Threads overview

Definition: A thread is a lightweight unit of execution within a process, allowing tasks to run concurrently within the same application.

Thread Lifecycle:

A thread in Java goes through several states during its lifecycle:

1. **New**: A thread is in the new state when it has been created but not yet started. The thread is not yet running.
    - Example: `Thread thread = new Thread();`
2. **Runnable**: When the `start()` method is called, the thread moves to the runnable state. It is ready to run and is waiting for CPU time to be scheduled.
    - Example: `thread.start();`
3. **Blocked/Waiting**: A thread enters this state when it is waiting for some resource to become available or waiting for another thread to perform a particular action.
    - Example: Waiting for I/O operations, or when `wait()` is called.
4. **Timed Waiting**: This state occurs when a thread is waiting for a specified period. After the time expires, it becomes runnable again.
    - Example: `Thread.sleep(1000);` or `wait(timeout);`
5. **Terminated**: A thread reaches this state when it has finished executing or has been explicitly stopped.
    - Example: When the `run()` method completes its execution, or `thread.interrupt();` is called.

Thread Pools

Definition: A thread pool is a managed collection of reusable threads that handle multiple tasks concurrently. It optimizes performance by limiting the number of active threads, reducing the overhead of thread creation, and ensuring efficient resource use.

Advantages: Thread pools offer better resource management, reusability, and improved efficiency, making them ideal for handling high volumes of short-lived tasks.

Synchronization

Definition: Synchronization is a mechanism used to control access to shared resources by multiple threads. It ensures that only one thread can access a critical section of code at a time, preventing data inconsistency and race conditions.

Locks and Monitors: Java provides synchronization mechanisms like intrinsic locks (using the `synchronized` keyword) and monitors to manage thread access to critical sections. These prevent multiple threads from entering a critical section simultaneously.

Synchronization: Ensures thread-safe access to shared resources, preventing errors like race conditions, but requires careful handling to avoid issues like deadlock.