Lifecycle and States of a Thread in Java

A thread in Java at any point of time exists in any one of the following states. A thread lies only in one of the shown states at any instant:

1. New
2. Runnable
3. Blocked
4. Waiting
5. Timed Waiting
6. Terminated

The diagram shown below represent various states of a thread at any instant of time.

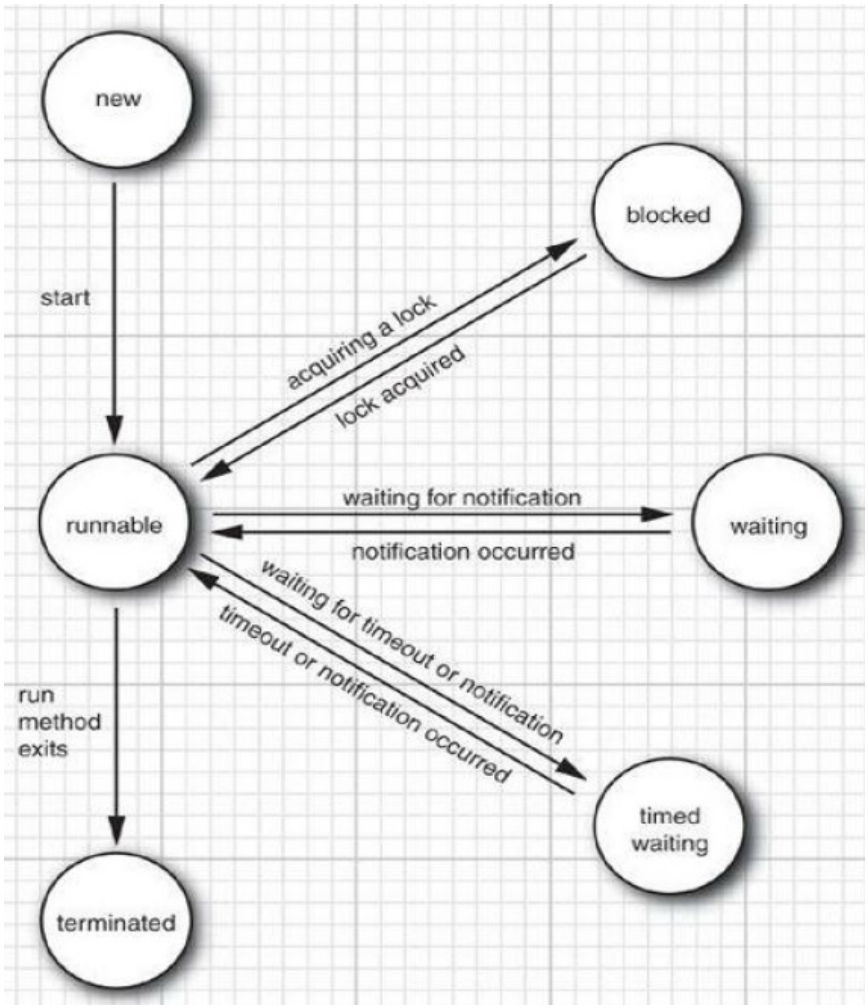

Image source: Core Java Vol 1, 9th Edition, Horstmann, Cay S. & Cornell, Gary_2013

Life Cycle of a thread

1. **New Thread:** When a new thread is created, it is in the new state. The thread has not yet started to run when thread is in this state. When a thread lies in the new state, it's 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 runnable state. In this state, a thread might actually be running or it might be ready 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 lies in runnable state.
3. **Blocked/Waiting state:**When a thread is temporarily inactive, then it's in one of the following states:
   - Blocked
   - Waiting

For example, when a thread is waiting for I/O to complete, it lies in the blocked state. It's the responsibility of the thread scheduler to reactivate and schedule a blocked/waiting thread. A thread in this state cannot continue its execution any further until it is moved to runnable state. Any thread in these states do not consume any CPU cycle.

A thread is in the blocked state when it tries to access a protected section of code that is currently locked by some other thread. When the protected section is unlocked, the schedule picks one of the thread which is blocked for that section and moves it to the runnable state. Whereas, a thread is in the waiting state when it waits for another thread on a condition. When this condition is fulfilled, the scheduler is notified and the waiting thread is moved to runnable state.

If a currently running thread is moved to blocked/waiting state, another thread in the runnable state is scheduled by the thread scheduler to run. It is the responsibility of thread scheduler to determine which thread to run.

4. **Timed Waiting:** A thread lies in 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 timed waiting state.
5. **Terminated State:** A thread terminates because of either of the following reasons:
   - Because it exists normally. This happens when the code of thread has entirely executed by the program.
   - Because there occurred some unusual erroneous event, like segmentation fault or an unhandled exception.

   A thread that lies in terminated state does no longer consumes any cycles of CPU.

