Synchronization in Java

Synchronization in Java is a critical concept in concurrent programming that ensures multiple threads can interact with shared resources safely. In a nutshell, synchronization prevents race conditions, where the outcome of operations depends on the timing of thread execution. It is the capability to control the access of multiple threads to any shared resource. Synchronization is a better option where we want to allow only one thread to access the shared resource.

Synchronization in Java tackles these problems through the capacity of a single thread to have exclusive access to either a synchronized block of code or a synchronized method associated with an object in question at a time. There are two primary mechanisms for synchronization in Java: synchronized blocks and synchronized methods.

1. **Synchronized Blocks**
   Synchronization in Java tackles these problems through the capacity of a single thread to have exclusive access to either a synchronized block of code or a synchronized method associated with an object in question at a time. There are two primary mechanisms for synchronization in Java: synchronized blocks and synchronized methods.

```
synchronized (object) {
    // Synchronized code block
}
```

   This monitor object or lock is the subject. While only one thread can be holding a lock on a monitor object at one instance. Other threads that want to go into the synchronized blocks with this object must wait till the lock becomes available.

2. **Synchronized Methods**
   In Java, you can declare entire methods as synchronized which prevent multiple threads from accessing the method simultaneously. With this, synchronization becomes a simpler process because the mechanism is applied to all invocations of the synchronized method automatically.

```
class SynchronizedCounter {
    private int count = 0;
    public synchronized void increment() {
        count++;
    }
    public synchronized int getCount() {
        return count;
    }
}
```

Types of Synchronization

There are the following two types of synchronization:

1. Process Synchronization
2. Thread Synchronization

Thread Synchronization

There are two types of thread synchronization mutual exclusive and inter-thread communication.

1. Mutual Exclusive
    1. Synchronized method.
    2. Synchronized block.
    3. Static synchronization.
2. Cooperation (Inter-thread communication in Java)

1. Mutual Exclusive

Mutual Exclusive helps keep threads from interfering with one another while sharing data. It can be achieved by using the following three ways:

1. By Using Synchronized Method
2. By Using Synchronized Block
3. By Using Static Synchronization

Concept of Lock in Java

Synchronization is built around an internal entity known as the lock or monitor. Every object has a lock associated with it. By convention, a thread that needs consistent access to an object's fields has to acquire the object's lock before accessing them, and then release the lock when it's done with them.

Understanding The Problem Without Synchronization

```
class Table {
    // Method to print the table, not synchronized
    void printTable(int n) {
```

```
        for(int i = 1; i <= 5; i++) {
            // Print the multiplication result
            System.out.println(n * i);
            try {
                // Pause execution for 400 milliseconds
                Thread.sleep(400);
            } catch(Exception e) {
                // Handle any exceptions
                System.out.println(e);
            }
        }
    }
}
class MyThread1 extends Thread {
    Table t;
    // Constructor to initialize Table object
    MyThread1(Table t) {
        this.t = t;
    }
    // Run method to execute thread
    public void run() {
        // Call printTable method with argument 5
        t.printTable(5);
    }
}
class MyThread2 extends Thread {
    Table t;
    // Constructor to initialize Table object
    MyThread2(Table t) {
        this.t = t;
    }
    // Run method to execute thread
    public void run() {
        // Call printTable method with argument 100
        t.printTable(100);
    }
}
class TestSynchronization1 {
    public static void main(String args[]) {
        // Create a Table object
        Table obj = new Table();
        // Create MyThread1 and MyThread2 objects with the same Table object
        MyThread1 t1 = new MyThread1(obj);
        MyThread2 t2 = new MyThread2(obj);
        // Start both threads
        t1.start();
        t2.start();
    }
}
```

Output:

5
100
10
200
15
300
20
400
25
500

2. Java Synchronized Method

If you declare any method as synchronized, it is known as synchronized method. Synchronized method is used to lock an object for any shared resource. When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.

```
class Table {
    // Synchronized method to print the table
    synchronized void printTable(int n) {
        for(int i = 1; i <= 5; i++) {
            // Print the multiplication result
            System.out.println(n * i);
            try {
                // Pause execution for 400 milliseconds
                Thread.sleep(400);
            } catch(Exception e) {
```

```
                // Handle any exceptions
                System.out.println(e);
            }
        }
    }
}
class MyThread1 extends Thread {
    Table t;
    // Constructor to initialize Table object
    MyThread1(Table t) {
        this.t = t;
    }
    // Run method to execute thread
    public void run() {
        // Call synchronized method printTable with argument 5
        t.printTable(5);
    }
}
class MyThread2 extends Thread {
    Table t;
    // Constructor to initialize Table object
    MyThread2(Table t) {
        this.t = t;
    }
    // Run method to execute thread
    public void run() {
        // Call synchronized method printTable with argument 100
        t.printTable(100);
    }
}
public class TestSynchronization2 {
    public static void main(String args[]) {
        // Create a Table object
        Table obj = new Table();
        // Create MyThread1 and MyThread2 objects with the same Table object
        MyThread1 t1 = new MyThread1(obj);
        MyThread2 t2 = new MyThread2(obj);
        // Start both threads
        t1.start();
        t2.start();
    }
}
```

