Advanced Java Session 1 and 2

Threads

1) Threads vs Proecesses
2) Thread Lifecycle
3) Creating and Managing Threads (Thread class , Runnable & Callable)
4) Synchrinozation
5) Inter Thread Communication
------ Session 2-----------
6) Thread pools and Executors
7) Handling Interrupts and Exceptions
8) Pitfalls with threads
    a) Deadlock
    b) Race Condition
    c) Dirty Read
9) Real life usage of Threads Example (producer – consumer)
10) Additional Topics
    a) Thread Schedulling

1) Threads and processes

Multithreading
Multithreading is a programming concept in which the application can create a small unit of tasks to execute in parallel.

Multithreading vs Multiprocessing

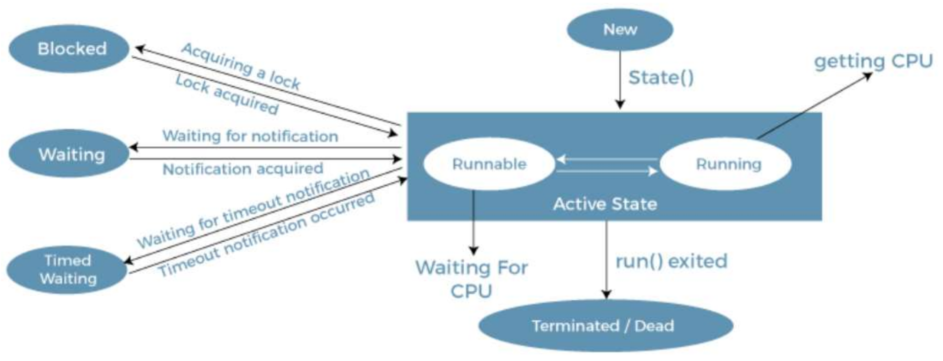
When we talk about multithreading, we don't care if the machine has a 2-core processor or a 16-core processor. Our work is to create a multithreaded application and let the OS handle the allocation and execution part. In short, multithreading has nothing to do with multiprocessing.

	Process	**Thread**
1.	Process means any program is in execution.	Thread means a segment of a process.
2.	The process takes more time to terminate.	The thread takes less time to terminate.
3.	It takes more time for creation.	It takes less time for creation.
4.	It also takes more time for context switching.	It takes less time for context switching.
5.	The process is less efficient in terms of communication.	Thread is more efficient in terms of communication.
6.	Multiprogramming holds the concepts of multi-process.	We don't need multi programs in action for multiple threads because a single process consists of multiple threads.
7.	The process is isolated.	Threads share memory.
8.	The process is called the heavyweight process.	A Thread is lightweight as each thread in a process shares code, data, and resources.

	Process	Thread
9.	Process switching uses an interface in an operating system.	Thread switching does not require calling an operating system and causes an interrupt to the kernel.
10.	If one process is blocked then it will not affect the execution of other processes	If a user-level thread is blocked, then all other user-level threads are blocked.
11.	The process has its own Process Control Block, Stack, and Address Space.	Thread has Parents' PCB, its own Thread Control Block, and Stack and common Address space.
12.	Changes to the parent process do not affect child processes.	Since all threads of the same process share address space and other resources so any changes to the main thread may affect the behavior of the other threads of the process.
13.	A system call is involved in it.	No system call is involved, it is created using APIs.
14.	The process does not share data with each other.	Threads share data with each other.

Thread Lifecycle

1. New
2. Active
   1. Runnable
   2. Running
3. Blocked / Waiting
4. Timed Waiting
5. Terminated



Life Cycle of a Thread

New: 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: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: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: 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 or Waiting: Whenever a thread is inactive for a span of time (not permanently) then, either the thread is in the blocked state or is in the waiting state.

Timed Waiting: Sometimes, waiting for leads to starvation. For example, a thread (its name is A) has entered the critical section of a code and is not willing to leave that critical section. In such a scenario, another thread (its name is B) has to wait forever, which leads to starvation. To avoid such scenario, a timed waiting state is given to thread B. Thus, thread lies in the waiting state for a specific span of time, and not forever. A real example of timed waiting is when we invoke the sleep() method on a specific thread. The sleep() method puts the thread in the timed wait state. After the time runs out, the thread wakes up and start its execution from when it has left earlier.

Terminated: A thread reaches the termination state because of the following reasons:

- When a thread has finished its job, then it exists or terminates normally.
- Abnormal termination**:**It occurs when some unusual events such as an unhandled exception or segmentation fault.