Implementing Thread States in Java

In Java, to get the current state of the thread, use **Thread.getState()** method to get the current state of the thread. Java provides **java.lang.Thread.State** class that defines the ENUM constants for the state of a thread, as summary of which is given below:

1. **Constant type:** New

```
Declaration: public static final Thread.State NEW
```

   Description: Thread state for a thread which has not yet started.

2. **Constant type:** Runnable

```
Declaration: public static final Thread.State RUNNABLE
```

Description: Thread state for a runnable thread. A thread in the runnable state is executing in the Java virtual machine but it may be waiting for other resources from the operating system such as processor.

3. **Constant type:** Blocked

```
Declaration: public static final Thread.State BLOCKED
```

Description: Thread state for a thread blocked waiting for a monitor lock. A thread in the blocked state is waiting for a monitor lock to enter a synchronized block/method or reenter a synchronized block/method after calling Object.wait().

4. **Constant type:** Waiting

```
Declaration: public static final Thread.State WAITING
```

Description: Thread state for a waiting thread. Thread state for a waiting thread. A thread is in the waiting state due to calling one of the following methods:

- Object.wait with no timeout
- Thread.join with no timeout
- LockSupport.park

A thread in the waiting state is waiting for another thread to perform a particular action.

5. **Constant type:** Timed Waiting

```
Declaration: public static final Thread.State TIMED_WAITING
```

Description: Thread state for a waiting thread with a specified waiting time. A thread is in the timed waiting state due to calling one of the following methods with a specified positive waiting time:

- Thread.sleep
- Object.wait with timeout
- Thread.join with timeout
- LockSupport.parkNanos
- LockSupport.parkUntil

6. **Constant type:** Terminated

```
Declaration: public static final Thread.State TERMINATED
```

Description: Thread state for a terminated thread. The thread has completed execution.

```
// Java program to demonstrate thread states
class thread implements Runnable
```

```
{
    public void run()
    {
        // moving thread2 to timed waiting state
        try
        {
            Thread.sleep(1500);
        }
        catch (InterruptedException e)
        {
            e.printStackTrace();
        }

        try
        {
            Thread.sleep(1500);
        }
        catch (InterruptedException e)
        {
            e.printStackTrace();
        }
        System.out.println("State of thread1 while it called join() method on thread2 -"+
            Test.thread1.getState());
        try
        {
            Thread.sleep(200);
        }
        catch (InterruptedException e)
        {
            e.printStackTrace();
        }
    }
}

public class Test implements Runnable
{
    public static Thread thread1;
    public static Test obj;

    public static void main(String[] args)
    {
        obj = new Test();
        thread1 = new Thread(obj);

        // thread1 created and is currently in the NEW state.
        System.out.println("State of thread1 after creating it - " + thread1.getState());
        thread1.start();

        // thread1 moved to Runnable state
        System.out.println("State of thread1 after calling .start() method on it - " +
            thread1.getState());
    }

    public void run()
    {
        thread myThread = new thread();
        Thread thread2 = new Thread(myThread);

        // thread1 created and is currently in the NEW state.
        System.out.println("State of thread2 after creating it - "+ thread2.getState());
        thread2.start();

        // thread2 moved to Runnable state
        System.out.println("State of thread2 after calling .start() method on it - " +
            thread2.getState());

        // moving thread1 to timed waiting state
        try
        {
            //moving thread2 to timed waiting state
```

```
            Thread.sleep(200);
        }
        catch (InterruptedException e)
        {
            e.printStackTrace();
        }
        System.out.println("State of thread2 after calling .sleep() method on it - "+
            thread2.getState() );


        try
        {
            // waiting for thread2 to die
            thread2.join();
        }
        catch (InterruptedException e)
        {
            e.printStackTrace();
        }
        System.out.println("State of thread2 when it has finished it's execution - " +
            thread2.getState());
    }

}
```

Run on IDE

Output:

```
State of thread1 after creating it - NEW
State of thread1 after calling .start() method on it - RUNNABLE
State of thread2 after creating it - NEW
State of thread2 after calling .start() method on it - RUNNABLE
State of thread2 after calling .sleep() method on it - TIMED_WAITING
State of thread1 while it called join() method on thread2 -WAITING
State of thread2 when it has finished it's execution - TERMINATED
```

Explanation: When a new thread is created, the thread is in the NEW state. When .start() method is called on a thread, the thread scheduler moves it to Runnable state. Whenever join() method is called on a thread instance, the current thread executing that statement will wait for this thread to move to Terminated state. So, before the final statement is printed on the console, the program calls join() on thread2 making the thread1 wait while thread2 completes its execution and is moved to Terminated state. thread1 goes to Waiting state because it it waiting for thread2 to complete it's execution as it has called join on thread2.

References used: Oracle, Core Java Vol 1, 9th Edition, Horstmann, Cay S. & Cornell, Gary_2013

This article is contributed by **Mayank Kumar**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.