

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.

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. 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.

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:

TestSynchronization1.java

```
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.

TestSynchronization2.java

```
//example of java synchronized method
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
```

Example of synchronized method by using anonymous class

In this program, we have created the two threads by using the anonymous class, so less coding is required.

TestSynchronization3.java

//Program of synchronized method by using anonymous class

```
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);}
        }
    }
}
```

```
public class TestSynchronization3{
    public static void main(String args[]){
        final Table obj = new Table();//only one object
```

```
        Thread t1=new Thread(){
            public void run(){
                obj.printTable(5);
            }
        };
        Thread t2=new Thread(){
            public void run(){
                obj.printTable(100);
            }
        };

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

Output:

5
10
15
20
25
100
200
300
400
500

[Next →](#)

AD

 **For Videos Join Our Youtube Channel: [Join Now](#)**

Feedback

- Send your Feedback to feedback@javatpoint.com

Help Others, Please Share



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.

Points to Remember

- Synchronized block is used to lock an object for any shared resource.
- Scope of synchronized block is smaller than the method.
- A Java synchronized block doesn't allow more than one JVM, to provide access control to a shared resource.
- The system performance may degrade because of the slower working of synchronized keyword.
- Java synchronized block is more efficient than Java synchronized method.

Syntax

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

Example of Synchronized Block

Let's see the simple example of synchronized block.

TestSynchronizedBlock1.java

```
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
```

Synchronized Block Example Using Anonymous Class

TestSynchronizedBlock2.java

```
// A Sender class
class Sender
{
    public void SenderMsg(String msg)
    {
        System.out.println("\nSending a Message: " + msg);
        try
        {
            Thread.sleep(800);
        }
        catch (Exception e)
        {
            System.out.println("Thread interrupted.");
        }
        System.out.println("\n" + msg + "Sent");
    }
}

// A Sender class for sending a message using Threads
class SenderWThreads extends Thread
{
    private String msg;
```

```
Sender sd;

// Receiver method to receive a message object and a message to be sent
SenderWThreads(String m, Sender obj)
{
    msg = m;
    sd = obj;
}

public void run()
{
    // Checks that only one thread sends a message at a time.
    synchronized(sd)
    {
        // synchronizing the sender object
        sd.SenderMsg(msg);
    }
}

// Driver Code
public class ShynchronizedMultithreading
{
    public static void main(String args[])
    {
        Sender sender = new Sender();
        SenderWThreads sender1 = new SenderWThreads( "Hola " , sender);
        SenderWThreads sender2 = new SenderWThreads( "Welcome to Javatpoint website " , sender);

        // Start two threads of SenderWThreads type
        sender1.start();
        sender2.start();

        // wait for threads to end
        try
        {
            sender1.join();
            sender2.join();
        }
```

```
catch(Exception e)
{
    System.out.println("Interrupted");
}
}
```

Output:

```
Sending a Message: Hola
Hola Sent
Sending a Message: Welcome to Javatpoint website
Welcome to Javatpoint website Sent
```

[< Prev](#)[Next >](#)

AD

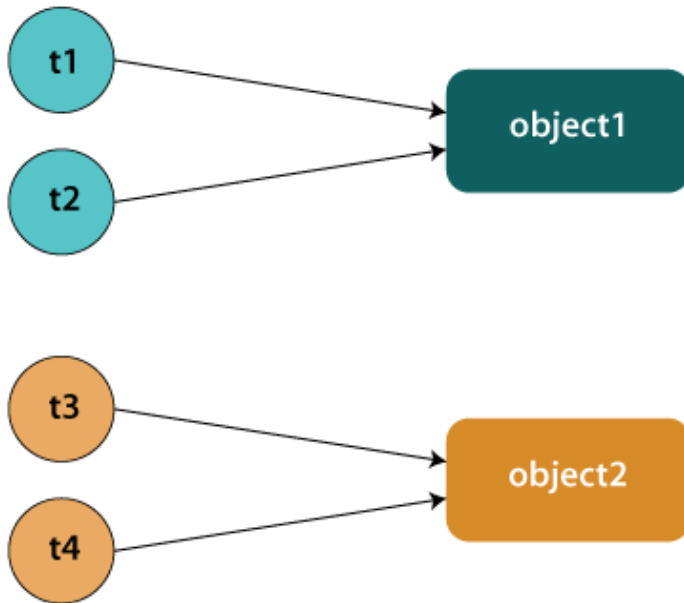
 Youtube **For Videos Join Our Youtube Channel: [Join Now](#)**

