13. Multithreading

1) Introduction to Threads

Answer: A **thread** in Java is a lightweight subprocess, a small unit of a process. It is the smallest part of a program that can be executed independently. Threads enable a program to perform multiple tasks simultaneously, making it more efficient, responsive, and faster, especially in applications like games, multimedia, or server-side programs.

2) Creating Threads by Extending Thread Class or Implementing Runnable Interface

Answer:

When you extend the `Thread` class, you override its `run()` method to define the task the thread will perform. This approach directly associates the task with the thread.

Steps:

1. Create a class that extends the Thread class.
2. Override the run() method to define the task.
3. Create an instance of the thread class and call start() to run the thread.

Example:

```
class MyThread extends Thread {
    public void run() {
        for (int i = 1; i <= 5; i++) {
            System.out.println("Thread running: " + i);
            try {
                Thread.sleep(500); // Pause for 500ms
            } catch (InterruptedException e) {
                System.out.println("Thread interrupted");
            }
        }
    }
}

public class ThreadExample {
    public static void main(String[] args) {
        MyThread t1 = new MyThread(); // Create thread instance
        MyThread t2 = new MyThread();

        t1.start(); // Start the thread
```

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

Output (Interleaved, as threads run concurrently):

Thread running: 1
Thread running: 1
Thread running: 2
Thread running: 2
Thread running: 3
Thread running: 3
Thread running: 4
Thread running: 4
Thread running: 5
Thread running: 5

3) Thread Life Cycle

Answer:

Life Cycle of a Thread

There are multiple states of the thread in a lifecycle as mentioned below:

1. **New Thread:** When a new thread is created, it is in the new state. The thread has not yet started to run when the thread is in this state. When a thread lies in the new state, its code is yet to be run and hasn't started to execute.

2. **Runnable State:** A thread that is ready to run is moved to a runnable state. In this state, a thread might actually be running or it might be ready to run at any instant of time. It is the responsibility of the thread scheduler to give the thread, time to run.
   A multi-threaded program allocates a fixed amount of time to each individual thread. Each and every thread runs for a short while and then pauses and relinquishes the CPU to another thread so that other threads can get a chance to run. When this happens, all such threads that are ready to run, waiting for the CPU and the currently running thread lie in a runnable state.

3. **Blocked:** 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.

4. **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.

5. **Timed Waiting:** 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.

6. **Terminated State:** A thread terminates because of either of the following reasons:

    - Because it exits normally. This happens when the code of the thread has been entirely executed by the program.
    - Because there occurred some unusual erroneous event, like a segmentation fault or an unhandled exception.

4) Synchronization and Inter-thread Communication
Answer:

In a multithreaded environment, multiple threads often share resources. This can lead to **data inconsistency** if multiple threads try to access and modify shared resources simultaneously. **Synchronization** and **Inter-thread Communication** help solve this problem by coordinating thread access to shared resources.

Synchronization is the process of controlling thread access to shared resources to prevent **race conditions** (i.e., multiple threads modifying data simultaneously in an inconsistent way).

==========
==========
==========
==========