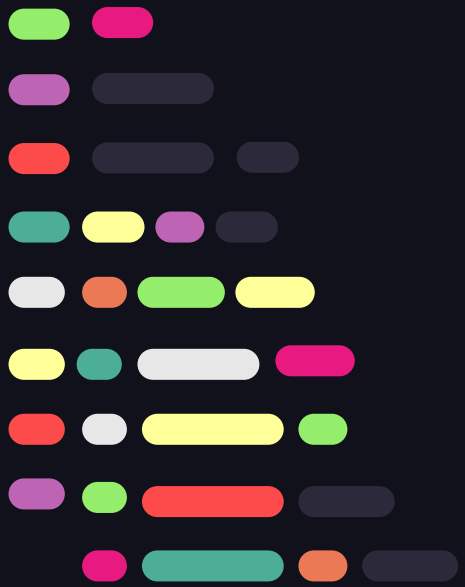
Java Programming Week-07

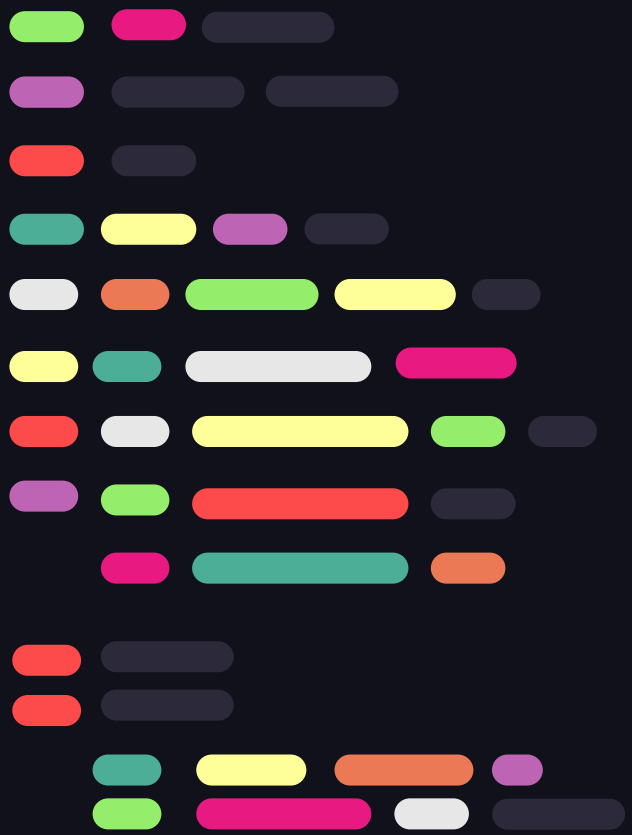
<Mohan De Zoysa –IIT JAVA CERTIFICATION>

Outline



Java Multi Threading

- Introduction
- Creating a Thread
- Thread lifecycle



Java Thread

In Java, a **thread** refers to a lightweight process that executes within a larger program. It is a unit of execution that can perform tasks independently. In a Java program, multiple threads can run concurrently, allowing for parallel execution of tasks.

The ability of a program to concurrently execute multiple regions of code provides capabilities that are difficult or impossible to achieve with strictly sequential languages.

```java
class MyThread extends Thread {
    @Override
    public void run() {
        // Code to be executed by the thread
        for (int i = 1; i <= 5; i++) {
            System.out.println("Thread: " + i);
        }
    }
}

public class ThreadExample {
    public static void main(String[] args) {
        MyThread myThread = new MyThread();
        myThread.start(); // Start the thread
    }
}
```

Creating a Thread

In Java, you can create threads using two primary methods:

Extending the Thread Class:

This method involves creating a new class that extends the java.lang.Thread class

- You override the run() method in your subclass to specify the code that the thread should execute.
- You then create an instance of your custom thread class and call the start() method on it to start the thread's execution

```java
class MyRunnable implements Runnable {

    @Override
    public void run() {
        // Code to be executed by the thread
        for (int i = 1; i <= 5; i++) {
            System.out.println("Runnable: " + i);
        }
    }
}

public class ThreadExample {

    public static void main(String[] args) {
        MyRunnable myRunnable = new MyRunnable();
        Thread thread = new Thread(myRunnable);
        thread.start(); // Start the thread
    }
}
```

Creating a Thread

Implementing the Runnable Interface:

- This method involves creating a class that implements the java.lang.Runnable interface.
- Define the thread's logic within the run() method of the Runnable interface.
- Then create an instance of Runnable class and pass it to a Thread object's constructor.
- Finally, call the start() method on the Thread object to start the thread's execution.

