Thread Synchronization in Java

1. Definition of Thread Synchronization

Thread synchronization in Java is a mechanism used to control the access of multiple threads to a shared resource. It ensures that only one thread at a time can execute a critical section of code when accessing a shared resource.

When multiple threads access and modify the same data simultaneously, a problem called a race condition may occur. Synchronization prevents inconsistent results by controlling the execution of threads.

Example:

Consider a bank account with a balance of ₹10,000. Two threads simultaneously attempt to withdraw ₹7,000.

- Thread 1 checks the balance.
- Thread 2 checks the balance.
- Both threads may find sufficient balance.
- Both may withdraw ₹7,000.

This can result in an incorrect account balance.

Synchronization ensures that one thread completes its withdrawal operation before another thread accesses the same account.

2. Need for Thread Synchronization

Thread synchronization is required for the following reasons:

1. Data consistency: Maintains correct data when multiple threads access shared variables.
2. Race condition prevention: Prevents unpredictable results due to simultaneous execution.
3. Resource protection: Ensures controlled access to shared resources.
4. Thread coordination: Helps threads communicate and coordinate their execution.
5. Avoiding data corruption: Prevents multiple threads from modifying shared data simultaneously.

3. Ways to Achieve Thread Synchronization in Java

Java provides several mechanisms to achieve synchronization.

1. Synchronized Method

A synchronized method allows only one thread at a time to execute that method on the same object.

Syntax:

```
synchronized void methodName() {
    // Critical section
}
```

Example:

```
class Table {
    synchronized void printTable(int n) {
        for (int i = 1; i <= 5; i++) {
            System.out.println(n * i);
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                System.out.println(e);
            }
        }
    }
}

class MyThread1 extends Thread {
    Table t;

    MyThread1(Table t) {
        this.t = t;
    }

    public void run() {
        t.printTable(5);
```

```
    }
}

class MyThread2 extends Thread {
    Table t;

    MyThread2(Table t) {
        this.t = t;
    }

    public void run() {
        t.printTable(10);
    }
}

public class SynchronizationDemo {
    public static void main(String[] args) {
        Table obj = new Table();

        MyThread1 t1 = new MyThread1(obj);
        MyThread2 t2 = new MyThread2(obj);

        t1.start();
        t2.start();
    }
}
```

Explanation:

- Both threads share the same Table object.
- The printTable() method is declared synchronized.

- When one thread executes the method, the other thread must wait.
- The second thread executes only after the first thread releases the object's lock.

Important: A synchronized instance method locks the current object (this).

2. Synchronized Block

A synchronized block allows us to synchronize only a particular portion of a method instead of the entire method.

Syntax:

```
synchronized (object) {
   // Critical section
}
```

Example:

```
class Display {
   void show() {
      System.out.println("Before synchronization");

      synchronized (this) {
         for (int i = 1; i <= 5; i++) {
            System.out.println(i);
         }
      }

      System.out.println("After synchronization");
   }
}
```

Explanation:

- Only the statements inside the synchronized block are protected.
- Other statements outside the block can execute without acquiring that lock.
- It improves performance when only a small portion of code requires synchronization.

3. Static Synchronization

Static synchronization is used when multiple threads need to access a shared static resource.

A static synchronized method locks the class object rather than an individual instance.

Syntax:

```
static synchronized void methodName() {
    // Critical section
}
```

Example:

```
class Student {
    static synchronized void display() {
        System.out.println(
            Thread.currentThread().getName()
            + " is accessing static method"
        );
    }
}

public class StaticSyncDemo {
    public static void main(String[] args) {

        Thread t1 = new Thread(() -> Student.display());
        Thread t2 = new Thread(() -> Student.display());

        t1.start();
        t2.start();
    }
}
```

Explanation:

- The display() method is static and synchronized.
- Both threads acquire the same class-level lock.

- Only one thread can execute this method at a time, regardless of how many objects of the class exist.

4. Inter-thread Communication

Java provides three important methods for communication between threads:

Method	Description
wait()	Makes the current thread wait until another thread notifies it.
notify()	Wakes up one thread waiting on the same object's monitor.
notifyAll()	Wakes up all threads waiting on the same object's monitor.

These methods belong to the Object class.

Example:

```
class Shared {
   synchronized void produce() {
      System.out.println("Producer is waiting");

      try {
         wait();
      } catch (InterruptedException e) {
         System.out.println(e);
      }

      System.out.println("Producer resumed");
   }

   synchronized void consume() {
      System.out.println("Consumer is notifying");
      notify();
   }
}
```

```
public class InterThreadDemo {
    public static void main(String[] args) {

        Shared obj = new Shared();

        Thread t1 = new Thread(() -> obj.produce());
        Thread t2 = new Thread(() -> obj.consume());

        t1.start();

        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            System.out.println(e);
        }

        t2.start();
    }
}
```