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. 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:

```
1. class Table{
2. void printTable(int n){//method not synchronized
3.   for(int i=1;i<=5;i++){
4.     System.out.println(n*i);
5.     try{
6.      Thread.sleep(400);
7.     }catch(Exception e){System.out.println(e);}
8.   }
9.
10.  }
11. }
12.
13. class MyThread1 extends Thread{
14. Table t;
15. MyThread1(Table t){
16. this.t=t;
17. }
18. public void run(){
19. t.printTable(5);
20. }
21.
22. }
23. 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.

```
1. //example of java synchronized method
2. class Table{
3.  synchronized void printTable(int n){//synchronized method
4.    for(int i=1;i<=5;i++){
5.      System.out.println(n*i);
6.      try{
7.       Thread.sleep(400);
8.      }catch(Exception e){System.out.println(e);}
```

```
9.     }
10.
11.  }
12. }
13.
14. class MyThread1 extends Thread{
15. Table t;
16. MyThread1(Table t){
17. this.t=t;
18. }
19. public void run(){
20. t.printTable(5);
21. }
22.
23. }
24. class MyThread2 extends Thread{
25. Table t;
26. MyThread2(Table t){
27. this.t=t;
28. }
29. public void run(){
30. t.printTable(100);
31. }
32. }
33.
34. public class TestSynchronization2{
35. public static void main(String args[]){
36. Table obj = new Table();//only one object
37. MyThread1 t1=new MyThread1(obj);
38. MyThread2 t2=new MyThread2(obj);
39. t1.start();
40. t2.start();
41. }
42. }
```

```
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 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
}
```

Example of synchronized block

Let's see the simple example of synchronized block.

Program of synchronized 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);}
     }
   }
 }//end of the method
}

class MyThread1 extends Thread{
Table t;
MyThread1(Table t){
this.t=t;
}
public void run(){
```

```
21. t.printTable(5);
22. }
23.
24. }
25. class MyThread2 extends Thread{
26. Table t;
27. MyThread2(Table t){
28. this.t=t;
29. }
30. public void run(){
31. t.printTable(100);
32. }
33. }
34.
35. public class TestSynchronizedBlock1{
36. public static void main(String args[]){
37. Table obj = new Table();//only one object
38. MyThread1 t1=new MyThread1(obj);
39. MyThread2 t2=new MyThread2(obj);
40. t1.start();
41. t2.start();
42. }
43. }
```

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

Suspending and Resuming Threads

Sometimes, suspending the execution of a thread is useful. For example, a separate thread can be used to display the time of day. If user does not want a clock, then its thread can be suspended. Whatever the case, suspending a thread is a simple matter. Once suspended, restarting the thread is also a simple matter.

The mechanisms to suspend, resume, and stop threads differ between early versions of Java, such as Java 1.0, and modern versions, beginning with Java 2.

Prior to Java 2, a program used the **suspend(), resume(),** and **stop()** methods, which are defined by the **Thread**, to pause, restart, and stop the execution of a thread. Although these methods seem to be a perfectly reasonable and convenient approach to managing the execution of threads, they must not be used for new Java programs.

Java suspend Thread

The **suspend()** method of **Thread** class was deprecated by Java 2 several years ago. This was done because the **suspend()** can sometimes cause serious system failures.

Assume that a thread has obtained locks on the critical data structures. If that thread is suspended at that point, those locks are not relinquished. Other threads that may be waiting for those resources can be deadlocked.

Java resume Thread

The **resume()** method is also deprecated. It doesn't cause problems, but cannot be used without the **suspend()** method as its counterpart.

Java stop Thread

The **stop()** method of **Thread** class, too, was deprecated by Java 2. This was done because this method can sometimes cause serious system failures.

Assume that a thread is writing to a critically important data structure and has completed only part of its changes. If that thread is stopped at that point, that data structure might be left in a corrupted state. The trouble is that, the **stop()** causes any lock, the calling thread holds, to be released. Thus, the corrupted data might be used by another thread that is waiting on the same lock.

Java suspend resume stop Thread Example

Because you cannot now use the **suspend(), resume()**, or **stop()** methods to control a thread, you might be thinking that no way exists to pause, restart, or terminate a thread. But, fortunately, this is not true. Instead, a thread must be designed so that the **run()** method periodically checks to determine whether that thread should suspend, resume, or stop its own execution. Typically, this is accomplished by

establishing a flag variable that indicates the execution state of the thread. As long as this flag is set to "running", **run()** method must continue to let the thread execute. If this variable is set to "suspend", the thread must pause. If it is set to "stop", the thread must terminate. Of course, a variety of ways exist in which to write such code, but the central theme will be same for all the programs.

