calls run() in that thread.
- run(): Executes code in the current thread, not creating a new one.

4. What is the synchronized keyword used for?

The synchronized keyword ensures that only one thread accesses a critical section at a time, preventing race conditions.

```java
public synchronized void increment() {
    // Critical section
}
```

Tip

Use synchronized when modifying shared data across multiple threads.

5. Explain the difference between a ReentrantLock and the synchronized block.

- ReentrantLock: Provides more control, such as timed locking and interruptible locking.
- synchronized block: Simpler but with fewer features.

6. What are deadlocks, and how can they be avoided in Java?

A deadlock occurs when two threads wait indefinitely for each other's locks.

To avoid it:

- Use a consistent order when acquiring multiple locks.
- Use tryLock with timeout.

7. What is the role of the ExecutorService in the Java Executor Framework?

ExecutorService provides a way to manage and control threads in a pool, allowing tasks to be submitted and managed effectively.

8. What is the difference between Runnable and Callable?

- Runnable: Returns no result and cannot throw checked exceptions.
- Callable: Returns a result and can throw checked exceptions.

9. How can you submit a task to an executor in Java?

You can submit tasks to an ExecutorService using submit() or execute().

```
ExecutorService executor = Executors.newFixedThreadPool(2);
executor.submit(() -> System.out.println("Task submitted"));
```

10. What is the Future interface used for in Java?

- Future represents the result of an asynchronous task.
- It provides methods to check if the task is completed and to retrieve the result.

11. Explain how the sleep() and join() methods work in thread management.

- sleep(): Pauses the current thread for a specified duration.
- join(): Waits for another thread to complete before continuing.

12. How do you ensure that a thread finishes before continuing with the main thread?

Use the join() method to ensure a thread completes before the main thread proceeds.

```
thread.join(); // Waits for 'thread' to finish
```

13. What is the role of the ThreadPoolExecutor in managing multiple threads?

ThreadPoolExecutor provides flexible control over thread pool behavior, such as the number of threads, idle timeout, and queue capacity.

14. How does Java handle thread priorities?

Java allows threads to have priorities (MIN_PRIORITY to MAX_PRIORITY), but it may not significantly affect scheduling on all platforms.

15. Explain the role of the wait() and notify() methods in inter-thread communication.

wait() releases the lock and waits, while notify() wakes up a waiting thread. They are used for coordinating threads' execution.

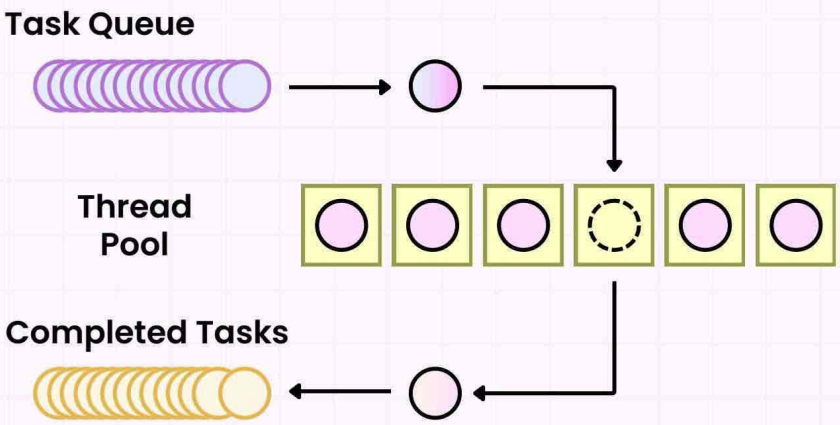
```
synchronized(obj) {
    obj.wait();
    obj.notify();
}
```

16. What are the differences between a FixedThreadPool and a CachedThreadPool in Java?

- FixedThreadPool: Fixed number of threads.
- CachedThreadPool: Creates new threads as needed and reuses idle threads.

17. How can you create a custom thread pool in Java?

Use ThreadPoolExecutor to create a custom thread pool with specific parameters.




```
ThreadPoolExecutor executor = new ThreadPoolExecutor(2, 4, 30, TimeUnit.SECONDS, new LinkedBlockingQueue<>());
```

18. What is the purpose of the shutdown() method in ExecutorService?

shutdown() initiates an orderly shutdown, allowing currently running tasks to finish but not accepting new tasks.

```
synchronized(obj) {
    obj.wait();
    obj.notify();
}
```

19. Can you forcefully stop a thread in Java?

Stopping a thread abruptly is discouraged and deprecated. Instead, use interruption or flags to stop a thread safely.