Feedback

- Send your Feedback to feedback@javatpoint.com

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);
        }
    }
}
```

```
        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();
    }
}
```

Test it Now**Output:**

```
1
2
3
4
5
6
7
8
9
10
10
20
30
40
50
60
70
80
90
100
100
200
300
400
500
600
700
800
900
1000
1000
2000
3000
4000
5000
6000
```

```
7000
8000
9000
10000
```

Example of static synchronization by Using the anonymous class

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

TestSynchronization5.javaHello

```
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();

}

}
```

Test it Now

Output:

```
1
2
3
4
5
6
7
8
9
10
10
20
30
40
```

```
50
60
70
80
90
100
100
200
300
400
500
600
700
800
900
1000
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
```

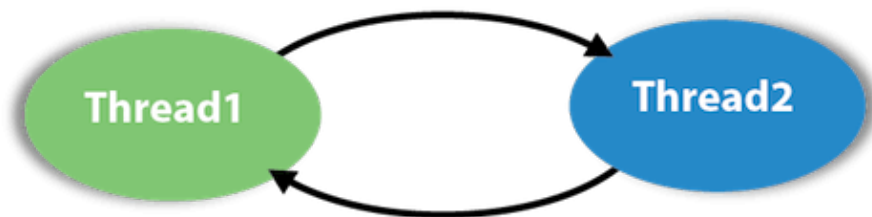
Synchronized block on a class lock:

The block synchronizes on the lock of the object denoted by the reference `.class` name `.class`. A static synchronized method `printTable(int n)` in class `Table` is equivalent to the following declaration:

```
static void printTable(int n) {
    synchronized (Table.class) {    // Synchronized block on class A
        // ...
    }
}
```

Deadlock in Java

Deadlock in Java is a part of multithreading. 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

TestDeadlockExample1.java

```
public class TestDeadlockExample1 {  
    public static void main(String[] args) {  
        final String resource1 = "ratan jaiswal";  
        final String resource2 = "vimal jaiswal";  
        // 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
```

More Complicated Deadlocks

A deadlock may also include more than two threads. The reason is that it can be difficult to detect a deadlock. Here is an example in which four threads have deadlocked:

Thread 1 locks A, waits for B

Thread 2 locks B, waits for C

Thread 3 locks C, waits for D

Thread 4 locks D, waits for A

Thread 1 waits for thread 2, thread 2 waits for thread 3, thread 3 waits for thread 4, and thread 4 waits for thread 1.

How to avoid deadlock?

A solution for a problem is found at its roots. In deadlock it is the pattern of accessing the resources A and B, is the main issue. To solve the issue we will have to simply re-order the statements where the code is accessing shared resources.

DeadlockSolved.java

```
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;
    }
}
```

```
}
```

Output:

```
In block 1  
In block 2
```

In the above code, class `DeadlockSolved` solves the deadlock kind of situation. It will help in avoiding deadlocks, and if encountered, in resolving them.

How to Avoid Deadlock in Java?

Deadlocks cannot be completely resolved. But we can avoid them by following basic rules mentioned below:

1. **Avoid Nested Locks:** We must avoid giving locks to multiple threads, this is the main reason for a deadlock condition. It normally happens when you give locks to multiple threads.
2. **Avoid Unnecessary Locks:** The locks should be given to the important threads. Giving locks to the unnecessary threads that cause the deadlock condition.
3. **Using Thread Join:** A deadlock usually happens when one thread is waiting for the other to finish. In this case, we can use **join** with a maximum time that a thread will take.

[< Prev](#)[Next >](#)

AD

 **For Videos Join Our Youtube Channel: [Join Now](#)**

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.

Method	Description
public final void wait()throws InterruptedException	It waits until object is notified.
public final void wait(long timeout)throws InterruptedException	It waits for the specified amount of time.

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()
```