Java Thread Lifecycle

Java threads can be in various states during their lifecycle. Here are the different states along with an explanation and a simplified diagram

New:

- When a thread is created but has not yet started, it is in the "New" state.
- At this point, the thread has been instantiated but not yet started using the start() method.

Runnable:

- A thread is in the "Runnable" state when it's ready to be executed.
- This means that the thread is either currently executing or is eligible to run.

Blocked (Waiting for a monitor lock)

- A thread enters the "Blocked" state when it's waiting for a monitor lock to enter a synchronized block or method.
- It's blocked because another thread holds the lock

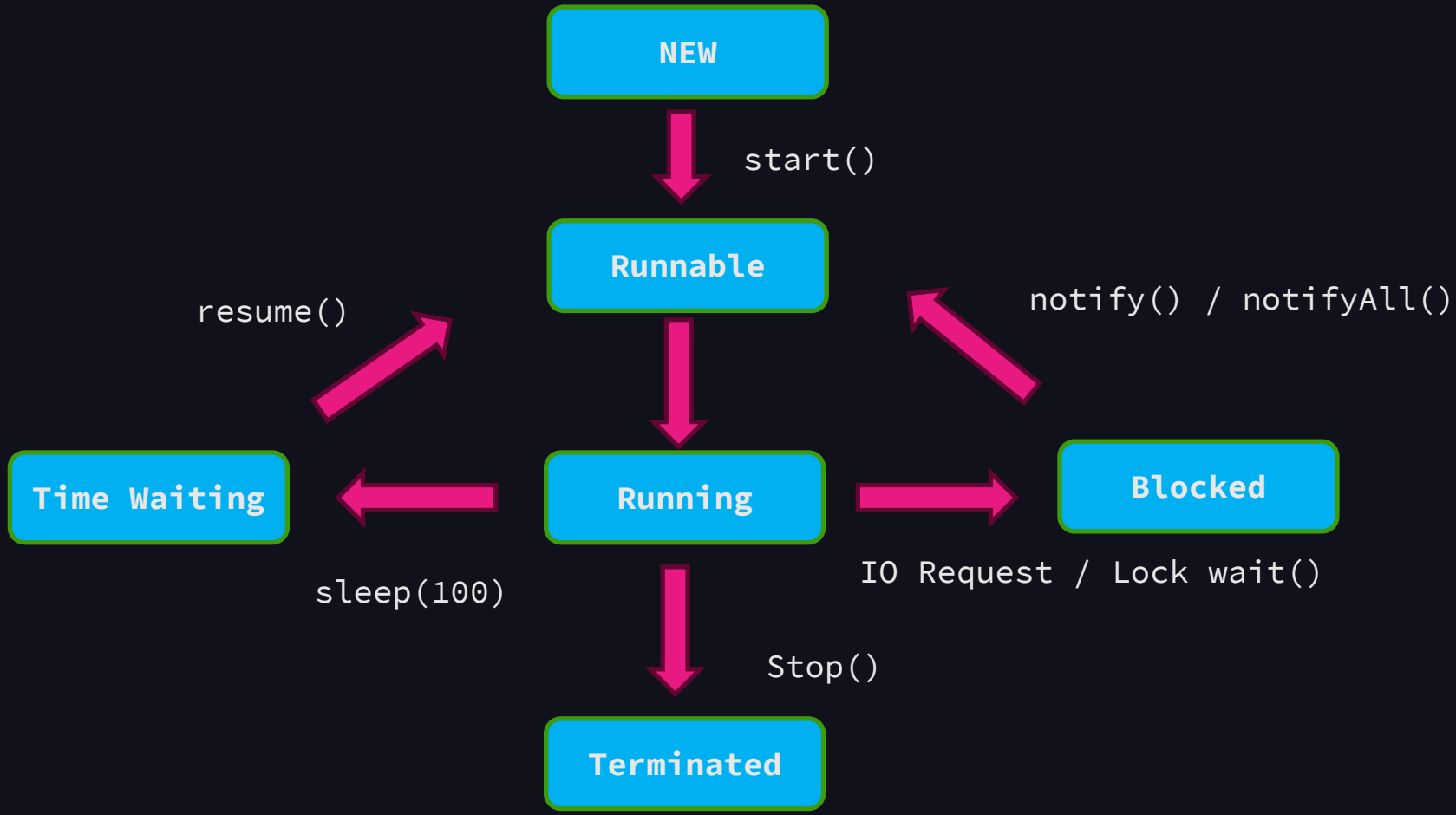
Timed Waiting:

A thread is in a "Timed Waiting" state when it's waiting for a specified amount of time (as defined by Thread.sleep() or similar methods).

