

A thread can be in one of the five states

•**New** - new state if you create an instance of Thread class but before the invocation of start() method

•**Runnable** - runnable state after invocation of start() method, but the thread scheduler has not selected it to be the running thread

•**Running** - running state if the thread scheduler has selected it

•**Non-Runnable (Blocked)** - state when the thread is still alive, but is currently not eligible to run

•**Terminated** - terminated or dead state when its run() method exits

**Ways of creating threads?**

There are two ways to create a thread:

•By **extending** Thread class

Provide constructors and methods to create and perform operations on a thread

Extends Object class and implements Runnable interface

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();**

}

}

•By **implementing** 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**()

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);**

**t1.start();**

}

}

**Scheduling of Threads?**

Two ways of scheduling are pre-emptive scheduling and time slicing

•Under pre-emptive scheduling, the highest priority task executes until it enters the waiting or dead states or a higher priority task comes into existence

•Under time slicing, a task executes for a predefined slice of time and then re-enters the pool of ready tasks. The scheduler then determines which task should execute next, based on priority and other factors.

## 10 Points-To-Remember About Synchronization In Java :

**1) You can use synchronized keyword only with methods but not with variables, constructors, static initializer and instance initializers.**

class Shared

{

synchronized int i; //compile time error, can't use synchronized keyword with variables

synchronized public Shared()

{

//compile time error, constructors can not be synchronized

}

synchronized static

{

//Compile time error, Static initializer can not be synchronized

}

synchronized

{

//Compile time error, Instance initializer can not be synchronized

}

}

**2) Constructors, Static initializer and instance initializer can’t be declared with synchronized keyword, but they can contain synchronized blocks.**

class Shared

{

public Shared()

{

synchronized (this)

{

//synchronized block inside a constructor

}

}

static

{

synchronized (Shared.class)

{

//synchronized block inside a static initializer

}

}

{

synchronized (this)

{

//synchronized block inside a instance initializer

}

}

}

**3) Both static and non-static methods can use synchronized keyword. For static methods, thread need class level lock and for non-static methods, thread need object level lock.**

class Shared

{

synchronized static void staticMethod()

{

//static synchronized method

}

synchronized void NonStaticMethod()

{

//Non-static Synchronized method

}

}

**4) It is possible that both static synchronized and non-static synchronized methods can run simultaneously. Because, static methods need class level lock and non-static methods need object level lock.**

**5) A method can contain any number of synchronized blocks. This is like synchronizing multiple parts of a method.**

**class Shared**

{

static void staticMethod()

{

synchronized (Shared.class)

{

//static synchronized block - 1

}

synchronized (Shared.class)

{

//static synchronized block - 2

}

}

void NonStaticMethod()

{

synchronized (this)

{

//Non-static Synchronized block - 1

}

synchronized (this)

{

//Non-static Synchronized block - 2

}

}

}

**6) Synchronization blocks can be nested.**

synchronized (this)

{

synchronized (this)

{

//Nested synchronized blocks

}

}

**7) Lock acquired by the thread before executing a synchronized method or block must be released after the completion of execution, no matter whether execution is completed normally or abnormally (due to exceptions).**

**8) Synchronization in java is Re-entrant in nature. A thread cannot acquire a lock that is owned by another thread. But, a thread can acquire a lock that it already owns. That means if a synchronized method gives a call to another synchronized method which needs same lock, then currently executing thread can directly enter into that method or block without acquiring the lock.**

**9) synchronized method or block is very slow. They decrease the performance of an application. So, special care need to be taken while using synchronization. Use synchronization only when you needed it the most.**

**10) Use synchronized blocks instead of synchronized methods. Because, synchronizing some part of a method improves the performance than synchronizing the whole method.**