The following example illustrates how **wait()** and **notify()** methods that are inherited from the **Object** can be used to control the execution of a thread. Let's consider its operation. The **NewThread** class contains a **boolean** instance variable named **suspendFlag** which is used to control the execution of the thread. It is initialized to **false** by the constructor. The **run()** method contains a **synchronized** statement block that checks **suspendFlag**. If that variable is **true**, the **wait()** method is invoked to suspend the execution of the thread.
The **mysuspend()** method sets **suspendFlag** to **true**. The **myresume()** method sets **suspendFlag** to **false** and invokes **notify()** to wake up the thread. Finally, the **main()** method has been modified to invoke the **mysuspend()** and **myresume()** methods.

```
/* Java Program Example - Java Suspend Resume Stop Thread
 * Suspending and resuming a thread the modern way */

 class NewThread implements Runnable
 {
     String name;       //name of thread
     Thread thr;
     boolean suspendFlag;

     NewThread(String threadname)
     {
         name = threadname;
         thr = new Thread(this, name);
         System.out.println("New thread : " + thr);
         suspendFlag = false;
         thr.start();     // start the thread
     }

     /* this is the entry point for thread */
     public void run()
     {
         try
         {
             for(int i=12; i>0; i--)
             {
                 System.out.println(name + " : " + i);
                 Thread.sleep(200);
```

```
                synchronized(this)
                {
                    while(suspendFlag)
                    {
                        wait();
                    }
                }
            }
        }
        catch(InterruptedException e)
        {
            System.out.println(name + " interrupted");
        }

        System.out.println(name + " exiting...");
    }

    synchronized void mysuspend()
    {
        suspendFlag = true;
    }

    synchronized void myresume()
    {
        suspendFlag = false;
        notify();
    }
}

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

        NewThread obj1 = new NewThread("One");
        NewThread obj2 = new NewThread("two");

        try
        {
            Thread.sleep(1000);
            obj1.mysuspend();
            System.out.println("Suspending thread One...");
            Thread.sleep(1000);
            obj1.myresume();
            System.out.println("Resuming thread One...");

            obj2.mysuspend();
            System.out.println("Suspending thread Two...");
            Thread.sleep(1000);
            obj2.myresume();
            System.out.println("Resuming thread Two...");
```

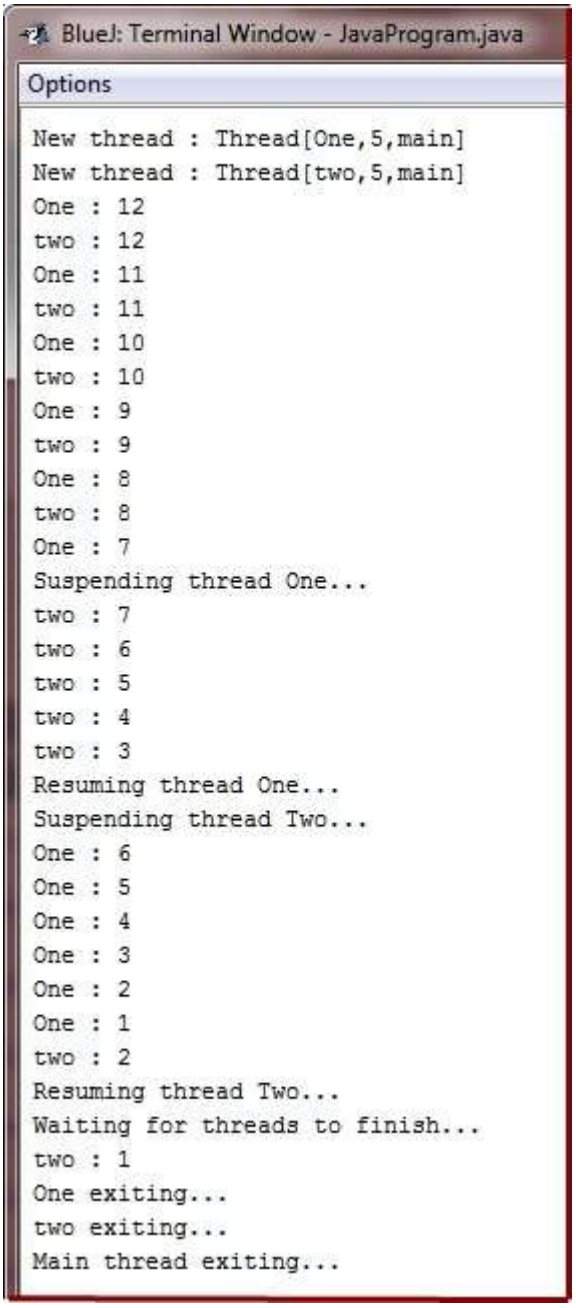
```
            }
            catch(InterruptedException e)
            {
                System.out.println("Main thread
Interrupted..!!");
            }

            /* wait for threads to finish */
            try
            {
                System.out.println("Waiting for threads to
finish...");
                obj1.thr.join();
                obj2.thr.join();
            }
            catch(InterruptedException e)
            {
                System.out.println("Main thread
Interrupted..!!");
            }

            System.out.println("Main thread exiting...");

        }
    }
```

When you compile and run the above Java program, you will see the threads suspend and resume as shown here :


BlueJ: Terminal Window - JavaProgram.java
Options
New thread : Thread[One,5,main]
New thread : Thread[two,5,main]
One : 12
two : 12
One : 11
two : 11
One : 10
two : 10
One : 9
two : 9
One : 8
two : 8
One : 7
Suspending thread One...
two : 7
two : 6
two : 5
two : 4
two : 3
Resuming thread One...
Suspending thread Two...
One : 6
One : 5
One : 4
One : 3
One : 2
One : 1
two : 2
Resuming thread Two...
Waiting for threads to finish...
two : 1
One exiting...
two exiting...
Main thread exiting...