Thread Communication - Thread communication in Java refers to the coordination and interaction between multiple threads to achieve a specific task or to exchange data safely. It allows threads to synchronize their activities, share information, and control the order of execution.

Synchronization –

Multi-threaded programs may often come to a situation where multiple threads try to access the same resources and finally produce erroneous and unforeseen results.

Thread synchronization in Java is a way of programming several threads to carry out independent tasks easily. It can control access to multiple threads to a particular shared resource. It is used to make sure that only one thread can access same resource at a given point of time.

Need of Synchronization –

- To prevent interference between threads.
- To prevent the problem of consistency.

Types of Synchronization -

1. **Process Synchronization** – It is a technique used to coordinate the execution of multiple processes. It ensures that the shared resources are safe and in order.
2. **Thread Synchronization -** Thread synchronization refers to the concept where only one thread is executed at a time while other threads are in the waiting state. This process is called thread synchronization. It is used because it avoids interference of thread and the problem of inconsistency.

Types of Thread Synchronization –

1. **Mutual Exclusive** - It is used for preventing threads from interfering with each other and to keep distance between the threads while sharing any resources.
   It can be achieved using –

   Synchronized Methods

   - In Java, when a thread invokes a synchronized method, it automatically acquires the intrinsic lock (also known as monitor lock) associated with the object the method belongs to.
   - This means that only one thread can execute any synchronized method of an object at a time. Other threads attempting to execute

synchronized methods on the same object will be blocked until the lock is released.

- This mechanism guarantees that the code within the synchronized method executes atomically (as a whole, without interruption from other threads) with respect to shared resources accessed by the method.

```
public synchronized void synchronizedMethod() {
   // Method body
}
```

Synchronized Blocks

- Sometimes we may not want to synchronize an entire method but only a specific section of code within the method. In such cases, synchronized blocks are used.
- In synchronized blocks, we specify the object whose lock we want to acquire. Only one thread can execute synchronized blocks that acquire the lock of the same object at a time.

```
public void someMethod() {
   synchronized (lockObject) {
      // Synchronized code block
   }
}
```

Static Synchronization

- Static synchronization is used when you want to synchronize access to static fields or methods at the class level rather than the instance level.
- When a method is declared as static synchronized, it acquires the lock on the class's Class object.
- Static synchronization applies to all instances of the class and prevents multiple threads from concurrently executing the static synchronized method or block on the same class.

```
public class MyClass {
   public static synchronized void staticSyncMethod() {
      // Method body
   }
}
```

2. **Inter-thread communication** or **Cooperation** - It is a mechanism in Java in which a thread running in the critical section is paused and another thread is allowed to enter or lock the same critical section that is executed.

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

    public synchronized void increment() {
        count++;
    }

    public synchronized int getCount() {
        return count;
    }
}

class IncrementThread extends Thread {
    private Counter counter;

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

    public void run() {
        for (int i = 0; i < 1000; i++) {
            counter.increment();
        }
    }
}

public class Synchronization {
    public static void main(String[] args) throws InterruptedException {
        Counter counter = new Counter();

        IncrementThread thread1 = new IncrementThread(counter);
        IncrementThread thread2 = new IncrementThread(counter);

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

        // Wait for threads to finish execution
        thread1.join();
        thread2.join();

        System.out.println("Final Count: " + counter.getCount());
    }
}
```

Output –

Final Count: 2000