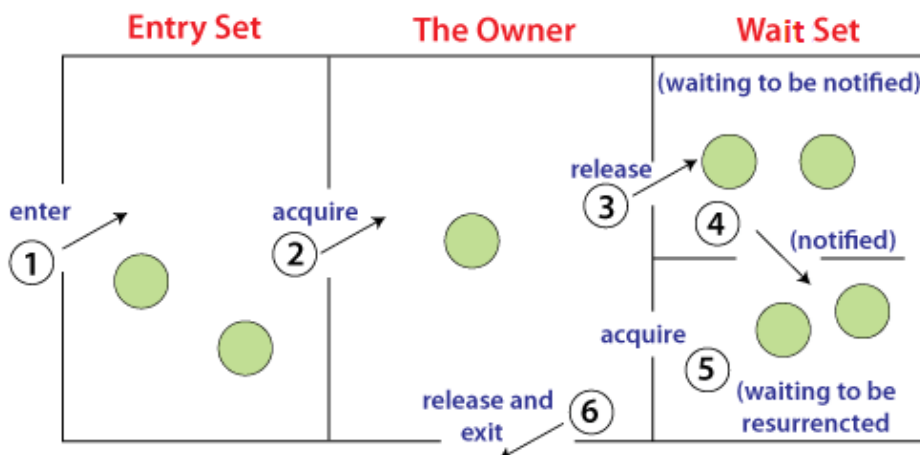

3) notifyAll() method

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

Syntax:

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

Understanding the process of inter-thread communication



The point to point explanation of the above diagram is as follows:

1. Threads enter to acquire lock.
2. Lock is acquired by on 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.

Why wait(), notify() and notifyAll() methods are defined in Object class not Thread class?

It is because they are related to lock and object has a lock.

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

Let's see the simple example of inter thread communication.

Test.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
```

[< Prev](#)[Next >](#)

AD

 Youtube For Videos Join Our Youtube Channel: [Join Now](#)

Interrupting a Thread:

If any thread is in sleeping or waiting state (i.e. `sleep()` or `wait()` is invoked), calling the `interrupt()` method on the thread, breaks out the sleeping or waiting state throwing `InterruptedException`. If the thread is not in the sleeping or waiting state, calling the `interrupt()` method performs normal behaviour and doesn't interrupt the thread but sets the interrupt flag to true. Let's first see the methods provided by the `Thread` class for thread interruption.

The 3 methods provided by the Thread class for interrupting a thread

- **`public void interrupt()`**
- **`public static boolean interrupted()`**
- **`public boolean isInterrupted()`**

Example of interrupting a thread that stops working

In this example, after interrupting the thread, we are propagating it, so it will stop working. If we don't want to stop the thread, we can handle it where `sleep()` or `wait()` method is invoked. Let's first see the example where we are propagating the exception.

TestInterruptingThread1.java

```
class TestInterruptingThread1 extends Thread{
    public void run(){
        try{
            Thread.sleep(1000);
            System.out.println("task");
        }catch(InterruptedException e){
            throw new RuntimeException("Thread interrupted..." + e);
        }
    }

    public static void main(String args[]){
        TestInterruptingThread1 t1 = new TestInterruptingThread1();
        t1.start();
        try{
            t1.interrupt();
        }
```

```
}catch(Exception e){System.out.println("Exception handled "+e);}

}
```

Test it Now

[download this example](#)

Output:

```
Exception in thread-0
    java.lang.RuntimeException: Thread interrupted...
    java.lang.InterruptedException: sleep interrupted
    at A.run(A.java:7)
```

Example of interrupting a thread that doesn't stop working

In this example, after interrupting the thread, we handle the exception, so it will break out the sleeping but will not stop working.

TestInterruptingThread2.java