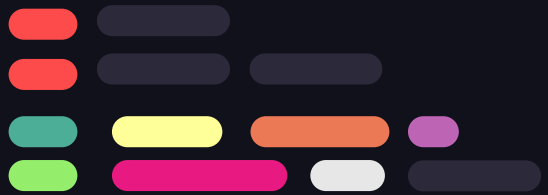
Terminated:

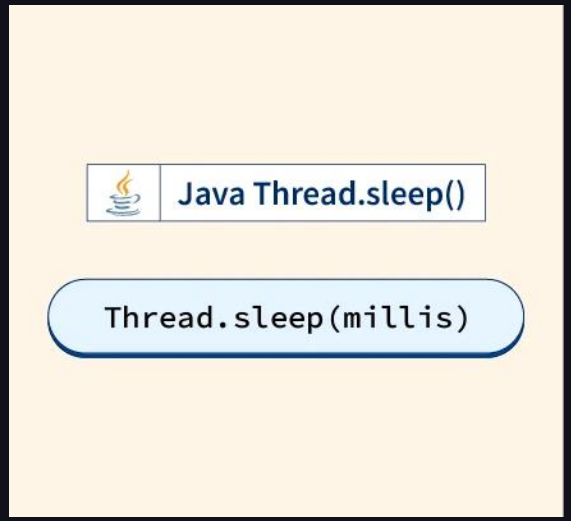
A runnable thread gets into the terminated state because it accomplishes its task or else terminates.

Java Thread Lifecycle



Sleep Method





Thread Sleep

Thread.sleep() is a mechanism in Java that allows a thread to pause its execution for a specified duration

Thread Sleep Interaction with Thread Scheduler: When a thread calls Thread.sleep(), it does interact with the thread scheduler, but it's essential to note that the thread scheduler is part of the Java Virtual Machine (JVM), not the operating system. The JVM's thread scheduler is responsible for managing the execution of Java threads on the underlying hardware threads provided by the operating system.

Wait State and Runnable State: When a thread invokes Thread.sleep(), it enters a "timed waiting" state, not a "wait" state. In this state, the thread is not actively executing, but it is scheduled to resume after the specified sleep duration. Once the sleep time elapses, the thread transitions back to the "runnable" state, indicating that it's ready to run and is waiting for its turn on the CPU.

Example - : 01

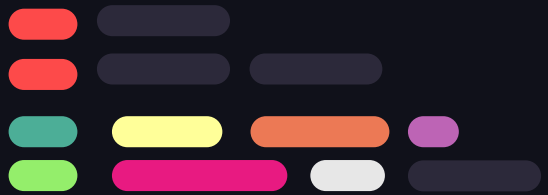
Create a countdown that simulates a clock-like display where the numbers change with a delay of a second

- Count down from 100 to 0
- At the end Print count down has completed

Sleep Method Sample

```
public class ThreadSleep {

    public static void main(String[] args) throws InterruptedException {
        long start = System.currentTimeMillis();

        Thread.sleep(2000);

        System.out.println("Sleep for = " + (System.currentTimeMillis() - start));
    }
}
```