Output:
5
10
15
20
25
100
200
300
400
500

3. Synchronized Block in Java

Synchronized block can be used to perform synchronization on any specific resource of the method. Suppose we have 50 lines of code in our method, but we want to synchronize only 5 lines, in such cases, we can use synchronized block. If we put all the codes of the method in the synchronized block, it will work same as the synchronized method.

```
class Table
{
 void printTable(int n){
   synchronized(this){//synchronized block
     for(int i=1;i<=5;i++){
      System.out.println(n*i);
      try{
       Thread.sleep(400);
      }catch(Exception e){System.out.println(e);}
     }
   }
 }//end of the method
}

class MyThread1 extends Thread{
        Table t;
        MyThread1(Table t){
                this.t=t;
```

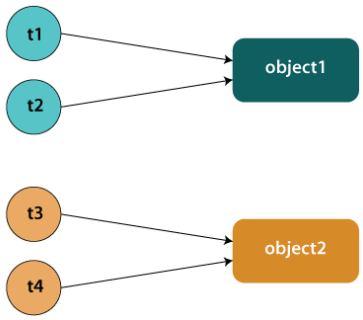
```
        }
        public void run(){
                t.printTable(5);
        }

}
class MyThread2 extends Thread{
        Table t;
        MyThread2(Table t){
                this.t=t;
        }
        public void run(){
                t.printTable(100);
        }
}

public class TestSynchronizedBlock1{
        public static void main(String args[]){
                Table obj = new Table();//only one object
                MyThread1 t1=new MyThread1(obj);
                MyThread2 t2=new MyThread2(obj);
                t1.start();
                t2.start();
        }
}
```

Output:
5
10
15
20
25
100
200
300
400
500

Static Synchronization
If you make any static method as synchronized, the lock will be on the class not on object.



Problem without static synchronization
Suppose there are two objects of a shared class (e.g. Table) named object1 and object2. In case of synchronized method and synchronized block there cannot be interference between t1 and t2 or t3 and t4 because t1 and t2 both refers to a common object that have a single lock. But there can be interference between t1 and t3 or t2 and t4 because t1 acquires another lock and t3 acquires another lock. We don't want interference between t1 and t3 or t2 and t4. Static synchronization solves this problem.

Example of Static Synchronization

```
class Table  {
         synchronized static void printTable(int n){
           for(int i=1;i<=10;i++){
             System.out.println(n*i);
             try{
               Thread.sleep(400);
             }catch(Exception e){}
           }
         }
        }
class MyThread1 extends Thread{
        public void run(){
                Table.printTable(1);
```

```
        }
}
class MyThread2 extends Thread{
        public void run(){
                Table.printTable(10);
        }
}
class MyThread3 extends Thread{
        public void run(){
                Table.printTable(100);
        }
}
class MyThread4 extends Thread{
        public void run(){
                Table.printTable(1000);
        }
}
public class TestSynchronization4{
        public static void main(String t[]){
                MyThread1 t1=new MyThread1();
                MyThread2 t2=new MyThread2();
                MyThread3 t3=new MyThread3();
                MyThread4 t4=new MyThread4();
                t1.start();
                t2.start();
                t3.start();
                t4.start();
        }
}
```

Advantages of Synchronization in Java

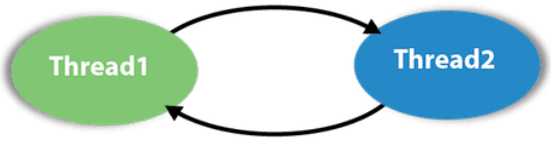
1. Thread Safety
2. Consistency
3. Data Visibility
4. Prevention of Deadlocks
5. Coordination
6. Efficient Resource Utilization
7. Compatibility with Legacy Code

Disdvantages of Synchronization in Java

1. Performance Overhead
2. Potential for Deadlocks
3. Reduced Scalability
4. Complexity and Maintenance
5. Potential for Livelocks
6. Difficulty in Debugging
7. Decreased Concurrency
8. Potential for Performance Degradation with I/O Operations

Deadlock in Java

Deadlock can occur in a situation when a thread is waiting for an object lock, that is acquired by another thread and second thread is waiting for an object lock that is acquired by first thread. Since, both threads are waiting for each other to release the lock, the condition is called deadlock.



