UNIT -IV

Multithreading

Multithreading in java is a process of executing multiple threads simultaneously.

Thread is basically a lightweight sub-process, a smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking.

But we use multithreading than multiprocessing because threads share a common memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.

Java Multithreading is mostly used in games, animation etc

Advantage of Java Multithreading

1) It **doesn't block the user** because threads are independent and you can perform multiple operations at same time.

2) You **can perform many operations together so it saves time**.

3) Threads are **independent** so it doesn't affect other threads if exception occur in a single thread.

Multitasking

Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to utilize the CPU. Multitasking can be achieved by two ways:

- o Process-based Multitasking(Multiprocessing)
- o Thread-based Multitasking(Multithreading)

1) Process-based Multitasking (Multiprocessing)
- o Each process have its own address in memory i.e. each process allocates separate memory area.
- o Process is heavyweight.
- o Cost of communication between the process is high.

o Switching from one process to another require some time for saving and loading registers, memory maps, updating lists etc.

2) Thread-based Multitasking (Multithreading)
   o Threads share the same address space.
   o Thread is lightweight.
   o Cost of communication between the thread is low.

What is Thread in java

A thread is a lightweight sub process, a smallest unit of processing. It is a separate path of execution.

Threads are independent, if there occurs exception in one thread, it doesn't affect other threads. It shares a common memory area.
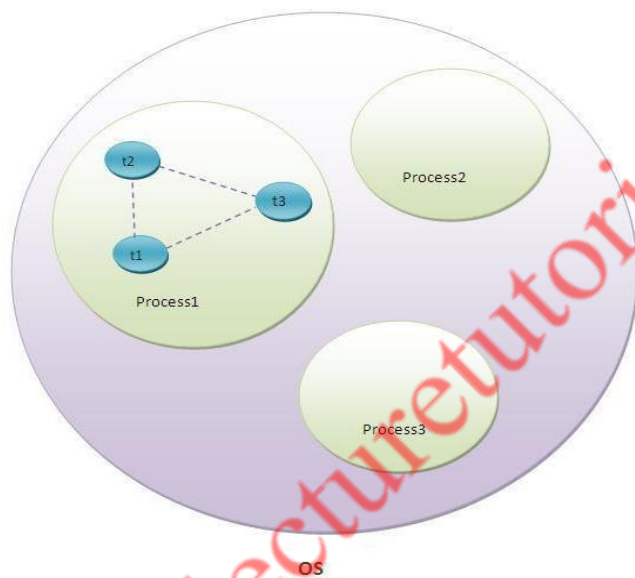

Life cycle of a Thread (Thread States)

A thread can be in one of the five states. According to sun, there is only 4 states in **thread life cycle in java** new, runnable, non-runnable and terminated. There is no running state.

But for better understanding the threads, we are explaining it in the 5 states.

The life cycle of the thread in java is controlled by JVM. The java thread states are as follows:

1. New
2. Runnable
3. Running
4. Non-Runnable (Blocked)z
5. Terminated
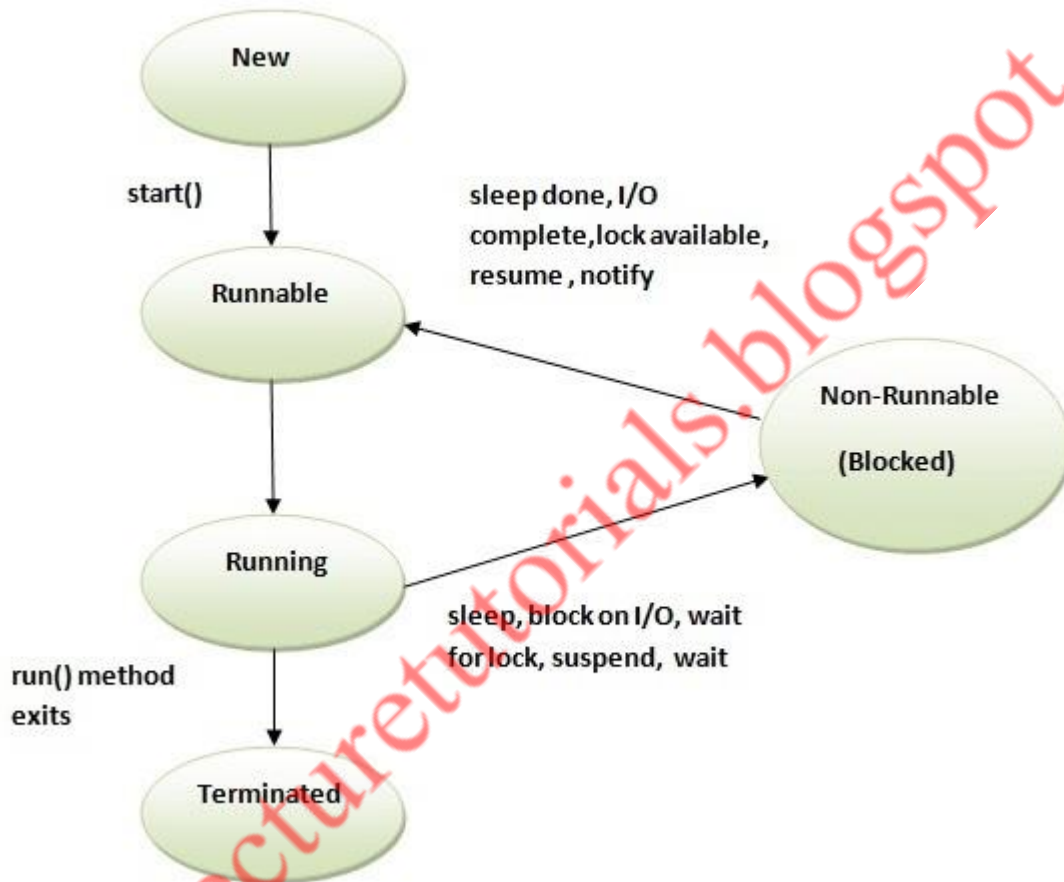
1) New

The thread is in new state if you create an instance of Thread class but before the invocation of start() method.

2) Runnable

The thread is in runnable state after invocation of start() method, but the thread scheduler has not selected it to be the running thread.

3) Running

The thread is in running state if the thread scheduler has selected it.

4) Non-Runnable (Blocked)

This is the state when the thread is still alive, but is currently not eligible to run.

5) Terminated

A thread is in terminated or dead state when its run() method exits.

How to create thread

There are two ways to create a thread:

1. By extending Thread class
2. By implementing Runnable interface.

How to create thread

There are two ways to create a thread:

1. By extending Thread class
2. By implementing Runnable interface.

Thread class:

Thread class provide constructors and methods to create and perform operations on a thread.Thread class extends Object class and implements Runnable interface.

Commonly used Constructors of Thread class:
- Thread()
- Thread(String name)
- Thread(Runnable r)
- Thread(Runnable r,String name)

Commonly used methods of Thread class:
1. **public void run():** is used to perform action for a thread.
2. **public void start():** starts the execution of the thread.JVM calls the run() method

on the thread.

3. **public void sleep(long miliseconds):** Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
4. **public void join():** waits for a thread to die.
5. **public void join(long miliseconds):** waits for a thread to die for the specified miliseconds.
6. **public int getPriority():** returns the priority of the thread.
7. **public int setPriority(int priority):** changes the priority of the thread.
8. **public String getName():** returns the name of the thread.
9. **public void setName(String name):** changes the name of the thread.
10. **public Thread currentThread():** returns the reference of currently executing thread.
11. **public int getId():** returns the id of the thread.
12. **public Thread.State getState():** returns the state of the thread.
13. **public boolean isAlive():** tests if the thread is alive.
14. **public void yield():** causes the currently executing thread object to temporarily pause and allow other threads to execute.
15. **public void suspend():** is used to suspend the thread(depricated).
16. **public void resume():** is used to resume the suspended thread(depricated).
17. **public void stop():** is used to stop the thread(depricated).
18. **public boolean isDaemon():** tests if the thread is a daemon thread.
19. **public void setDaemon(boolean b):** marks the thread as daemon or user thread.
20. **public void interrupt():** interrupts the thread.
21. **public boolean isInterrupted():** tests if the thread has been interrupted.
22. **public static boolean interrupted():** tests if the current thread has been interrupted.

