

Synchronization in Java

Synchronization in java is the capability *to control the access of multiple threads to any shared resource*.

Java Synchronization is better option where we want to allow only one thread to access the shared resource.

Why use Synchronization

The synchronization is mainly used to

1. To prevent thread interference.
2. To prevent consistency problem.

Types of Synchronization

There are two types of synchronization

1. Process Synchronization
2. Thread Synchronization

Here, we will discuss only 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)

Mutual Exclusive

Mutual Exclusive helps keep threads from interfering with one another while sharing data. This can be done by three ways in java:

1. by synchronized method
2. by synchronized block
3. by static synchronization

Concept of Lock in Java

Synchronization is built around an internal entity known as the lock or monitor. Every object has an 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.

From Java 5 the package `java.util.concurrent.locks` contains several lock implementations.

Understanding the problem without Synchronization

In this example, there is no synchronization, so output is inconsistent. Let's see the example:

```
class Table{
    void printTable(int n){//method not synchronized
        for(int i=1;i<=5;i++){
            System.out.println(n*i);
            try{
                Thread.sleep(400);
            } catch(Exception 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(100);
    }
}
```

```
}
```

```
class TestSynchronization1{  
    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();  
    }  
}
```

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.

```
Synchronized void printTable(int n){.....}
```

Synchronized Block in Java

Synchronized block can be used to perform synchronization on any specific resource of the method.

Suppose you have 50 lines of code in your method, but you want to synchronize only 5 lines, you can use synchronized block.

If you put all the codes of the method in the synchronized block, it will work same as the synchronized method.

Points to remember for Synchronized block

- Synchronized block is used to lock an object for any shared resource.
- Scope of synchronized block is smaller than the method.

Syntax to use synchronized block

```
synchronized (object reference expression) {  
    //code block }
```

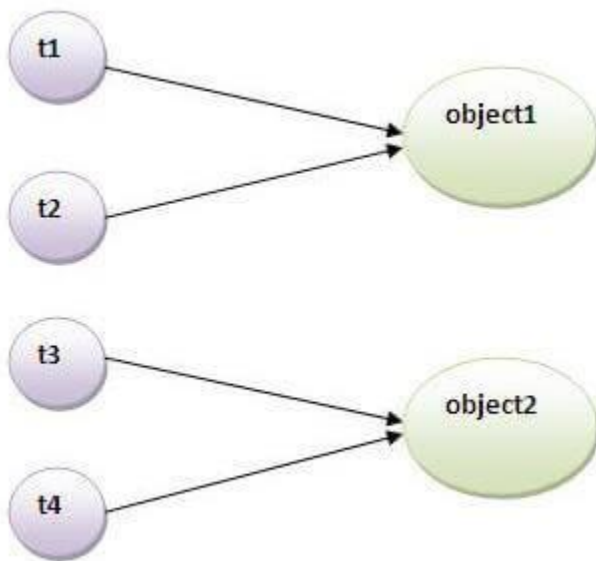
```

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);}
            }
        }
    }
}

```

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. I want no interference between t1 and t3 or t2 and t4. Static synchronization solves this problem.

Example of static synchronization

In this example we are applying synchronized keyword on the static method to perform 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);
    }
}

public class TestSynchronization4{
    public static void main(String t[]){
        MyThread1 t1=new MyThread1();
        MyThread2 t2=new MyThread2();
        MyThread3 t3=new MyThread3();
        t1.start();
        t2.start();
        t3.start();
    }
}
```

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()

```
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 ThreadMain{
    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();
    }
}
```

```

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 Myt1 extends Thread{
    Customer c;
    Myt1(Customer c){
        this.c=c;
    }
    public void run(){
        c.withdraw(15000);
    }
}

class Myt2 extends Thread{
    Customer c;
    Myt2(Customer c){
        this.c=c;
    }
    public void run(){
        c.deposit(10000);
    }
}

class ThreadMain{
    public static void main(String args[]){
        final Customer c=new Customer();
        Myt1 t1=new Myt1(c);
        Myt2 t2=new Myt2(c);
        t1.start();
        t2.start();
    }
}

```