Example of Deadlock in Java

```
public class TestDeadlockExample1 {
          public static void main(String[] args) {
            final String resource1 = "Virat Kohli";
            final String resource2 = "Rohit Sharma";
            // t1 tries to lock resource1 then resource2
            Thread t1 = new Thread() {
              public void run() {
                  synchronized (resource1) {
                   System.out.println("Thread 1: locked resource 1");

                   try { Thread.sleep(100);} catch (Exception e) {}

                   synchronized (resource2) {
                    System.out.println("Thread 1: locked resource 2");
                   }
                 }
              }
            };

            // t2 tries to lock resource2 then resource1
            Thread t2 = new Thread() {
              public void run() {
                 synchronized (resource2) {
                   System.out.println("Thread 2: locked resource 2");

                   try { Thread.sleep(100);} catch (Exception e) {}

                   synchronized (resource1) {
                     System.out.println("Thread 2: locked resource 1");
                   }
                 }
              }
            };
            t1.start();
            t2.start();
          }
}
```

Output:
Thread 1: locked resource 1
Thread 2: locked resource 2

```
public class DeadlockSolved {

    public static void main(String ar[]) {
        DeadlockSolved test = new DeadlockSolved();

        final resource1 a = test.new resource1();
        final resource2 b = test.new resource2();

   // Thread-1
Runnable b1 = new Runnable() {
    public void run() {
        synchronized (b) {
            try {
                /* Adding delay so that both threads can start trying to lock resources */
                Thread.sleep(100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            // Thread-1 have resource1 but need resource2 also
            synchronized (a) {
                System.out.println("In block 1");
            }
        }
    }
};
```

```
// Thread-2
Runnable b2 = new Runnable() {
    public void run() {
        synchronized (b) {
            // Thread-2 have resource2 but need resource1 also
            synchronized (a) {
                System.out.println("In block 2");
            }
        }
    }
};


        new Thread(b1).start();
        new Thread(b2).start();
    }

    // resource1
    private class resource1 {
        private int i = 10;

        public int getI() {
            return i;
        }

        public void setI(int i) {
            this.i = i;
        }
    }

    // resource2
    private class resource2 {
        private int i = 20;

        public int getI() {
            return i;
        }

        public void setI(int i) {
            this.i = i;
        }
    }
}
```

Inter-thread Communication in Java

Inter-thread communication or **Co-operation** is all about allowing synchronized threads to communicate with each other.

Cooperation (Inter-thread communication) 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. It is implemented by following methods of **Object class**:

- wait()
- notify()
- notifyAll()

1) wait() method

The wait() method causes current thread to release the lock and wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.

The current thread must own this object's monitor, so it must be called from the synchronized method only otherwise it will throw exception.

2) notify() method

The notify() method wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation.

Syntax:

```
public final void notify()
```

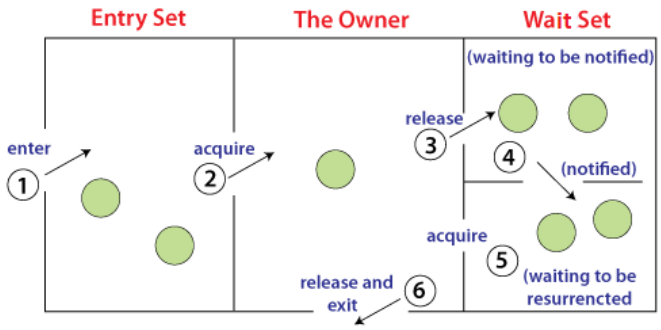
3) notifyAll() method

Wakes up all threads that are waiting on this object's monitor.

Syntax:

```
public final void notifyAll()
```

Understanding the process of inter-thread communication



1. Threads enter to acquire lock.
2. Lock is acquired by one thread.
3. Now thread goes to waiting state if you call wait() method on the object. Otherwise it releases the lock and exits.
4. If you call notify() or notifyAll() method, thread moves to the notified state (runnable state).
5. Now thread is available to acquire lock.
6. After completion of the task, thread releases the lock and exits the monitor state of the object.

Difference between wait and sleep?

Let's see the important differences between wait and sleep methods.

wait()	**sleep()**
The wait() method releases the lock.	The sleep() method doesn't release the lock.
It is a method of Object class	It is a method of Thread class
It is the non-static method	It is the static method
It should be notified by notify() or notifyAll() methods	After the specified amount of time, sleep is completed.

Example of Inter Thread Communication in Java

```
class Customer{
        int amount=10000;

        synchronized void withdraw(int amount){
        System.out.println("going to withdraw...");

        if(this.amount<amount){
        System.out.println("Less balance; waiting for deposit...");
        try{wait();}catch(Exception e){}
        }
        this.amount-=amount;
        System.out.println("withdraw completed...");
        }

        synchronized void deposit(int amount){
        System.out.println("going to deposit...");
        this.amount+=amount;
        System.out.println("deposit completed... ");
        notify();
        }
        }

        class Test{
        public static void main(String args[]){
                final Customer c=new Customer();
        new Thread(){
                public void run(){
                        c.withdraw(15000);}
        }.start();
        new Thread(){
                public void run(){
                        c.deposit(10000);}
        }.start();

}}
```

Output:

going to withdraw...
Less balance; waiting for deposit...
going to deposit...
deposit completed...
withdraw completed