Runnable interface:

The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. Runnable interface have only one method named run().

1. **public void run():** is used to perform action for a thread.

Starting a thread:

start() method of Thread class is used to start a newly created thread. It performs

following tasks:

- A new thread starts(with new callstack).
- The thread moves from New state to the Runnable state.
- When the thread gets a chance to execute, its target run() method will run.

1)By extending Thread class:

```
1.        class Multi extends Thread{
2.        public void run(){
3.        System.out.println("thread is running...");
4.        }
5.        public static void main(String args[]){
6.        Multi t1=new Multi();
7.        t1.start();
8.         }
9.         }
```
Output:thread is running...

Who makes your class object as thread object?

Thread class constructor allocates a new thread object.When you create object of Multi class,your class constructor is invoked(provided by Compiler) fromwhere Thread class constructor is invoked(by super() as first statement).So your Multi class object is thread object now.

2)By implementing the Runnable interface:

```
1.        class Multi3 implements Runnable{
2.        public void run(){
3.        System.out.println("thread is running...");
4.        }
5.
6.        public static void main(String args[]){
7.        Multi3 m1=new Multi3();
8.        Thread t1 =new Thread(m1);
9.        t1.start();
10.        }
11.        }
```
Output:thread is running...

If you are not extending the Thread class,your class object would not be treated as a thread object.So you need to explicitely create Thread class object.We are passing the object of your class that implements Runnable so that your class run() method may execute.

Priority of a Thread (Thread Priority):

Each thread have a priority. Priorities are represented by a number between 1 and 10. In most cases, thread schedular schedules the threads according to their priority (known as preemptive scheduling). But it is not guaranteed because it depends on JVM specification that which scheduling it chooses.

3 constants defiend in Thread class:

1. public static int MIN_PRIORITY
2. public static int NORM_PRIORITY
3. public static int MAX_PRIORITY

Default priority of a thread is 5 (NORM_PRIORITY). The value of MIN_PRIORITY is 1 and the value of MAX_PRIORITY is 10.

Example of priority of a Thread:

```
1.      class TestMultiPriority1 extends Thread{
2.       public void run(){
3.        System.out.println("running thread name is:"+Thread.currentThread().getName()
);
4.        System.out.println("running thread priority is:"+Thread.currentThread().getPriori
ty());
5.
6.       }
7.      public static void main(String args[]){
8.       TestMultiPriority1 m1=new TestMultiPriority1();
9.       TestMultiPriority1 m2=new TestMultiPriority1();
10.      m1.setPriority(Thread.MIN_PRIORITY);
11.      m2.setPriority(Thread.MAX_PRIORITY);
12.      m1.start();
13.      m2.start();
14.
15.      }
16.    }
```

Output:running thread name is:Thread-0

    running thread priority is:10

    running thread name is:Thread-1

    running thread priority is:1

Joining threads

Sometimes one thread needs to know when another thread is ending. In java, **isAlive()** and **join()** are two different methods to check whether a thread has finished its execution.

The **isAlive()** methods return **true** if the thread upon which it is called is still running otherwise it return **false**.

final boolean **isAlive()**

But, **join()** method is used more commonly than **isAlive()**. This method waits until the thread on which it is called terminates.

final void **join()** throws **InterruptedException**

Using **join()** method, we tell our thread to wait until the specifid thread completes its execution. There are overloaded versions of **join()** method, which allows us to specify time for which you want to wait for the specified thread to terminate.

final void **join**(long *milliseconds*) throws **InterruptedException**

Example of isAlive method

public class **MyThread** extends **Thread**

{

public void run()

  {

    System.out.println("r1 ");

```
   try{

    Thread.sleep(500);

  }catch(InterruptedException ie){}


    System.out.println("r2 ");

 }

public static void main(String[] args)

{

MyThread t1=new MyThread();

MyThread t2=new MyThread();

t1.start();

t2.start();

System.out.println(t1.isAlive());

System.out.println(t2.isAlive());

}

}
```

Output

r1

true

true

r1

r2

r2

Example of thread without `join()` method

```java
public class MyThread extends Thread
{
public void run()
  {
    System.out.println("r1 ");
    try{
     Thread.sleep(500);
  }catch(InterruptedException ie){}


    System.out.println("r2 ");
 }
public static void main(String[] args)
{
MyThread t1=new MyThread();
MyThread t2=new MyThread();
t1.start();
t2.start();
}
}
```