Join Method



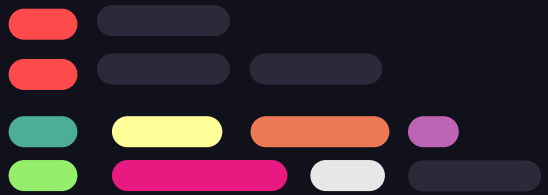
Thread Join

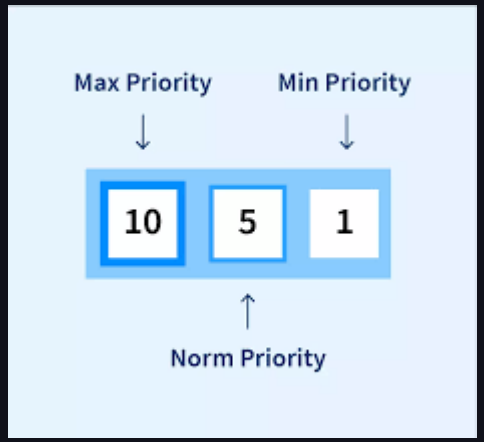
In Java, the join() method is used to ensure that a thread waits for another thread to complete its execution before it continues. This is useful when you need to coordinate the execution of multiple threads.

```
public class JoinExample {
    public static void main(String[] args) {
        Thread thread1 = new Thread(new MyRunnable(), "Thread 1");
        Thread thread2 = new Thread(new MyRunnable(), "Thread 2");

        thread1.start();
        thread2.start();

        try {
            // Wait for thread1 to finish
            thread1.join();

            // Wait for thread2 to finish
            thread2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("Both threads have finished executing.");
    }
}
```

```
class MyRunnable implements Runnable {
    @Override
    public void run() {
        for (int i = 1; i <= 5; i++) {

System.out.println(Thread.currentThread().ge
tName() + ": " + i);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e)
{
                e.printStackTrace();
            }
        }
    }
}
```

Java Thread Priorities





Java Thread Priority

Thread priority in Java is a way to influence the scheduling order of threads, but it does not determine the number of services or tasks a thread can handle.

1 is known as the lowest priority.
5 is known as standard priority.
10 represents the highest level of priority.

The main thread's priority is set to 5 by default, and each child thread will have the same priority as its parent thread. We have the ability to adjust the priority of any thread, whether it is the main thread or a user-defined thread. It is advised to adjust the priority using the Thread class's constants, which are as follows:

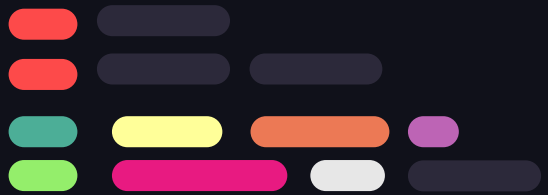
- Thread.MIN_PRIORITY;
- Thread.NORM_PRIORITY;
- Thread.MAX_PRIORITY;

Java Thread Priority Example

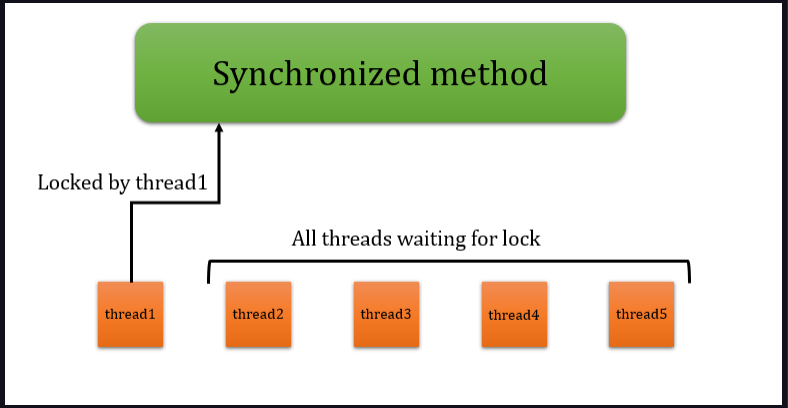
```
public class PriorityExample {
    public static void main(String[] args) {
        Thread thread1 = new Thread(new MyRunnable(), "Thread 1");
        Thread thread2 = new Thread(new MyRunnable(), "Thread 2");

        thread1.setPriority(Thread.MAX_PRIORITY);
        thread2.setPriority(Thread.MIN_PRIORITY);
        thread1.start();
        thread2.start();
    }
} MyRunnable

class MyRunnable implements Runnable {
    @Override
    public void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println(Thread.currentThread().getName() + " is running");
        }
    }
}
```

{..

Java Thread synchronization



}..



Java Thread Synchronization

Thread synchronization in Java is the process of coordinating multiple threads to ensure that they access shared resources or sections of code in a way that prevents data corruption and race conditions. Synchronization is essential in multithreaded programs to maintain data integrity and prevent unpredictable behavior. Java provides several mechanisms for thread synchronization

Synchronized Blocks and Methods:

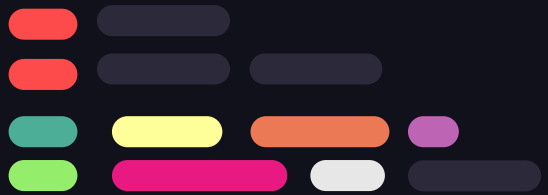
use the synchronized keyword to create synchronized blocks or methods. When a thread enters a synchronized block or method, it acquires the lock on the associated object. Other threads attempting to enter synchronized blocks or methods on the same object will be blocked until the lock is released.

