MULTI-TASKING

Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to utilize the CPU. Multitasking can be achieved in two ways:

➔ Process-Based Multi-tasking
➔ Thread-Based Multi-tasking

Process-Based Multi-tasking:

- Each process has an address in memory. In other words, each process allocates a separate memory area.
- A process is heavyweight.
- Cost of communication between the process is high.

Thread-Based Multi-tasking:

- Threads share the same address space.
- A thread is lightweight.
- Cost of communication between the thread is low.

MULTI-THREADING

The process of executing multiple tasks at a time is known as Multi-Threading.

Advantages of Multi-Threading: -

1) It doesn't block the user because threads are independent and you can perform multiple operations at the same time.

2) You **can perform many operations together, so it saves time**.

3) Threads are **independent**, so it doesn't affect other threads if an exception occurs in a single thread.

Thread: -

A thread is a light weight sub-process, the smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking.

Java-Thread-class: -

Java provides **Thread class** to achieve thread programming. Thread class provides **constructors** and **methods** to create and perform operations on a thread. Thread class extends **Object** class and implements **Runnable** interface.

Thread Can be created in Two Ways: -

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

Thread Creation By Extending the Thread Class: -

We create a class that extends the **java.lang.Thread** class. This class overrides the run () method available in the Thread class. A thread begins its life inside run () method. We create an object of our new class and call start () method to start the execution of a thread. Start () invokes the run () method on the Thread object.

Thread creation by implementing the Runnable Interface: -

We create a new class which implements **java.lang.Runnable** interface and override run() method. Then we instantiate a Thread object and call start() method on this object.

Thread Class vs Runnable Interface: -

- ➔ If we extend the Thread class, our class cannot extend any other class because Java doesn't support multiple inheritance. But, if we implement the Runnable interface, our class can still extend other base classes.
- ➔ We can achieve basic functionality of a thread by extending Thread class because it provides some inbuilt methods like yield(), interrupt() etc. that are not available in Runnable interface.
- ➔ Using runnable will give you an object that can be shared amongst multiple threads.

Synchronization: -

Multi-threaded programs may often come to a situation where multiple threads try to access the same resources and finally produce erroneous and unforeseen results.

Java Synchronization is used to make sure by some synchronization method that only one thread can access the resource at a given point in time.

Java provides a way of creating threads and synchronizing their tasks using synchronized blocks.

A synchronized block in Java is synchronized on some object. All synchronized blocks synchronize on the same object and can only have one thread executed inside them at a time. All

other threads attempting to enter the synchronized block are blocked until the thread inside the synchronized block exits the block.

Note: - *Synchronized blocks in java are marked with synchronized keyword.*

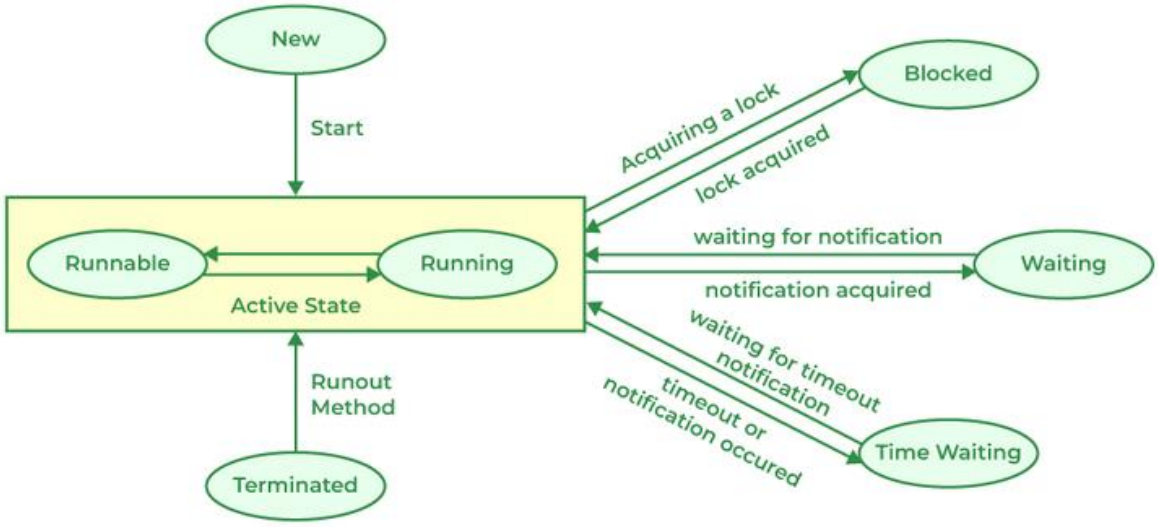
Synchronization is implemented in Java with a concept called monitors or locks. Only one thread can own a monitor at a given time. When a thread acquires a lock, it is said to have entered the monitor. All other threads attempting to enter the locked monitor will be suspended until the first thread exits the monitor.

DEADLOCK IN JAVA :

Deadlock can occur in a situation when a thread is waiting for an object lock, that is acquired by another thread and second thread is waiting for an object lock that is acquired by first thread. Since, both threads are waiting for each other to release the lock, the condition is called deadlock.

Thread Life Cycle: -

1. NEW
2. ACTIVE
3. BLOCKED
4. WAITING
5. TIMED WAITING
6. TERMINATED



New State: -

Whenever a new thread is created, it is always in the new state. For a thread in the new state, the code has not been run yet and thus has not begun its execution.

Active State: -

When a thread invokes the start() method, it moves from the new state to the active state. The active state contains two states within it: one is **runnable**, and the other is **running**.

Runnable State: -

A thread, that is ready to run is then moved to the runnable state. In the runnable state, the thread may be running or may be ready to run at any given instant of time. It is the duty of the thread scheduler to provide the thread time to run, i.e., moving the thread the running state.

Running State: -

When the thread gets the CPU, it moves from the runnable to the running state. Generally, the most common change in the state of a thread is from runnable to running and again back to runnable.

Blocked State: -

The thread will be in blocked state when it is trying to acquire a lock but currently the lock is acquired by the other thread. The thread will move from the blocked state to runnable state when it acquires the lock.

Waiting State: -

The thread will be in waiting state when it calls wait() method or join() method. It will move to the runnable state when other thread will notify or that thread will be terminated.

Timed Waiting State: -

A thread lies in a timed waiting state when it calls a method with a time-out parameter. A thread lies in this state until the timeout is completed or until a notification is received. For example, when a thread calls sleep or a conditional wait, it is moved to a timed waiting state.

Terminated State: -

when the code of the thread has been entirely executed by the program and occurred some unusual erroneous event, like a segmentation fault or an unhandled exception.