UNIT V

MULTITHREADING IN JAVA

Life cycle of a Thread (Thread States)

In Java, a thread always exists in any one of the following states. These states are:

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

Explanation of Different Thread States

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.
  A program implementing multithreading acquires a fixed slice of time to each individual thread. Each and every thread runs for a short span of time and when that allocated time slice is over, the thread voluntarily gives up the CPU to the other thread, so that the other threads can also run for their slice of time. Whenever such a scenario occurs, all those

threads that are willing to run, waiting for their turn to run, lie in the runnable state. In the runnable state, there is a queue where the threads lie.

- **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.

For example, a thread (let's say its name is A) may want to print some data from the printer. However, at the same time, the other thread (let's say its name is B) is using the printer to print some data. Therefore, thread A has to wait for thread B to use the printer. Thus, thread A is in the blocked state. A thread in the blocked state is unable to perform any execution and thus never consume any cycle of the Central Processing Unit (CPU). Hence, we can say that thread A remains idle until the thread scheduler reactivates thread A, which is in the waiting or blocked state.

When the main thread invokes the join() method then, it is said that the main thread is in the waiting state. The main thread then waits for the child threads to complete their tasks. When the child threads complete their job, a notification is sent to the main thread, which again moves the thread from waiting to the active state.

If there are a lot of threads in the waiting or blocked state, then it is the duty of the thread scheduler to determine which thread to choose and which one to reject, and the chosen thread is then given the opportunity to run.

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.

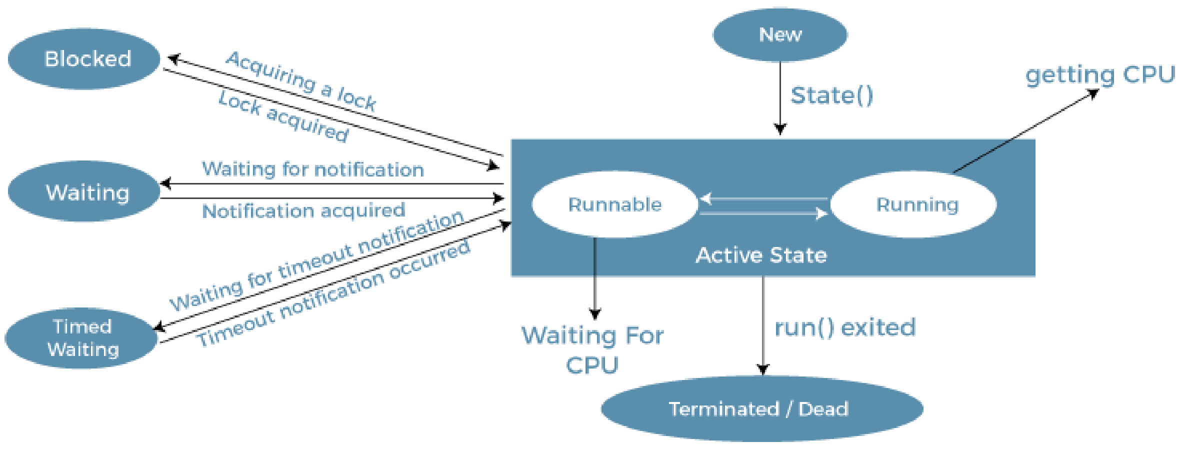
Terminated: A thread reaches the termination state because of the following reasons:

- 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.

The following diagram shows the different states involved in the life cycle of a thread.



Life Cycle of a Thread

Implementation of Thread States

In Java, one can get the current state of a thread using the **Thread.getState()** method. The **java.lang.Thread.State** class of Java provides the constants ENUM to represent the state of a thread. These constants are:

- **public static final** Thread.State NEW

It represents the first state of a thread that is the NEW state.

- **public static final** Thread.State RUNNABLE

It represents the runnable state.It means a thread is waiting in the queue to run.

- **public static final** Thread.State BLOCKED

It represents the blocked state. In this state, the thread is waiting to acquire a lock.

