Java Threads

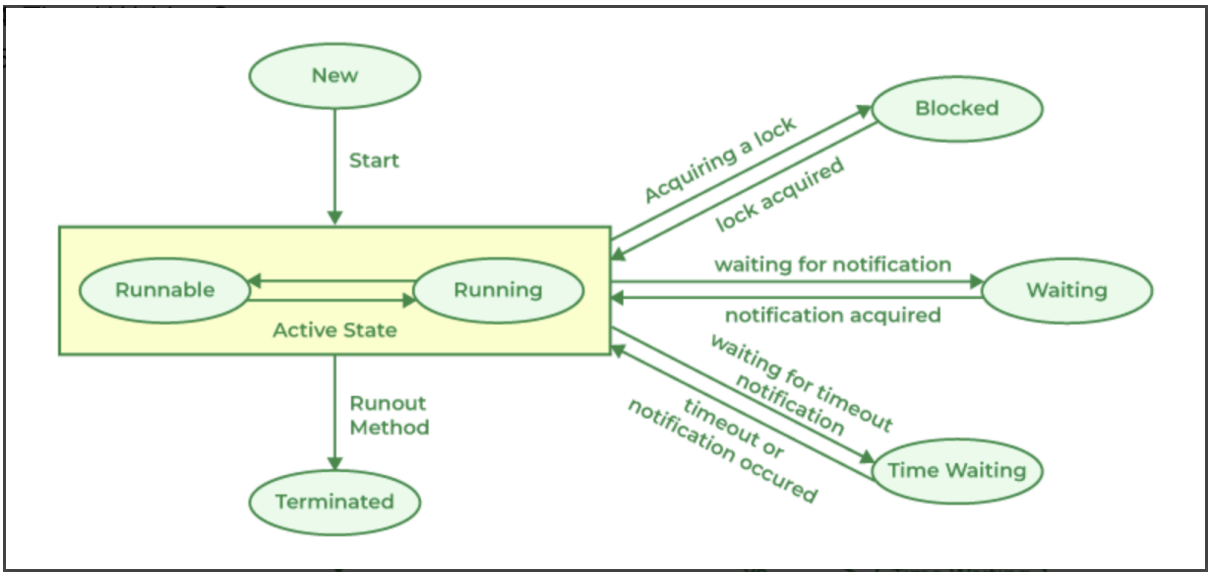
A Java thread is the smallest unit of execution within a program. It is a lightweight subprocess that runs independently but shares the same memory space of the process, allowing multiple tasks to execute concurrently.

For example, In MS Word one thread formats the document while another takes user input. Multithreading also keeps applications responsive, since other threads can continue running even if one gets stuck.

Life Cycle of Thread

During its lifetime, a thread transitions through several states, they are:

- New State
- Active State
- Waiting/Blocked State
- Timed Waiting State
- Terminated State



Working of Thread States

1. New State: By default, a thread will be in a new state, in this state, code has not yet started execution.

2. Active State: When a thread calls the start() method, it enters the Active state, which has two sub-states:

- Runnable State: The thread is ready to run but is waiting for the Thread Scheduler to give it CPU time. Multiple runnable threads share CPU time in small slices.

- Running State: When the scheduler assigns CPU to a runnable thread, it moves to the Running state. After its time slice ends, it goes back to the Runnable state, waiting for the next chance to run.

3. Waiting/Blocked State: a thread is temporarily inactive, it may be in the Waiting or Blocked state:

- Waiting: If thread T1 needs to use a camera but thread T2 is already using it, T1 waits until T2 finishes.
- Blocked: If two threads try to use the same resource at the same time, one may be blocked until the other releases it.

When multiple threads are waiting or blocked, the Thread Scheduler decides which one gets CPU based on priority.

4. Timed Waiting State: Sometimes threads may face starvation if one thread keeps using the CPU for a long time while others keep waiting.

For example: If T1 is doing a long important task, T2 may wait indefinitely. To avoid this, Java provides the Timed Waiting state, where methods like sleep() allow a thread to pause only for a fixed time. After the time expires, the thread gets a chance to run.

5. Terminated State: A thread enters the Terminated state when its task is finished or it is explicitly stopped. In this state, the thread is dead and cannot be restarted. If you try to call start() on a terminated thread, it will throw an exception.

Create Threads in Java

We can create threads in java using two ways

1. Extending Thread Class
2. Implementing a Runnable interface

1. By Extending Thread Class

Create a class that extends Thread. Override the run() method, this is where you put the code that the thread should execute. Then create an object of your class and call the start() method. This will internally call run() in a new thread.

Example:

```
class MyThread extends Thread {
    // initiated run method for Thread
    public void run() {
        String str = "Thread Started Running...";
        System.out.println(str);
    }
}
Output
Thread Started Running...
```

2. Using Runnable Interface

Create a class that implements Runnable. Override the run() method, this contains the code for the thread. Then create a Thread object, pass your Runnable object to it and call start().

