**MULTITHREADING & SYNCHRONIZATION**

**MULTITHREADING:**

* To increase performance.
* Multiple threads work simultaneously to complete a work and therefore using this will increase performance.
* Using **Thread class,** we make threads or by **implementing Runnable interface**.

**STEPS:**

* Thread t = new Thread (); // new state or born state.
* t.start () // runnable state.
* When the thread is executing run () method thread will be in running state.

**Thread: Light weight piece of code.**

**Start () method will be** **calling run () method**. Else, these all will be called after creating an object.

int a = 10;

Thread t1 is trying to update a =20;

Thread t2 is trying to update a =30;

System.out.println(a) -> So here I may get 10 or 20 or 30 as well because we don't know which thread will be executed because all of them are using different things internally that we don't know.

**To Overcome this**, we use synchronization or sequentially or 1 by 1.

int a = 10;

Thread t1 is trying to update a = 20;

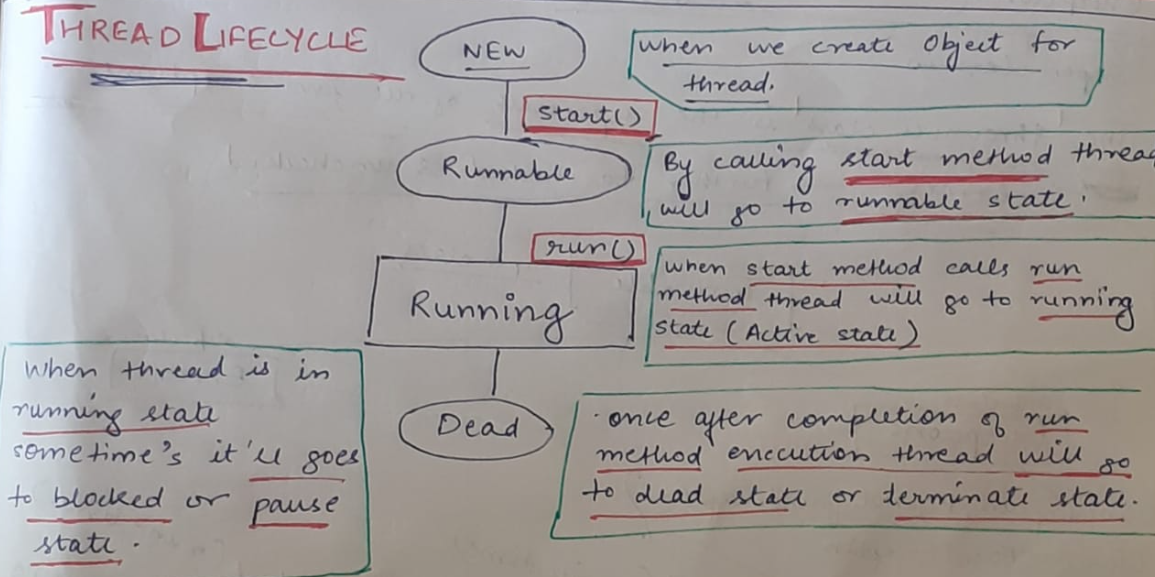
System.out.println(a); // 20

Thread t2 is trying to update a = 30;

System.out.println(a); //30

**That is, we use (1 by 1) instead of (all of them at once)**.

**THREAD LIFECYCLE:**



Synchronization will execute 1 by 1 thread execution and won't allow all the thread execution in simultaneously manner. It's a locking method.

**String buffer is synchronized. Due to which it is slow and gives less performance.**

**String builder is not synchronized; therefore, it is fast and gives more performance.**

Synchronization = Locking mechanism.

We have two types of level lock:

* **Block level lock**: It is only for block, which is (customizable) and the Specific number of lines, and not all the method lines.
* **Method level lock**: Whole method.

So, we can say more the locking, more synchronization, more thread safer and lesser performance.

* **Thread safe in Java is equal to synchronization**.

**CREATION OF THREAD:**

**By extending Thread class:**

* Create a class that extends the Thread class.
* Override the run () method to define the code that the thread will execute.
* Instantiate an object of this subclass and call its start () method to begin execution of the thread.
* class MyThread extends Thread {
* public void run () {
* // Thread execution logic
* }
* public static void main (String [] args) {
* MyThread thread = new MyThread();
* thread.start(); // Starts the thread
* }
* }

**By implementing Runnable interface:**

* Create a class that implements the Runnable interface.
* Implement the run () method with the thread's code.
* Instantiate a Thread object, passing the Runnable object as an argument to the Thread constructor.
* Call the start () method on the Thread object to start execution
* class MyRunnable implements Runnable {
* public void run () {
* // Thread execution logic
* }
* public static void main (String [] args) {
* MyRunnable myRunnable = new MyRunnable();
* Thread thread = new Thread(myRunnable);
* thread.start(); // Starts the thread
* }
* }

**CALLABLE INTERFACE:**

* There are two ways of creating threads – one by extending the Thread class and other by creating a thread with a Runnable. However, one feature lacking in Runnable is that we cannot make a thread return result when it terminates, i.e. when run () completes. For supporting this feature, the Callable interface is present in Java.

**WHY RUNNABLE INTERFACE IS PREFERRED:**

Runnable remains preferred for its simplicity, compatibility, and ease of integration with Java's concurrency utilities and functional programming features. It strikes a good balance between flexibility and ease of use, making it a suitable choice for many multithreading applications in Java.

**DEADLOCK:**

Deadlock is a situation in concurrent programming where two or more threads are blocked forever, each waiting on the other to release a resource that they need in order to proceed. In other words, it's a state where two or more threads are stuck in a circular wait for resources that are held by each other, preventing any of them from making progress.

So, let's say thread 2 is doing some tasks on object 2 and similarly thread 1 on object 1.

So, whenever thread 2 is working on object 2 at the same time, it is trying to acquire a lock on object 1 as well. And similarly thread 1 is working on object 1 and also trying to lock on object 2 as well. And therefore, both initially wanted to or supposed to work, they won't be able to complete their respective tasks because they deviated to another object instead of their own and in the end the task won’t be completed.

**EXECUTOR FRAMEWORK:**

* Java provides the ExecutorService framework to manage and execute threads.
* This approach abstracts thread creation and management details and provides more control over thread execution.
* import java.util.concurrent.ExecutorService;
* import java.util.concurrent.Executors;
* public class Main {
* public static void main (String [] args) {
* ExecutorService executor = Executors.newFixedThreadPool(10); // Example: Fixed pool of 10 threads
* executor.submit(() -> {
* // Thread execution logic
* });
* executor.shutdown(); // Shutdown the executor when done
* }
* }

**FixedThreadPool**:

* This type of thread pool maintains a fixed number of threads. Once created, the pool has a fixed number of threads that remain active until the pool is shut down.

**SingleThreadExecutor**:

* This thread pool consists of a single thread. Tasks submitted to this executor are executed sequentially by the single thread.

**ScheduledThreadPool**:

* This type of thread pool is suitable for scheduling tasks to run after a delay or periodically.

**CachedThreadPool**:

* A cached thread pool can dynamically create threads as needed and reuse idle threads when they become available.