UNIT-2

MULTI-THREADING

What is Thread in java

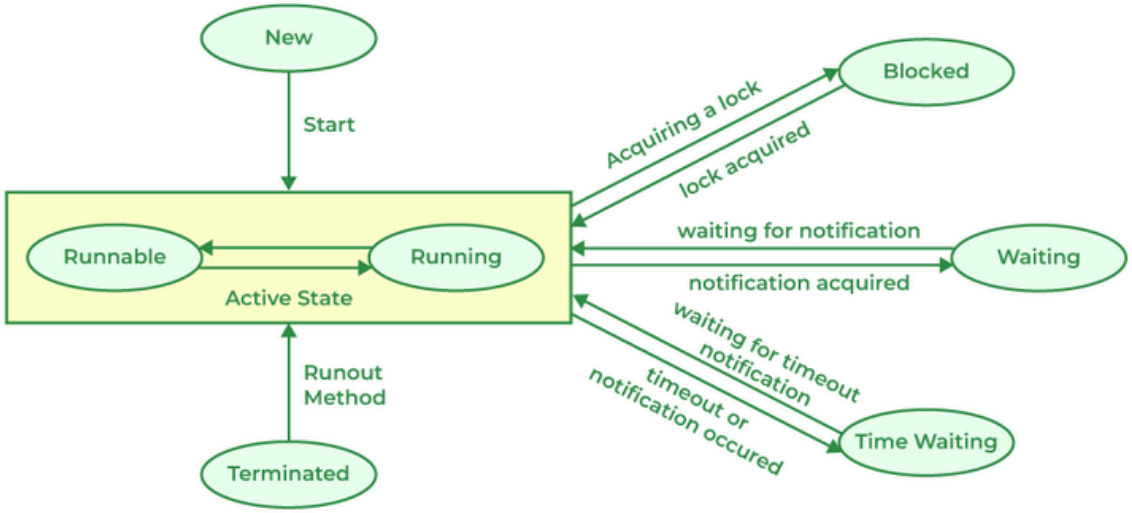
A thread is a lightweight subprocess, the smallest unit of processing. It is a separate path of execution.

Threads are independent. If there occurs exception in one thread, it doesn't affect other threads. It uses a shared memory area.

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 State
2. Runnable State
3. Blocked State
4. Waiting State
5. Timed Waiting State
6. Terminated State

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



States of Thread in its Lifecycle

Life Cycle of a Thread

There are multiple states of the thread in a lifecycle as mentioned below:

1. ***New Thread:*** *When a new thread is created, it is in the new state. The thread has not yet started to run when the thread is in this state. When a thread lies in the new state, its 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 a runnable state. In this state, a thread might actually be running or it might be ready to 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 lie in a runnable state.

3. ***Blocked:** The thread will be in blocked state when it is trying to acquire a lock but currently the lock is acquired by the other thread. The thread will move from the blocked state to runnable state when it acquires the lock.*
4. ***Waiting state:** The thread will be in waiting state when it calls wait() method or join() method. It will move to the runnable state when other thread will notify or that thread will be terminated.*
5. ***Timed Waiting:** A thread lies in a 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 a timed waiting state.*
6. ***Terminated State:** A thread terminates because of either of the following reasons:*
   - *Because it exits normally. This happens when the code of the thread has been entirely executed by the program.*
   - *Because there occurred some unusual erroneous event, like a segmentation fault or an unhandled exception.*

Creating a Thread

There are two ways to create a thread.

It can be created by extending the `Thread` class and overriding its `run()` method:

Ex

Extend Syntax

```
public class Main extends Thread {

  public void run() {

    System.out.println("This code is running in a thread");

  }

}
```

Another way to create a thread is to implement the `Runnable` interface:

Implement Syntax

```
public class Main implements Runnable {

  public void run() {
```

```
    System.out.println("This code is running in a thread");
  }
}
```

Running Threads

If the class extends the `Thread` class, the thread can be run by creating an instance of the class and call its `start()` method:

Extend Example

```
public class Main extends Thread {
  public static void main(String[] args) {
    Main thread = new Main();
    thread.start();
    System.out.println("This code is outside of the thread");
  }
  public void run() {
    System.out.println("This code is running in a thread");
  }
}
```

If the class implements the `Runnable` interface, the thread can be run by passing an instance of the class to a `Thread` object's constructor and then calling the thread's `start()` method:

Implement Example

```
public class Main implements Runnable {
  public static void main(String[] args) {
    Main obj = new Main();
    Thread thread = new Thread(obj);
    thread.start();
```

```
    System.out.println("This code is outside of the thread");
  }
  public void run() {
    System.out.println("This code is running in a thread");
  }
}
```

Priority of a Thread (Thread Priority)

Each thread has a priority. Priorities are represented by a number between 1 and 10. In most cases, the thread scheduler schedules the threads according to their priority (known as preemptive scheduling). But it is not guaranteed because it depends on JVM specification that which scheduling it chooses. Note that not only JVM a Java programmer can also assign the priorities of a thread explicitly in a Java program.

Setter & Getter Method of Thread Priority

Let's discuss the setter and getter method of the thread priority.

public final int getPriority(): The java.lang.Thread.getPriority() method returns the priority of the given thread.

public final void setPriority(int newPriority): The java.lang.Thread.setPriority() method updates or assign the priority of the thread to newPriority. The method throws IllegalArgumentException if the value newPriority goes out of the range, which is 1 (minimum) to 10 (maximum).

