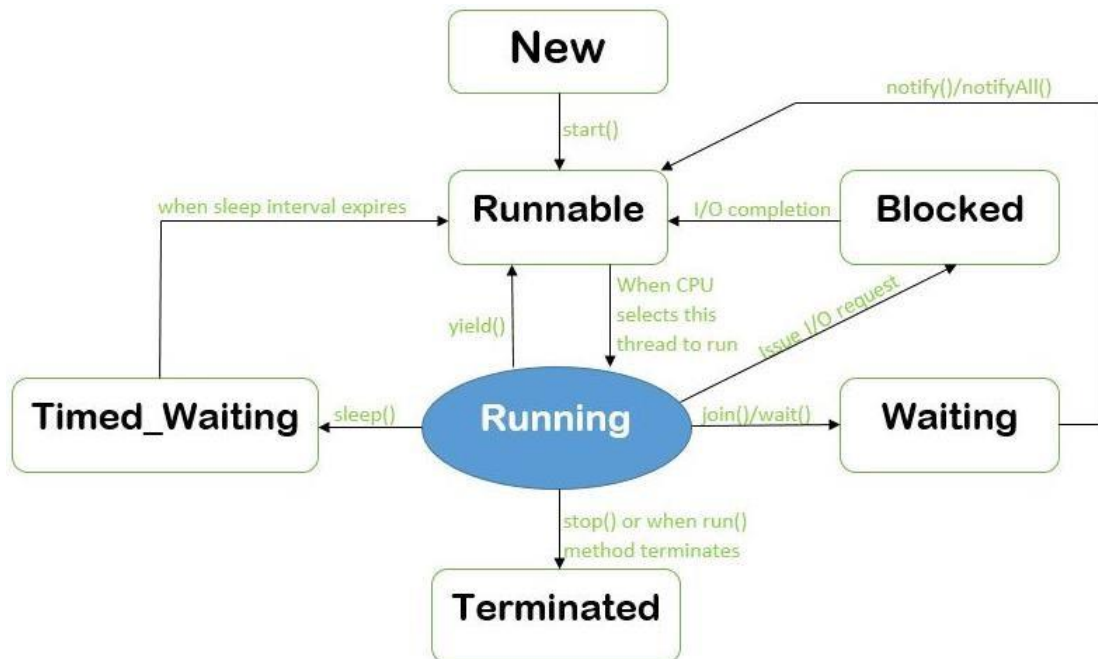


UNIT V – MULTITHREADING, GUI PROGRAMMING & EVENT HANDLING

5.1 Thread lifecycle:

- In Java, a thread represents a separate path of execution. The lifecycle of a thread is controlled by the Thread class and Runnable interface.
- A thread goes through the following states (defined in Thread.State enum):



1. New (Created):

- When a thread is created but not yet started, it is in the new state. The thread has not begun execution.
- `Thread t = new Thread();`
- `System.out.println(t.getState());` // Output: NEW

2. Runnable:

- After calling `start()`, the thread is ready to run but waiting for CPU scheduling.
- Example:
- `t.start();` // moves thread to runnable state
- `System.out.println(t.getState());` // Output: RUNNABLE

3. Running:

- The thread is in this state when the CPU picks it up from the runnable pool and executes its `run()` method.
- Only one thread per core can run at a time.

4. Waiting:

- A thread is in the Waiting state when it calls methods like wait(), join(), or LockSupport.park().
- In this state, the thread is waiting indefinitely for another thread to perform a specific action (like notifying it to continue execution, notify() or notifyAll()).

5. Timed Waiting:

- Similar to the Waiting state, a thread enters the Timed Waiting state when it calls methods with a timeout, such as sleep(milliseconds) or wait(milliseconds).
- The thread will remain in this state until the specified time elapses or it receives a notification.
- Example: sleep(1000), join(2000), wait(500).

6. Terminated (Dead):

- A thread enters the Terminated state when it has completed its execution, either by returning from the run() method or by being terminated due to an exception.
- At this point, the thread is no longer alive and cannot be restarted.

5.2 Thread Creation (Thread class, Runnable interface):

There are two main ways to create threads in Java:

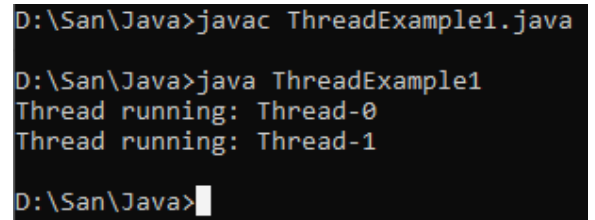
1. By Extending Thread Class:

- Create a class that extends Thread.
- Override the run() method.
- Create an object and call start() (not run() directly!).

