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

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)

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{

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();
}
}
```

Output:

```
5
    100
```

```
10
200
15
300
20
400
25
500
```

****

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 void printTable(int n){//synchronized method
  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);
}
}

public class TestSynchronization2{
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
```

****

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.

Syntax

```
synchronized (object reference expression) {

  //code block

}
```

Program:

```
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

****

WAP to prevent concurrent booking of a ticket using the concept of thread synchronization.

Code:

```
// Example that shows multiple threads
// can execute the same method but in
// synchronized way.
import java.util.*;
class TicketBooking
{
// if multiple threads(trains) trying to access
// this synchronized method on the same Object
// but only one thread will be able
// to execute it at a time.
Scanner sc=new Scanner(System.in);
synchronized public void bookTicket()
{
int fare;
try
{
```

```
System.out.println("Enter fair of ticket");
fare=sc.nextInt();
System.out.println("Wait your ticket booking is in process");
for(int i=0;i<8;i++)
{
System.out.print("..\t");
Thread.sleep(500);
}
}
catch (Exception e)
{
System.out.println(e);
}
System.out.println("\nYour Ticket has been confirmed Successfully..");
}
}
class Passenger extends Thread
{
// Reference variable of type Line.
TicketBooking ticket;
Passenger(TicketBooking ticket)
{
this.ticket = ticket;
}
@Override
```

```
public void run()
{
ticket.bookTicket();
}
}
class BookTicket
{
public static void main(String[] args)
{
TicketBooking obj = new TicketBooking();
// we are creating two threads which share
// same Object.
Passenger p1 = new Passenger(obj);
Passenger p2 = new Passenger(obj);
// both passengers will try to book ticket at same time .
p1.start();
p2.start();
}
}
```

****

Write a multithreaded program that generates the Fibonacci sequence.
This program should work as follows:

create a class Input that reads the number of Fibonacci numbers that the program is to generate.
The class will then create a separate thread that will generate the Fibonacci numbers, placing the sequence in an array.
When the thread finishes execution, the parent thread (Input class) will output the sequence generated by the child thread.
Because the parent thread cannot begin outputting the Fibonacci sequence until the child thread finishes, the parent thread will have to wait for the child thread to finish.

```
import java.util.*;

public class Input

{

public static void main(String[] args)

{

int no;

Scanner sc=new Scanner(System.in);

System.out.println("Enter No of elements to be generate in Fibonacci series: ");

no=sc.nextInt();

FibonacciThread f = new FibonacciThread(no);

f.start();

synchronized(f) //Lock an object

{

try

{

System.out.println("Parent Thread Waiting for Fibonacci Thread to generate all fibonacci Numbers and store the same in an array...");

f.wait();

}catch(InterruptedException e)

{

e.printStackTrace();
```

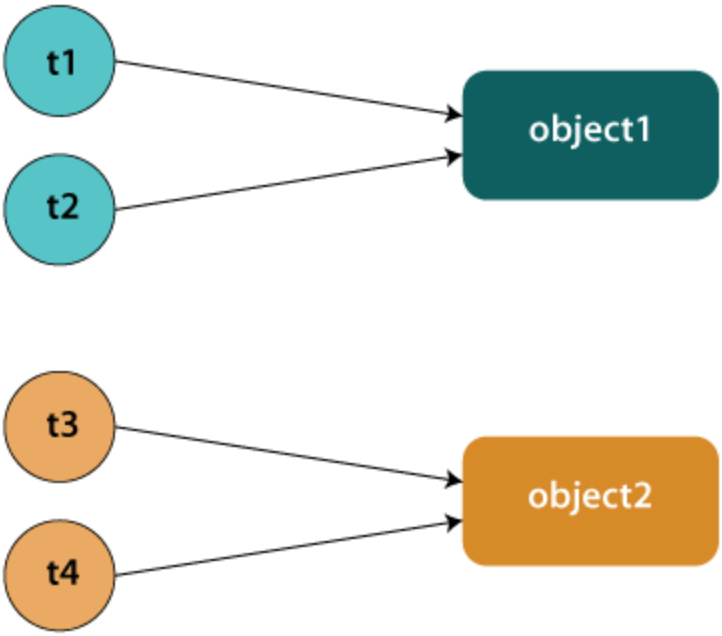
```
}

System.out.println("Fibonacci Thread has been Notified to Parent Class: Series has been
generated Successfully.");

System.out.println("Main Thread(parent Thread) Printing the Fibonacci Elements from Array...");

f.printFibonacci(no);

System.out.println("Parent Thread(Main Thread) has completed its Porcess....");

}

}

}

class FibonacciThread extends Thread

{

int noOfElements,fib1=0,fib2=1,fib3,fib[];

FibonacciThread(int no)

{

noOfElements=no;

}

@Override

public void run()

{

synchronized(this)

{

fib=new int[noOfElements];

fib[0]=fib1;

fib[1]=fib2;

System.out.println("Fibonacci Thread processing to generate Series of "+noOfElements+"
Elements...");

for(int i=2; i<noOfElements ; i++)

{
```

```
try

{

fib3=fib1+fib2;

fib[i]=fib3;

fib1=fib2;

fib2=fib3;

System.out.println("...");

Thread.sleep(1000);

}

catch(InterruptedException ie)

{

System.out.println(ie);

}

}

notify(); //Notify to Parent thread that Process is completed take over the control to process
further task

}

}

public void printFibonacci(int n)

{

System.out.println("Fibonacci Series: ");

for(int i=0; i<n ; i++)

{

System.out.print(fib[i]+"\t");

}

System.out.println();

}
```

```
}
```

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

In this example we have used **synchronized** keyword on the static method to perform static synchronization.

TestSynchronization4.java

```
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();
}
}
```

Example of static synchronization by Using the anonymous class

In this example, we are using anonymous class to create the threads.

TestSynchronization5.java Hello

```
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){}
   }
 }
}

public class TestSynchronization5 {
public static void main(String[] args) {

    Thread t1=new Thread(){
        public void run(){
            Table.printTable(1);
        }
    };

    Thread t2=new Thread(){
        public void run(){
            Table.printTable(10);
        }
    };

    Thread t3=new Thread(){
```

```
        public void run(){
            Table.printTable(100);
        }
    };

    Thread t4=new Thread(){
        public void run(){
            Table.printTable(1000);
        }
    };
    t1.start();
    t2.start();
    t3.start();
    t4.start();

}
}
```