Example:

```
class MyThread implements Runnable {
    // Method to start Thread
    public void run() {
        String str = "Thread is Running Successfully";
        System.out.println(str);
    }

}
Output
Thread is Running Successfully
```

Note: Extend Thread when you don't need to extend any other class. Implement Runnable when your class already extends another class (preferred in most cases).

Running Threads in Java

There are two methods used for running Threads in Java:

- run() Method in Java
- start() Method in Java

Example: Running a thread using start() Method

```
import java.io.*;
import java.util.*;

// Method 1 - Thread Class
class ThreadImpl extends Thread {
      // Method to start Thread
      @Override
      public void run() {
          String str = "Thread Class Implementation Thread"
                      + " is Running Successfully";
        System.out.println(str);
    }
}

// Method 2 - Runnable Interface
class RunnableThread implements Runnable {
      // Method to start Thread
      @Override
    public void run() {
          String str = "Runnable Interface Implementation Thread"
                      + " is Running Successfully";
        System.out.println(str);
    }

}

public class Main {
    public static void main(String[] args){
          // Method 1 - Thread Class
          ThreadImpl t1 = new ThreadImpl();
          t1.start();

          // Method 2 - Runnable Interface
        RunnableThread g2 = new RunnableThread();
        Thread t2 = new Thread(g2);
        t2.start();
```

```
         try {
              // Ensures t1 finishes before proceeding
              t1.join();

              // Ensures t2 finishes before proceeding
              t2.join();
          } catch (InterruptedException e) {
              e.printStackTrace();
          }
    }
}

Output
Thread Class Implementation Thread is Running Successfully
Runnable Interface Implementation Thread is Running Successfully
```

The common mistake is starting a thread using run() instead of start() method.

```
Thread myThread = new Thread(MyRunnable());
myThread.run();  //should be start();
```

Note: We use start() to launch a new thread, which then calls the run() method in parallel. If we call run() directly, it works like a normal method call and no new thread is created.

Checking States of Thread in Java

Let us see the working of thread states by implementing them on Threads t1 and t2.

Example:

```
class ThreadExample implements Runnable {
    public void run() {
        try {
            Thread.sleep(102);
        } catch (InterruptedException i1) {
            i1.printStackTrace();
        }

        System.out.println("The state for t1 after it invoked join
method() on thread t2: " + " " + ThreadState.t1.getState());
```

```
        // implementing try-catch block
        try {
            Thread.sleep(202);
        } catch (InterruptedException i2) {
            i2.printStackTrace();
        }
    }
}

// Creation of ThreadState class
public class ThreadState implements Runnable {
    // t1 static to access it in other classes
    public static Thread t1;
    public static ThreadState o1;

    public void run() {
        ThreadExample threadExample = new ThreadExample();
        Thread t2 = new Thread(threadExample);

        t2.start();
        System.out.println("State of t2 Thread, post-calling of start()
method is: " + " " + t2.getState());

        try {
            Thread.sleep(202);
        } catch (InterruptedException i2) {
            i2.printStackTrace();
        }

        System.out.println("State of Thread t2 after invoking to method
sleep() is:" + " " + t2.getState());

        try {
            t2.join();
            System.out.println("State of Thread t2 after join() is: " +
t2.getState());
        } catch (InterruptedException i3) {
            i3.printStackTrace();
        }

        System.out.println("State of Thread t1 after completing the
execution is: " + " " + t1.getState());
    }
```

```
      public static void main(String args[]){
        o1 = new ThreadState();
        t1 = new Thread(o1);

        System.out.println("Post-spanning, state of t1 is: " +
t1.getState());

        // lets invoke start() method on t1
        t1.start();

        // Now, Thread t1 is moved to runnable state
        System.out.println("Post invoking of start() method, state of t1
is: " + " " + t1.getState());
    }
}
Output:
Post-spanning, state of t1 is: NEW
Post invoking of start() method, state of t1 is: RUNNABLE
State of t2 Thread, post-calling of start() method is: RUNNABLE
The state for t1 after it invoked join method() on thread t2:
TIMED_WAITING
State of Thread t2 after invoking to method sleep() is: TIMED_WAITING
State of Thread t2 after join() is: TERMINATED
State of Thread t1 after completing the execution is: RUNNABLE
```

Java Thread Class

The Thread class (in java.lang package) is used to create and control threads in Java. Each object of this class represents a single thread of execution.

Syntax

```
public class Thread extends Object implements Runnable
```

Constructors of Thread Class

Constructor	Action Performed
Thread()	Allocates a new Thread object.
Thread(Runnable target)	Allocates a new Thread object.
Thread(Runnable target, String name)	Allocates a new Thread object.
Thread(String name)	Allocates a new Thread object.
Thread(ThreadGroup group, Runnable target)	Allocates a new Thread object.