Example:

```
class MyThread extends Thread
{
    public void run()
    {
        System.out.println("Thread running: " +
            Thread.currentThread().getName());
    }
}
```

```
public class ThreadExample1
{
    public static void main(String[] args)
    {
        MyThread t1 = new
        MyThread(); MyThread t2 =
        new MyThread(); t1.start(); //
        starts thread 1 t2.start(); //
        starts thread 2
    }
}
```



```
D:\San\Java>javac ThreadExample1.java
D:\San\Java>java ThreadExample1
Thread running: Thread-0
Thread running: Thread-1
D:\San\Java>
```

2. By Implementing Runnable Interface

- Create a class that implements Runnable.
- Override run() method.
- Pass the object of this class to a Thread object and call start().

Example:

class MyRunnable implements Runnable

```
{
    public void run() {
        System.out.println("Thread running: " + Thread.currentThread().getName());
    }
}

public class ThreadExample2
{
    public static void main(String[] args)
    {
        MyRunnable task = new MyRunnable();
        Thread t1 = new Thread(task);
        Thread t2 = new Thread(task);
        t1.start(); // starts thread 1
        t2.start(); // starts thread 2
    }
}
```

```
D:\San\Java>javac ThreadExample2.java
D:\San\Java>java ThreadExample2
Thread running: Thread-1
Thread running: Thread-0
D:\San\Java>
```

Difference between Thread class and Runnable interface

Aspect	Extending Thread	Implementing Runnable
Inheritance	Cannot extend any other class	Can still extend another class
Code reusability	Less flexible	More flexible
Usage	Simple for small programs	Preferred in real-world applications

5.3 Thread synchronization:

- When multiple threads access shared resources (like variables, objects, files, databases), race conditions can occur.
- Synchronization ensures that only one thread can access the shared resource at a time that prevents data inconsistency.

Why Synchronization?

Without synchronization:

- Two or more threads may update the same data simultaneously.
- This causes race conditions that lead to unexpected or incorrect results.

1. Synchronized Method

- We can declare a method as synchronized using the synchronized keyword. This ensures that only one thread can execute the method at a time for a given object.

2. Synchronized Block

- Instead of synchronizing the entire method, we can synchronize only a part of the code using a synchronized block.
- This improves performance (less locking overhead).

3. Inter-Thread Communication (wait, notify, notifyAll)

Synchronization also allows threads to communicate safely.

- wait() → causes current thread to wait until another thread calls notify().
- notify() → wakes up one waiting thread.
- notifyAll() → wakes up all waiting threads.

Key Points about Synchronization

- Every object in Java has a monitor lock.
- A synchronized method/block acquires the lock before executing.
- At most one thread can hold the lock at a time.
- Synchronization avoids race conditions but can cause deadlocks if not handled properly.

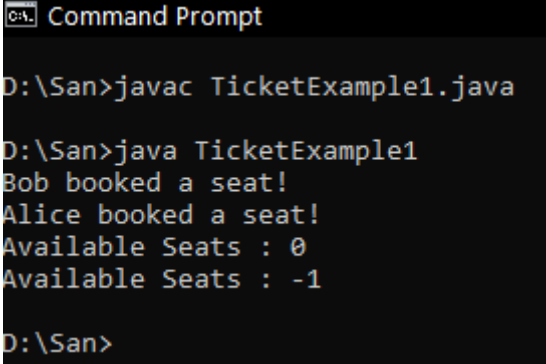
Example Program - Without Synchronization (Ticket Booking)

```

class TicketBooking {
    private int availableSeats = 1; // only 1 seat
    public void bookTicket(String name)
    {
        if (availableSeats > 0)
        {
            System.out.println(name + " booked a seat!");
            availableSeats--;
            System.out.println("Available Seats : "+availableSeats);
        }
        else
        {
            System.out.println("Sorry, " + name + " no seats available.");
        }
    }
}

public class TicketExample1
{
    public static void main(String[] args)
    {
        TicketBooking booking = new TicketBooking();
        Thread t1 = new Thread(() -> booking.bookTicket("Alice"));
        Thread t2 = new Thread(() -> booking.bookTicket("Bob"));
        t1.start();
        t2.start();
    }
}

```



```

C:\> Command Prompt

D:\San>javac TicketExample1.java

D:\San>java TicketExample1
Bob booked a seat!
Alice booked a seat!
Available Seats : 0
Available Seats : -1

D:\San>