```
class TestInterruptingThread2 extends Thread{
    public void run(){
        try{
            Thread.sleep(1000);
            System.out.println("task");
        }catch(InterruptedException e){
            System.out.println("Exception handled "+e);
        }
        System.out.println("thread is running...");
    }

    public static void main(String args[]){
        TestInterruptingThread2 t1=new TestInterruptingThread2();
        t1.start();

        t1.interrupt();
    }
}
```

```
}  
}
```

Test it Now

[download this example](#)

Output:

```
Exception handled  
    java.lang.InterruptedException: sleep interrupted  
    thread is running...
```

Example of interrupting thread that behaves normally

If thread is not in sleeping or waiting state, calling the `interrupt()` method sets the interrupted flag to true that can be used to stop the thread by the java programmer later.

TestInterruptingThread3.java

```
class TestInterruptingThread3 extends Thread{  
  
    public void run(){  
        for(int i=1;i<=5;i++){  
            System.out.println(i);  
        }  
  
        public static void main(String args[]){  
            TestInterruptingThread3 t1=new TestInterruptingThread3();  
            t1.start();  
  
            t1.interrupt();  
  
        }  
    }  
}
```

Test it Now

Output:

1

2

3

4

5



What about isInterrupted and interrupted method?

The `isInterrupted()` method returns the interrupted flag either true or false. The static `interrupted()` method returns the interrupted flag after that it sets the flag to false if it is true.

TestInterruptingThread4.java

```
public class TestInterruptingThread4 extends Thread{
```

```
    public void run(){
```

```
        for(int i=1;i<=2;i++){
```

```
            if(Thread.interrupted()){
```

```
                System.out.println("code for interrupted thread");
```

```
            }
```

```
        else{
```

```
            System.out.println("code for normal thread");
```

```
        }
```

```
    } //end of for loop
```

```
}
```

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

```
    TestInterruptingThread4 t1=new TestInterruptingThread4();
```

```
    TestInterruptingThread4 t2=new TestInterruptingThread4();
```

```
    t1.start();
```

```
    t1.interrupt();
```

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

Test it Now

Output:

```
Code for interrupted thread  
    code for normal thread  
    code for normal thread  
    code for normal thread
```

[download this example](#)

← Prev

Next →

AD

 **For Videos Join Our Youtube Channel: [Join Now](#)**

Feedback

- Send your Feedback to feedback@javatpoint.com

Reentrant Monitor in Java

According to Sun Microsystems, **Java monitors are reentrant** means java thread can reuse the same monitor for different synchronized methods if method is called from the method.

Advantage of Reentrant Monitor

It eliminates the possibility of single thread deadlocking

Let's understand the java reentrant monitor by the example given below:

```
class Reentrant {  
    public synchronized void m() {  
        n();  
        System.out.println("this is m() method");  
    }  
    public synchronized void n() {  
        System.out.println("this is n() method");  
    }  
}
```

In this class, m and n are the synchronized methods. The m() method internally calls the n() method.

Now let's call the m() method on a thread. In the class given below, we are creating thread using anonymous class.

```
public class ReentrantExample{  
    public static void main(String args[]){  
        final ReentrantExample re=new ReentrantExample();  
  
        Thread t1=new Thread(){  
            public void run(){  
                re.m();//calling method of Reentrant class  
            }  
        };  
        t1.start();  
    }  
}
```

```
}}
```

Test it Now

```
Output: this is n() method  
this is m() method
```

<<Prev

AD

Next>>

 **For Videos Join Our Youtube Channel: [Join Now](#)**




Feedback

- Send your Feedback to feedback@javatpoint.com

Help Others, Please Share



Learn Latest Tutorials

 **Splunk tutorial**
Splunk **SPSS tutorial**
SPSS **Swagger tutorial**
Swagger **T-SQL tutorial**
Transact-SQL