Java Thread Priority Example

```
//Synchronized Method
public synchronized int getCount() {
        return count;
    }
```

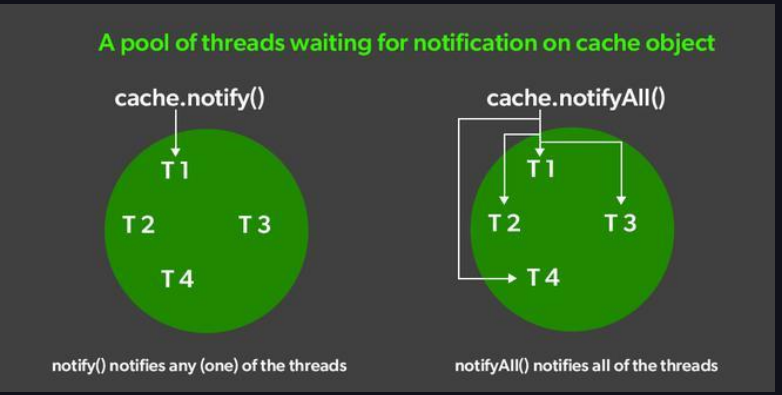
{ . .

wait(), notify() and notifyAll() methods



} . .

Java Thread Synchronization



The threads can communicate with each other through **wait(), notify()** and **notifyAll()** methods in Java. These are **final** methods defined in the **Object** class and can be called only from within a **synchronized** context.

The **wait()** method causes the current thread to wait until another thread invokes the **notify()** or **notifyAll()** methods for that object.

The **notify()** method **wakes up a single thread** that is waiting on that object's monitor.

The **notifyAll()** method **wakes up all threads** that are waiting on that object's monitor. A thread waits on an object's monitor by calling one of the **wait()** method.

These methods can throw **IllegalMonitorStateException** if the current thread is not the owner of the object's monitor.

Example 01

Java program that creates two threads, one for printing odd numbers and the other for printing even numbers. Both threads share a common counter and synchronize access to it to ensure that odd and even numbers are printed alternatively when the printNumbers method is called:

Scenario: Online Ticket Booking System

- Consider an online ticket booking system for a popular event, like a concert or a sports match. In such a system, multiple users are trying to book tickets simultaneously. Multithreading can be used to manage and improve the performance and user experience of this system

- **User Requests:** When users visit the website to book tickets, their requests are handled by the web server.

- **Multithreading:** The web server employs multithreading to handle these incoming requests concurrently. Each incoming request is assigned to a separate thread for processing. This allows multiple users to interact with the system simultaneously.

- **Ticket Availability:** The system needs to keep track of the available tickets for the event. Multiple threads may attempt to access and update this information at the same time
- Seat Reservation: When a user selects seats and proceeds to book them, the booking process involves multiple steps like payment processing and seat reservation. These steps can be executed in parallel using multiple threads, improving the efficiency of the booking process.

- **Confirmation:** After successful booking, the system sends confirmation emails or tickets to the users. This can also be done in parallel for multiple users, enhancing the system's performance.

Question: Restaurant Table Allocation Scenario

You are tasked with simulating a restaurant scenario in Java where there is a limited number of tables, and multiple customers (threads) want to dine at the restaurant. Your goal is to implement a program that manages table allocation for customers using thread coordination.

A Restaurant class that represents the restaurant and manages table availability. Customer threads that enter the restaurant, dine, and leave. The Restaurant class should have the following methods:

enterRestaurant(): Allows customers to enter the restaurant.
If there are no available tables, customers should wait until a table becomes available. Use wait() and notify() or notifyAll() for thread coordination.

dine(): Simulates the customer's dining activity.

leaveRestaurant(): Allows customers to leave the restaurant. Increment the available tables and notify waiting customers that a table is available.

Each customer should follow this sequence:

- Enter the restaurant.
- Dine (simulated by a sleep).
- Leave the restaurant.

Create a main program that initiates the restaurant and handles multiple customer threads. Ensure that customers coordinate to access tables appropriately.