Multithreading

LifeCycle

- **new** -> when you create a thread
- **runnable** -> after invocation of start()
- **running** -> when thread scheduler actually selects the thread
- **non-runnable (blocked)** -> sleep(), wait(), suspend()
- **terminated** -> end of run() or stop()

Creation

- By extending thread class
- By implementing runnable interface

Eg1 :-

```
class Example extends Thread {
    public void run {
        syso("Thread running..");
    }

    pvsm(){
        Example e = new Example();
        e.start();
    }
}
```

Eg2 :-

```
class Example implements Runnable {
    public void run {
        syso("Thread running..");
    }

    pvsm(){
        Example e = new Example();
        Thread t = new Thread(e);
        t.start();
    }
}
```

- ***IllegalThreadStateException*** *is throw if a thread is started twice.*
- Each thread starts in a separate call stack. If we call ***run()*** method directly, the ***run()*** method goes onto the current call stack rather than at the beginning of a new call stack.
- The ***join()*** method waits for a thread to die. It causes the currently

running threads to stop executing until the thread it joins with completes its task.

- ***Daemon thread*** in java is a service provider thread that provides services to the user thread. Its life depend on the mercy of user threads i.e. when all the user threads dies, JVM terminates this thread automatically. e.g. gc, finalizer etc
- A user thread must be marked as Daemon before it starts otherwise it will throw ***IllegalThreadStateException.***
    - ***t1.setDaemon(true);***
    - ***t1.start();***

Java Thread Pool

- a group of fixed size threads are created. A thread from the thread pool is pulled out and assigned a job by the service provider. After completion of the job, thread is contained in the thread pool again.
- It saves time because there is no need to create new thread.

Eg :-

```
class WorkerThread implements Runnable {
    public void run(){
        syso("running thread" +
Thread.currentThread().getName());
    }
}

class ThreadPoolTest {
    pvsm(){
        ExecutorService e = Executors.newFixedThreadPool(5);

        for(int i=0;i<10;i++) {
            Runnable worker = new Worker();
            e.execute(worker);
        }
        e.shutdown();
    }
}
```

- ***finalize()*** method is invoked each time before the object is garbage collected. This method can be used to perform cleanup processing.
- The Garbage collector of JVM collects only those objects that are created by new keyword. So if you there are any objects created without new, finalize method can perform the cleanup.
- The ***gc()*** method is used to invoke the garbage collector to perform cleanup processing.
- ***Synchronization*** is the capability *to control the access of multiple threads to any shared resource*.

- Process Synchronization
- Thread Synchronization

- Thread Synchonization
  - Mutual Exclusive
    - Synchonized Method
    - Synchonized Block
    - Static Synchonization
  - Inter-thread communication
- ***Synchronization*** is built around an internal entity known as the lock or monitor. Every object has an lock associated with it. 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.
- Scope of synchronized block is smaller than the method.
- **Static Synchronization**
  - Suppose there are two objects of a shared class(e.g. Table) named ***obj1*** and ***obj2*** and ***t1*** & ***t2*** are threads operating on ***obj1*** whereas ***t3*** & ***t4*** are operating on ***obj2***. In case of ***synchronized method and block*** there cannot be interference between ***t1*** and ***t2*** or ***t3*** and ***t4*** because ***t1*** and ***t2*** both refers to a common object that have a single lock.But there can be interference between ***t1*** and ***t3*** or ***t2*** and ***t4*** because ***t1*** acquires another lock and ***t3*** acquires another lock.I want no interference between ***t1*** and ***t3*** or ***t2*** and ***t4***.Static synchronization solves this problem. If you make any static method as synchronized, the lock will be on the class not on object.

DeadLock

It 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.

Eg:-

```
class DeadlockTest {
    pvsm() {
        final String str1 = “Resource1”;
        final String str2= “Resource2”;

        Thread t1 = new Thread() {
            public void run() {
                synchronized(resource1) {
                    syso(“some task);
                    Thread.sleep(1000);

                synchronized(resource2) {
```

```
                        syso("some task");
                    }
                }
            }}

            Thread t2 = new Thread() {
                public void run() {
                    synchronized(resource2) {
                        syso("some task);
                        Thread.sleep(1000);

                    synchronized(resource1) {
                        syso("some task");
                    }
                }
            }}

        t1.start();
        t2.start();
    }
}
```

- ***wait()***, ***notify()*** and ***notifyAll()*** methods are defined in Object class because they are related to lock and object has a lock.
- ***wait()*** method releases the lock whereas ***sleep()*** doesn't.