20. How does the awaitTermination() method work with thread pools?

awaitTermination() waits for the thread pool to terminate within a specified timeout period.

21. What is a thread-safe collection in Java?

A thread-safe collection is a collection designed to support concurrent access without needing additional synchronization (e.g., ConcurrentHashMap).

22. How does Java handle thread interruptions?

- Java handles interruptions using the interrupt() method.
- Threads can check their interrupted status using isInterrupted() or interrupted().

23. Explain the concept of thread starvation and how it can be avoided.

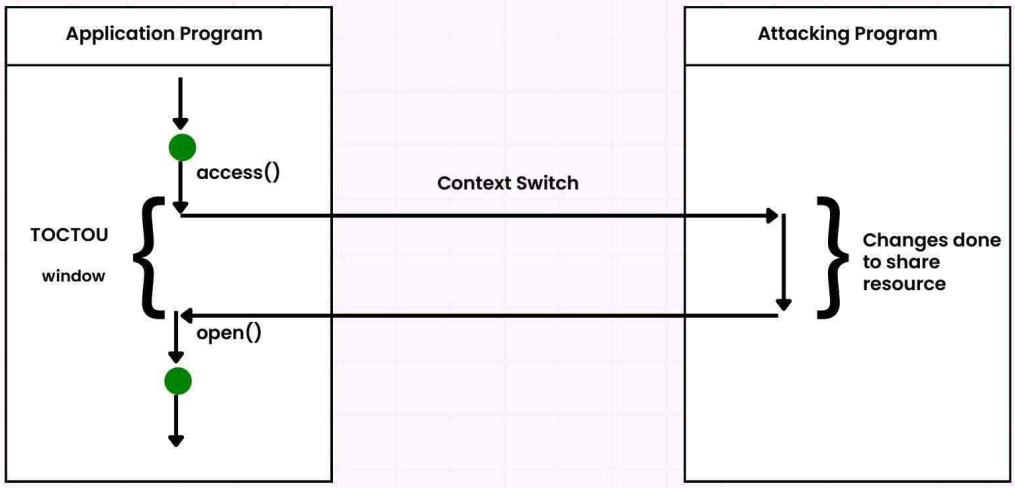
- Thread starvation occurs when low-priority threads are deprived of CPU time due to high-priority threads.
- Using fair locks and avoiding excessive priorities can help avoid starvation.

24. What is the ForkJoinPool in Java, and when should it be used?

- ForkJoinPool is used for parallelism, breaking large tasks into smaller sub-tasks that can run concurrently.
- Ideal for divide-and-conquer algorithms.

25. How can you measure the performance of a multithreaded application?

- Use synchronization.
- Use thread-safe collections.
- Use volatile or Atomic variables.



27. What is the purpose of the CyclicBarrier class in Java?

CyclicBarrier allows multiple threads to wait for each other to reach a common barrier point, useful in scenarios where threads must work in phases.

28. How does Java's CountDownLatch work?

CountDownLatch allows one or more threads to wait until a set of operations in other threads completes.

29. How can you ensure that a task completes within a certain time in Java?

Use Future.get(timeout, TimeUnit) to attempt retrieving a result within a given timeframe, or cancel the task if it exceeds the timeout.

30. What is the difference between cooperative and preemptive multitasking in Java?

- Cooperative multitasking: Threads voluntarily yield control.
- Preemptive multitasking: The OS forcibly switches threads based on priority or time slices.

31. How does the invokeAll() method work in the ExecutorService?

invokeAll() submits a collection of Callable tasks and waits for all of them to complete, returning a list of Future objects.

32. How would you handle exceptions in multithreaded code?

Handle exceptions in a try-catch block within the thread, and use UncaughtExceptionHandler for unhandled exceptions.

33. What is the difference between a user thread and a daemon thread?

- User thread: Keeps the JVM running.
- Daemon thread: Runs in the background and terminates when all user threads finish.

```
Thread daemonThread = new Thread(task);
daemonThread.setDaemon(true);
```

34. How does the ScheduledExecutorService handle timed tasks?

ScheduledExecutorService schedules tasks to run at a fixed rate or with a delay.

35. How does the ThreadLocal class work?

ThreadLocal provides a separate instance of a variable for each thread, ensuring thread isolation.

```
ThreadLocal<Integer> threadLocal = ThreadLocal.withInitial(() -> 1);
```