```

Example Program - With Synchronization (Safe Booking)

```

class TicketBooking
{
    private int availableSeats = 1; // only 1 seat
    public synchronized void bookTicket(String name)
    {

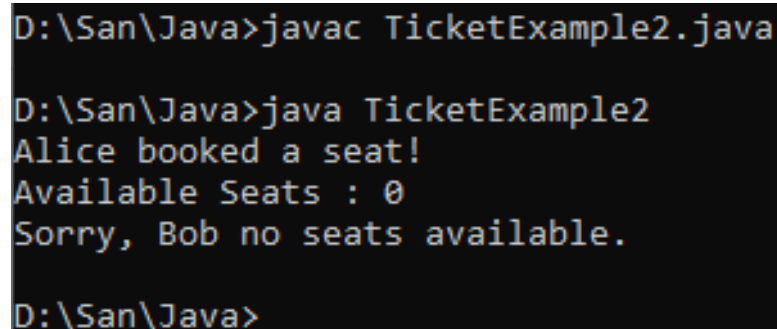
```

```

        if (availableSeats > 0)
        {
            System.out.println(name + " booked a seat!");
            availableSeats--;
            System.out.println("Available Seats : "+availableSeats);
        }
        else {
            System.out.println("Sorry, " + name + " no seats available.");
        }
    }
}

public class TicketExample2
{
    public static void main(String[] args)
    {
        TicketBooking booking = new TicketBooking();
        Thread t1 = new Thread(() -> booking.bookTicket("Alice"));
        Thread t2 = new Thread(() -> booking.bookTicket("Bob"));
        t1.start();
        t2.start();
    }
}

```



```

D:\San\Java>javac TicketExample2.java

D:\San\Java>java TicketExample2
Alice booked a seat!
Available Seats : 0
Sorry, Bob no seats available.

D:\San\Java>

```

5.4 Inter-thread communication:

When multiple threads work on shared resources, they often need to coordinate.

Example: In a restaurant,

- The chef (Producer) prepares food.
- The waiter (Consumer) serves food to customers.
- If the chef keeps cooking without waiting, the kitchen overflows.
- If the waiter serves without food, customers get angry.

They must coordinate. That's what inter-thread communication is.

Java provides three important methods (in Object class) for communication:

1. wait()

- Causes the current thread to release the lock and go into a waiting state.
- It will wait until another thread notifies it.
- Must be called inside synchronized.

2. notify()

- Wakes up one thread that is waiting on the object's lock.
- The chosen thread will move from waiting → runnable state.

3. notifyAll()

- Wakes up all threads waiting on the object's lock.
- But only one will get the lock and proceed.

Flow of Inter Thread Communication:

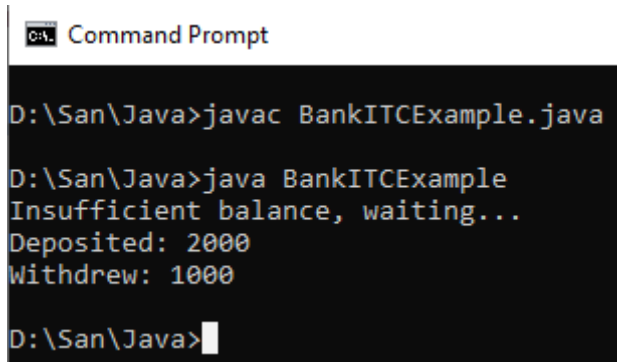
- Thread acquires lock (synchronized).
- Thread checks a condition.
- If condition not met → calls wait() → releases lock → goes into waiting pool.
- Another thread does its work and calls notify() or notifyAll().
- Waiting thread(s) move back to runnable and fight for CPU time.

Example 1: Simple Explanation with Bank Deposit/Withdraw

```
class BankAccount
{
    private int balance = 0;
    public synchronized void deposit(int amount)
    {
        balance += amount;
        System.out.println("Deposited: " + amount);
        notify(); // wake up waiting withdraw thread
    }
    public synchronized void withdraw(int amount)
    {
        while (balance < amount)
        { // wait until enough balance
            System.out.println("Insufficient balance, waiting...");
            try
            {
                wait();
            }
            catch (InterruptedException e) {}
        }
        balance -= amount;
    }
}
```



```
        System.out.println("Withdrew: " + amount);
    }
}
public class BankITCExample
{
    public static void main(String[] args)
    {
        BankAccount account = new BankAccount();
        Thread withdrawThread = new Thread(() -> account.withdraw(1000));
        Thread depositThread = new Thread(() -> account.deposit(2000));
        withdrawThread.start();
        depositThread.start();
    }
}
```

Output:

```
C:\> Command Prompt

D:\San\Java>javac BankITCExample.java

D:\San\Java>java BankITCExample
Insufficient balance, waiting...
Deposited: 2000
Withdrew: 1000

D:\San\Java>
```