3 constants defined in Thread class:

1. public static int MIN_PRIORITY
2. public static int NORM_PRIORITY
3. public static int MAX_PRIORITY

Default priority of a thread is 5 (NORM_PRIORITY). The value of MIN_PRIORITY is 1 and the value of MAX_PRIORITY is 10.

Example

- Java

```
// Java Program to Illustrate Priorities in Multithreading
// via help of getPriority() and setPriority() method

// Importing required classes
import java.lang.*;

// Main class
class ThreadDemo extends Thread {

    // Method 1
    // run() method for the thread that is called
    // as soon as start() is invoked for thread in main()
    public void run()
    {
        // Print statement
        System.out.println("Inside run method");
    }

    // Main driver method
    public static void main(String[] args)
    {
        // Creating random threads
```

```
        // with the help of above class
        ThreadDemo t1 = new ThreadDemo();
        ThreadDemo t2 = new ThreadDemo();
        ThreadDemo t3 = new ThreadDemo();

        // Thread 1
        // Display the priority of above thread
        // using getPriority() method
        System.out.println("t1 thread priority : "
                           + t1.getPriority());

        // Thread 1
        // Display the priority of above thread
        System.out.println("t2 thread priority : "
                           + t2.getPriority());

        // Thread 3
        System.out.println("t3 thread priority : "
                           + t3.getPriority());

        // Setting priorities of above threads by
        // passing integer arguments
```

```
        t1.setPriority(2);

        t2.setPriority(5);

        t3.setPriority(8);



        // t3.setPriority(21); will throw

        // IllegalArgumentException



        // 2

        System.out.println("t1 thread priority : "

                           + t1.getPriority());



        // 5

        System.out.println("t2 thread priority : "

                           + t2.getPriority());



        // 8

        System.out.println("t3 thread priority : "

                           + t3.getPriority());



        // Main thread



        // Displays the name of

        // currently executing Thread
```

```
        System.out.println(

            "Currently Executing Thread : "

            + Thread.currentThread().getName());



        System.out.println(

            "Main thread priority : "

            + Thread.currentThread().getPriority());



        // Main thread priority is set to 10

        Thread.currentThread().setPriority(10);



        System.out.println(

            "Main thread priority : "

            + Thread.currentThread().getPriority());

    }

}
```

Output

```
t1 thread priority : 5
t2 thread priority : 5
t3 thread priority : 5
t1 thread priority : 2
t2 thread priority : 5
t3 thread priority : 8
Currently Executing Thread : main
Main thread priority : 5
Main thread priority : 10
```

Output explanation:

- Thread with the highest priority will get an execution chance prior to other threads. Suppose there are 3 threads t1, t2, and t3 with priorities 4, 6, and 1. So, thread t2 will execute first based on maximum priority 6 after that t1 will execute and then t3.
- The default priority for the main thread is always 5, it can be changed later. The default priority for all other threads depends on the priority of the parent thread.

Thread Synchronization in Java

Thread Synchronization is used to coordinate and ordering of the execution of the threads in a multi-threaded program. There are two types of thread synchronization are mentioned below:

- Mutual Exclusive
- Cooperation (Inter-thread communication in Java)

Mutual Exclusive

Mutual Exclusive helps keep threads from interfering with one another while sharing data. There are three types of Mutual Exclusive mentioned below:

- Synchronized method.
- Synchronized block.
- Static synchronization.

- **Synchronized method** It is a method that can be declared synchronized using the keyword “synchronized” before the method name. By writing this, it will make the code in a method thread-safe so that no resources are shared when the method is executed.

- **Synchronized Block**
  If a block is declared as synchronized then the code which is written inside a method is only executed instead of the whole code. It is used when sequential access to code is required.

Static Synchronization

The method is declared static in this case. It means that lock is applied to the class instead of an object and only one thread will access that class at a time.

Importance of Thread Synchronization in Java

- Thread synchronization in java is used to avoid the error called memory consistency which is caused due to inconsistency view of shared memory.
- If a method is declared synchronized then the thread holds that object of the thread until the other thread is done executing.
- It is used for controlling the access of multiple threads and to access shared resources.
- It avoids the deadlock situations in the thread.

Inter-thread communication in Java is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed.

There are several situations where communication between threads is important.

For example, suppose that there are two threads A and B. Thread B uses data produced by Thread A and performs its task.

If Thread B waits for Thread A to produce data, it will waste many CPU cycles. But if threads A and B communicate with each other when they have completed their tasks, they do not have to wait and check each other's status every time.

Note: *Inter-thread communication is also known as* ***Cooperation in Java.***

To avoid polling, Java uses three methods, namely, **wait(), notify(), and notifyAll().** All these methods belong to object class as final so that all classes have them. They must be used within a synchronized block only.

- **wait():** It tells the calling thread to give up the lock and go to sleep until some other thread enters the same monitor and calls notify().
- **notify():** It wakes up one single thread called wait() on the same object. It should be noted that calling notify() does not give up a lock on a resource.
- **notifyAll():** It wakes up all the threads called wait() on the same object.

These methods can be called only from within a synchronized method or synchronized block of code otherwise, an exception named **IllegalMonitorStateException** is thrown.
All these methods are declared as final. Since it throws a checked exception, therefore, you must be used these methods within Java try-catch block.