- **public static final** Thread.State WAITING

It represents the waiting state. A thread will go to this state when it invokes the Object.wait() method, or Thread.join() method with no timeout. A thread in the waiting state is waiting for another thread to complete its task.

- **public static final** Thread.State TIMED_WAITING

It represents the timed waiting state. The main difference between waiting and timed waiting is the time constraint. Waiting has no time constraint, whereas timed waiting has the time constraint. A thread invoking the following method reaches the timed waiting state.

  - sleep
  - join with timeout
  - wait with timeout
  - parkUntil
  - parkNanos

- **public static final Thread.State TERMINATED**

It represents the final state of a thread that is terminated or dead. A terminated thread means it has completed its execution.

Java Program for Demonstrating Thread States

The following Java program shows some of the states of a thread defined above.

FileName: ThreadState.java

// ABC class implements the interface Runnable

```
class ABC implements Runnable
{
public void run()
{

// try-catch block
try
{
```

```
// moving thread t2 to the state timed waiting
Thread.sleep(100);
}
catch (InterruptedException ie)
{
ie.printStackTrace();
}


System.out.println("The state of thread t1 while it invoked the method join() on thread t2 -
"+ ThreadState.t1.getState());

// try-catch block
try
{
Thread.sleep(200);
}
catch (InterruptedException ie)
{
ie.printStackTrace();
}
}
}

// ThreadState class implements the interface Runnable
public class ThreadState implements Runnable
{
public static Thread t1;
public static ThreadState obj;

// main method
public static void main(String argvs[])
{
// creating an object of the class ThreadState
obj = new ThreadState();
```

```
t1 = new Thread(obj);

// thread t1 is spawned
// The thread t1 is currently in the NEW state.
System.out.println("The state of thread t1 after spawning it - " + t1.getState());

// invoking the start() method on
// the thread t1
t1.start();

// thread t1 is moved to the Runnable state
System.out.println("The state of thread t1 after invoking the method start() on it -
 " + t1.getState());
}

public void run()
{
ABC myObj = new ABC();
Thread t2 = new Thread(myObj);

// thread t2 is created and is currently in the NEW state.
System.out.println("The state of thread t2 after spawning it - "+ t2.getState());
t2.start();

// thread t2 is moved to the runnable state
System.out.println("the state of thread t2 after calling the method start() on it - " + t2.getState());

// try-catch block for the smooth flow of the  program
try
{
// moving the thread t1 to the state timed waiting
Thread.sleep(200);
}
catch (InterruptedException ie)
{
```

```
ie.printStackTrace();
}

System.out.println("The state of thread t2 after invoking the method sleep() on it -
 "+ t2.getState() );

// try-catch block for the smooth flow of the  program
try
{
// waiting for thread t2 to complete its execution
t2.join();
}
catch (InterruptedException ie)
{
ie.printStackTrace();
}
System.out.println("The state of thread t2 when it has completed it's execution - " + t2.getState());
}

}
```

Output:

```
The state of thread t1 after spawning it - NEW
The state of thread t1 after invoking the method start() on it - RUNNABLE
The state of thread t2 after spawning it - NEW
the state of thread t2 after calling the method start() on it - RUNNABLE
The state of thread t1 while it invoked the method join() on thread t2 -TIMED_WAITING
The state of thread t2 after invoking the method sleep() on it - TIMED_WAITING
The state of thread t2 when it has completed it's execution - TERMINATED
```

Explanation: Whenever we spawn a new thread, that thread attains the new state. When the method start() is invoked on a thread, the thread scheduler moves that thread to the runnable state. Whenever the join() method is invoked on any thread instance, the current thread executing that statement has to wait for this thread to finish its execution, i.e., move that thread to the terminated state. Therefore, before the final print statement is printed on the console, the program invokes the method join() on thread t2, making the thread t1 wait while the thread t2 finishes its execution and thus, the thread t2 get to the terminated or dead state. Thread t1 goes to the waiting state because it is waiting for thread t2 to finish it's execution as it has invoked the method join() on thread t2.