A terminated thread means the thread is no more in the system. In other words, the thread is dead, and there is no way one can respawn (active after kill) the dead thread.

Creating and Managing Threads

We can create Threads by either implementing Runnable interface or by extending Thread Class.

Note : Similar to Runnable we can also use Callable interface (will be discussed in Thread pool)

Thread class:

Thread class provide constructors and methods to create and perform operations on a thread.Thread class extends Object class and implements Runnable interface.

Commonly used Constructors of Thread class:

- Thread()
- Thread(String name)
- Thread(Runnable r)
- Thread(Runnable r,String name)

Commonly used methods of Thread class:

1. **public void run():** is used to perform action for a thread.
2. **public void start():** starts the execution of the thread.JVM calls the run() method on the thread.
3. **public void sleep(long miliseconds):** Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
4. **public void join():** waits for a thread to die.
5. **public void join(long miliseconds):** waits for a thread to die for the specified miliseconds.
6. **public int getPriority():** returns the priority of the thread.
7. **public int setPriority(int priority):** changes the priority of the thread.
8. **public String getName():** returns the name of the thread.
9. **public void setName(String name):** changes the name of the thread.
10. **public Thread currentThread():** returns the reference of currently executing thread.
11. **public int getId():** returns the id of the thread.
12. **public Thread.State getState():** returns the state of the thread.

13.**public boolean isAlive():** tests if the thread is alive.

14.**public void yield():** causes the currently executing thread object to temporarily pause and allow other threads to execute.

15.**public void suspend():** is used to suspend the thread(depricated).

16.**public void resume():** is used to resume the suspended thread(depricated).

17.**public void stop():** is used to stop the thread(depricated).

18.**public boolean isDaemon():** tests if the thread is a daemon thread.

19.**public void setDaemon(boolean b):** marks the thread as daemon or user thread.

20.**public void interrupt():** interrupts the thread.

21.**public boolean isInterrupted():** tests if the thread has been interrupted.

22.**public static boolean interrupted():** tests if the current thread has been interrupted.

Thread Class Example

```
class Multi extends Thread{
        public void run(){
                System.out.println("thread is running...");
        }
        public static void main(String args[]){
                Multi t1=new Multi();
                t1.start();
        }
}
```

Runnable interface:

The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. Runnable interface have only one method named run().

public void run(): is used to perform action for a thread.

Runnable Example

```
class Multi3 implements Runnable{
public void run(){
        System.out.println("thread is running...");
}

public static void main(String args[]){
        Multi3 m1=new Multi3();
        Thread t1 =new Thread(m1);   // Using the constructor Thread(Runnable r)
        t1.start();
}
}
```

Launching a Thread

thread.start() --> invokes the run() method .

Note : Callable interface has a method call () which returns an object .

Synchronization

Thread Synchronization

There are two types of thread synchronization mutual exclusive and inter-thread communication.

1.Mutual Exclusive

- 1.Synchronized method.
- 2.Synchronized block.
- 3.Static synchronization.

2.Cooperation (Inter-thread communication in java)

Lock

Synchronization is built around an internal entity known as the lock or monitor. Every object has a lock associated with it. By convention, a thread that needs consistent access to an object's fields has to acquire the object's lock before accessing them, and then release the lock when it's done with them.

Synchronized method

If you declare any method as synchronized, it is known as synchronized method.
Synchronized method is used to lock an object for any shared resource.

When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.

Synchronized Block

Synchronization in Java is the capability to control the access of multiple threads to any shared resource.

Static Synchronization (**Not Recomended)**

If you make any static method as synchronized, the lock will be on the class not on object.

InterThread Communication

1. wait()
2. notify()
3. notifyAll()

1) **wait() method**
The wait() method causes current thread to release the lock and wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.

The current thread must own this object's monitor, so it must be called from the synchronized method only otherwise it will throw exception.

Method	Description
public final void wait()throws InterruptedException	It waits until object is notified.
public final void wait(long timeout)throws InterruptedException	It waits for the specified amount of time.

2) **notify() method**
The notify() method wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation.

Syntax:

public final void notify()

3) notifyAll() method
Wakes up all threads that are waiting on this object's monitor.

Syntax:

public final void notifyAll()

Difference between wait and sleep?

Let's see the important differences between wait and sleep methods.

wait()	sleep()
The wait() method releases the lock.	The sleep() method doesn't release the lock.
It is a method of Object class	It is a method of Thread class
It is the non-static method	It is the static method
It should be notified by notify() or notifyAll() methods	After the specified amount of time, sleep is completed.