Thread(ThreadGroup group, Runnable target, String name)	It creates a new Thread with a target runnable object, a given name and belonging to a specified thread group.
Thread(ThreadGroup group, Runnable target, String name, long stackSize)	It creates a new Thread with a target runnable, a given name, a specified thread group and a defined stack size.
Thread(ThreadGroup group, String name)	Allocates a new Thread object.

Methods of Thread Class

Method	Action Performed
activeCount()	Returns estimate of active threads in current thread group & subgroups.
checkAccess()	Checks if the current thread has permission to modify this thread.
clone()	Throws CloneNotSupportedException (Thread cannot be cloned).
currentThread()	Returns reference of currently executing thread.
dumpStack()	Prints stack trace of current thread to standard error.
enumerate(Thread[] tarray)	Copies active threads into given array.
getAllStackTraces()	Returns map of stack traces for all live threads.
getContextClassLoader()	Gets context ClassLoader of this thread.
getDefaultUncaughtExceptionHandler( )	Returns default handler for uncaught exceptions.
getId()	Returns a unique identifier of thread.
getName()	Returns thread’s name.
getPriority()	Returns thread’s priority.
getStackTrace()	Returns stack trace elements of this thread.
getState()	Returns state of this thread (NEW, RUNNABLE, etc.).

getThreadGroup()	Returns the thread group of this thread.
getUncaughtExceptionHandler()	Returns handler for uncaught exceptions.
holdsLock(Object obj)	Checks if the current thread holds monitor lock on a given object.
interrupt()	Interrupts this thread.
interrupted()	Tests if current thread is interrupted (clears status).
isAlive()	Tests if thread is alive (started & not dead).
isDaemon()	Tests if thread is daemon.
isInterrupted()	Tests if thread is interrupted (without clearing status).
join()	Waiting for this thread to die.
join(long millis)	Waits max millis for this thread to die.
run()	Executes run() method (if Runnable given).
setContextClassLoader(ClassLoader cl)	Set context ClassLoader. setDaemon(boolean on) Marks thread as daemon or user.
setDefaultUncaughtExceptionHandler(handler)	Sets default handler for uncaught exceptions.
setName(String name)	Changes thread’s name.
setUncaughtExceptionHandler(handler)	Sets handler for uncaught exceptions of this thread.
setPriority(int newPriority)	Changes thread’s priority.
sleep(long millis)	Pauses current thread for given milliseconds.
start()	Start thread (JVM calls run()).
toString()	Returns string with thread name, priority & group.
yield()	Hints scheduler to pause current thread & give chance to others.

Concurrency Problems

- Race Condition: Occurs when multiple threads access shared data simultaneously, leading to inconsistent results. It happens when the code is not thread-safe.
- Deadlock: Happens when two or more threads are blocked forever, each waiting for the other to release a lock.
- Livelock: Threads are active but unable to make progress because they keep responding to each other in an endless loop.
- Thread Starvation: A thread is perpetually denied access to resources because other threads are given priority.
- Priority Inversion: Occurs when a low-priority thread holds a lock needed by a high-priority thread, blocking its progress.

Solution of the Problem: Synchronization , Locks , Atomic Variables , Thread-safe Collections , Avoiding Deadlocks , Thread Pools, Using volatile Keyword.

Let us see how we can avoid concurrency problems.

```
class Counter {
    private int count = 0;

    // Synchronized method to ensure thread-safe increment
    public synchronized void increment() {
        count++;
    }

    public int getCount() { return count; }
}

class CounterThread extends Thread {
    private Counter counter;

    public CounterThread(Counter counter){
        this.counter = counter;
    }

    @Override
    public void run(){
        System.out.println("Running the Thread");
        for (int i = 0; i < 1000; i++) {
            counter.increment();
        }
    }
}
```

```
public class SynchronizationExample {
    public static void main(String[] args) {
        // Create a shared Counter object
        Counter counter = new Counter();

        // Create multiple threads that will increment the counter
        Thread t1 = new CounterThread(counter);
        Thread t2 = new CounterThread(counter);
        Thread t3 = new CounterThread(counter);

        // Start the threads
        t1.start();
        t2.start();
        t3.start();

        // Wait for all threads to finish
        try {
            t1.join();
            t2.join();
            t3.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        // Print the final count
        System.out.println("Final count: " + counter.getCount());
    }
}

Output
Running the Thread
Running the Thread
Running the Thread
Final count: 3000
```

Explanation of the above Program:

- **Counter Class:** Contains a private integer count as the shared resource. The increment method is synchronized, ensuring that only one thread can execute it at a time, preventing concurrent modifications.
- **CounterThread Class: E**xtends thread and is designed to increment the shared counter object. The run method increments the counter 1000 times.
- **Main Class:** Creates a single counter instance shared among three CounterThread instances. Starts the threads and waits for them to finish using join. Prints the final count, which should be 3000 (1000 increments from each of the three threads).