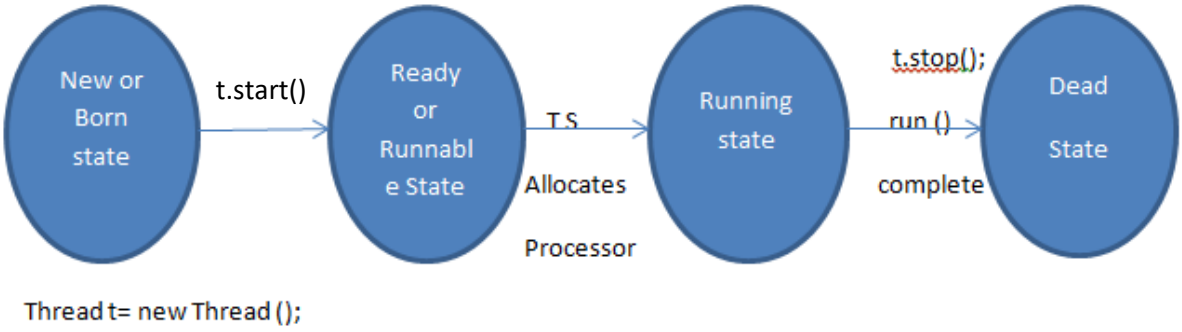
Life Cycle of Thread



Thread Life Cycle

There are different types of thread state are as follows as-

1. New or Born State-
   The thread is in new state if you create an instance of Thread class but before the invocation of start () method.

2. Runnable state-

The thread is in runnable state after invocation of start () method, but the thread scheduler has not selected it to be the running thread. After starting a Thread we are not allowed to restart the same Thread once again otherwise we will get runtime exception saying "IllegalThreadStateException".

3. Running state-
   The thread is in running state if the thread scheduler has selected it.

4. Dead state-
   A thread is in terminated or dead state when its run () method exits.

5. Waiting state-
   When a thread is temporarily inactive, then it's in one of the following states: Blocked and Waiting state. Or Running thread calls join method then it will enter into waiting state (Blocking for joining).

6. Sleep state-
   If running thread calls sleep method then it will enter into sleep state. If sleeping thread got interrupted or time expire then it will enter into ready state.

7. Waiting state-
   If running thread calls wait method then it will enter into waiting state. If waiting state got notification then it will enter into another wating state.

8. Suspended state-
   If running state called suspend method then thread will enter into suspended state.

9. Resume state-
   If we call thread from resume () method then it will enter into ready state.

Synchronization in Java-

We can apply synchronization on method and block only. We cannot apply it on variables and class.

If a method or block declared as the synchronized then at a time only one Thread is allow to execute that method or block on the given object. It solves data inconsistence issue.

But the main disadvantage of synchronized keyword is it increases waiting time of the Thread and effects performance of the system.

Internally synchronization concept is implemented by using lock concept.

Why?

```
package com.synchronizaitons;

public class Account {

     private int balance=5000;

     public int getBalance() {
          return balance;
```

```
	}

	public int withdraw(int amount) {
		balance= balance-amount;
		return balance;
	}


}

package com.synchronizaitons;

public class AccountDetails implements Runnable {

	Account account = new Account();

	@Override
	public void run() {

		for (int x = 0; x < 5; x++) {

			makeWithdrawal(500);

			if (account.getBalance() <= 0) {
				System.out.println("Account is
overdrawn...");
			}
		}

	}

	private void makeWithdrawal(int amt) {

		if (account.getBalance() >= amt) {

	System.out.println(Thread.currentThread().getName() +
					"is going to withdraw=>");
		}
```

```
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                System.out.println(e.getMessage());
            }

            int bal = account.withdraw(amt);
            System.out.println(Thread.currentThread().getName()
+
                    "complete the withdrawal=>" + bal);

        }

}
```

```
package com.synchronizaitons;

public class MainTest {

    public static void main(String[] args) {

            AccountDetails accountDetails= new
AccountDetails();
            Thread thread1=new Thread(accountDetails);
            Thread thread2= new Thread(accountDetails);
            thread1.setName("ram");
            thread2.setName("soham");
            thread1.start();
            thread2.start();
    }
}
```

In this example, there are two thread which are executed randomly but I want to execute one by one thread at a time then go for synchronization.

Note- Just make the `makeWithdrawal` method as synchronized, so you will get the output like as

Output- using synchronization

```
ram is going to withdraw>>
ram complete withdraw>>4500
ram is going to withdraw>>
ram complete withdraw>>4000
soham is going to withdraw>>
soham complete withdraw>>3500
ram is going to withdraw>>
ram complete withdraw>>3000
ram is going to withdraw>>
ram complete withdraw>>2500
soham is going to withdraw>>
soham complete withdraw>>2000
ram is going to withdraw>>
ram complete withdraw>>1500
soham is going to withdraw>>
soham complete withdraw>>1000
soham is going to withdraw>>
soham complete withdraw>>500
soham is going to withdraw>>
soham complete withdraw>>0
account is overdrawn.....
```

Output- without synchronization

```
ram is going to withdraw>>
soham is going to withdraw>>
ram complete withdraw>>4000
soham complete withdraw>>4500
ram is going to withdraw>>
soham is going to withdraw>>
soham complete withdraw>>3500
ram complete withdraw>>3000
ram is going to withdraw>>
```

```
soham is going to withdraw>>
ram complete withdraw>>2000
soham complete withdraw>>2500
soham is going to withdraw>>
ram is going to withdraw>>
soham complete withdraw>>1500
soham is going to withdraw>>
ram complete withdraw>>1000
ram is going to withdraw>>
soham complete withdraw>>500
ram complete withdraw>>0
account is overdrawn.....
account is overdrawn.....
```

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.

Example- synchronized void test (){

// write code here

}

Synchronized Block-

Synchronized block can be used to perform synchronization on any specific resource of the method.

Suppose you have 50 lines of code in your method, but you want to synchronize only 5 lines, you can use synchronized block.

If you put all the codes of the method in the synchronized block, it will work same as the synchronized method.

Syntax-

```
synchronized (object reference) {
  //code block
}
```

For ex. In println method we have a synchronized block using current/this object reference.

Inter Thread Communication :

Two Threads can communicate with each other by using wait(), notify() and notifyAll() methods.

Once a Thread calls wait() method on the given object first it releases the lock of that object immediately and entered into waiting state.

Once a Thread calls notify() (or) notifyAll() methods it releases the lock of that object but may not immediately.

Except these (wait(),notify(),notifyAll()) methods there is no other place(method) where the lock release will be happen.

1. public final void wait()throws InterruptedException
2. public final native void wait(long ms)throws InterruptedException
3. public final void wait(long ms,int ns)throws InterruptedException
4. public final native void notify()
5. public final void notifyAll()

DeadLock :

If two Threads are waiting for each other forever (without end) such type of situation (infinite waiting) is called dead lock.

A long waiting of a Thread which ends at certain point is called starvation.

There are no resolution techniques for dead lock but several prevention (avoidance) techniques are possible.

Synchronized keyword is the cause for deadlock hence whenever we are using synchronized keyword we have to take special care..

Daemon Thread :

The Threads which are executing in the background are called daemon Threads.

The main objective of daemon Threads is to provide support for non-daemon Threads like main Thread.

For ex. Garbage Collector is a daemon thread which keeps running on low priority in background. Once the program runs on less memory, JVM increase the priority of Garbage Collector and clean up the memory.