Output

r1

r1

r2

r2

In this above program two thread t1 and t2 are created. t1 starts first and after printing "r1" on console thread t1 goes to sleep for 500 mls.At the same time Thread t2 will start its

process and print "r1" on console and then goes into sleep for 500 mls. Thread t1 will wake up from sleep and print "r2" on console similarly thread t2 will wake up from sleep and print "r2" on console. So you will get output like r1 r1 r2 r2

Example of thread with join() method

```
public class MyThread extends Thread
{
public void run()
  {
    System.out.println("r1 ");
    try{
     Thread.sleep(500);
  }catch(InterruptedException ie){}


    System.out.println("r2 ");
 }
public static void main(String[] args)
{
MyThread t1=new MyThread();
MyThread t2=new MyThread();
t1.start();


try{
 t1.join();                  //Waiting for t1 to finish
}catch(InterruptedException ie){}


t2.start();
}
```

```
}
```

Output

r1

r2

r1

r2

In this above program join() method on thread t1 ensure that t1 finishes it process before thread t2 starts.

Specifying time with join()

If in the above program, we specify time while using **join()** with **m1**, then **m1** will execute for that time, and then **m2** and **m3** will join it.

m1.join(1500);

Doing so, initially m1 will execute for 1.5 seconds, after which m2 and m3 will join it.

In the last chapter we have seen the ways of naming thread in java. In this chapter we will be learning the different priorities that a thread can have.

Java Thread Priority :

1. Logically we can say that threads run simultaneously but practically its not true, only one Thread can run at a time in such a ways that user feels that concurrent environment.

2. Fixed priority scheduling algorithm is used to select one thread for execution based on priority.

Example #1 : Default Thread Priority

getPriority() method is used to get the priority of the thread.

```
package com.c4learn.thread;
```

```
public class ThreadPriority extends Thread {

 public void run() {
  System.out.println(Thread.currentThread().getPriority());
 }

 public static void main(String[] args)
              throws InterruptedException {

  ThreadPriority t1 = new ThreadPriority();
  ThreadPriority t2 = new ThreadPriority();

  t1.start();
  t2.start();

 }
}
```

Output :

```
5
5
```

default priority of the thread is 5.

Each thread has normal priority at the time of creation. We can change or modify the thread priority in the following example 2.

Example #2 : Setting Priority

```
package com.c4learn.thread;

public class ThreadPriority extends Thread {
```

```java
public void run() {

  String tName = Thread.currentThread().getName();
  Integer tPrio = Thread.currentThread().getPriority();

  System.out.println(tName + " has priority " + tPrio);
}

public static void main(String[] args)
           throws InterruptedException {

  ThreadPriority t0 = new ThreadPriority();
  ThreadPriority t1 = new ThreadPriority();
  ThreadPriority t2 = new ThreadPriority();

  t1.setPriority(Thread.MAX_PRIORITY);
  t0.setPriority(Thread.MIN_PRIORITY);
  t2.setPriority(Thread.NORM_PRIORITY);

  t0.start();
  t1.start();
  t2.start();

 }
}
```

Output :

Thread-0 has priority 1

Thread-2 has priority 5

Thread-1 has priority 10

Explanation of Thread Priority :

1. We can modify the thread priority using the **setPriority()** method.

2. Thread can have integer priority between 1 to 10

3. Java Thread class defines following constants –

4. At a time many thread can be ready for execution but the thread with highest priority is selected for execution

5. Thread have default priority equal to 5.

Thread : Priority and Constant

Thread Priority	Constant
MIN_PRIORITY	1
MAX_PRIORITY	10
NORM_PRIORITY	5

Java Thread Synchronization :

1. In multi-threading environment, When two or more threads need access to a any shared resource then their should be some mechanism to ensure that the resource will be used by only one thread at a time.

2. Thread Synchronization is a process by which this synchronization is achieved.

3. Thread Synchronization is used to prevent thread interference and consistency problem.

4. Thread Synchronization is achieved through keyword **synchronized**.

Types of Synchronization :
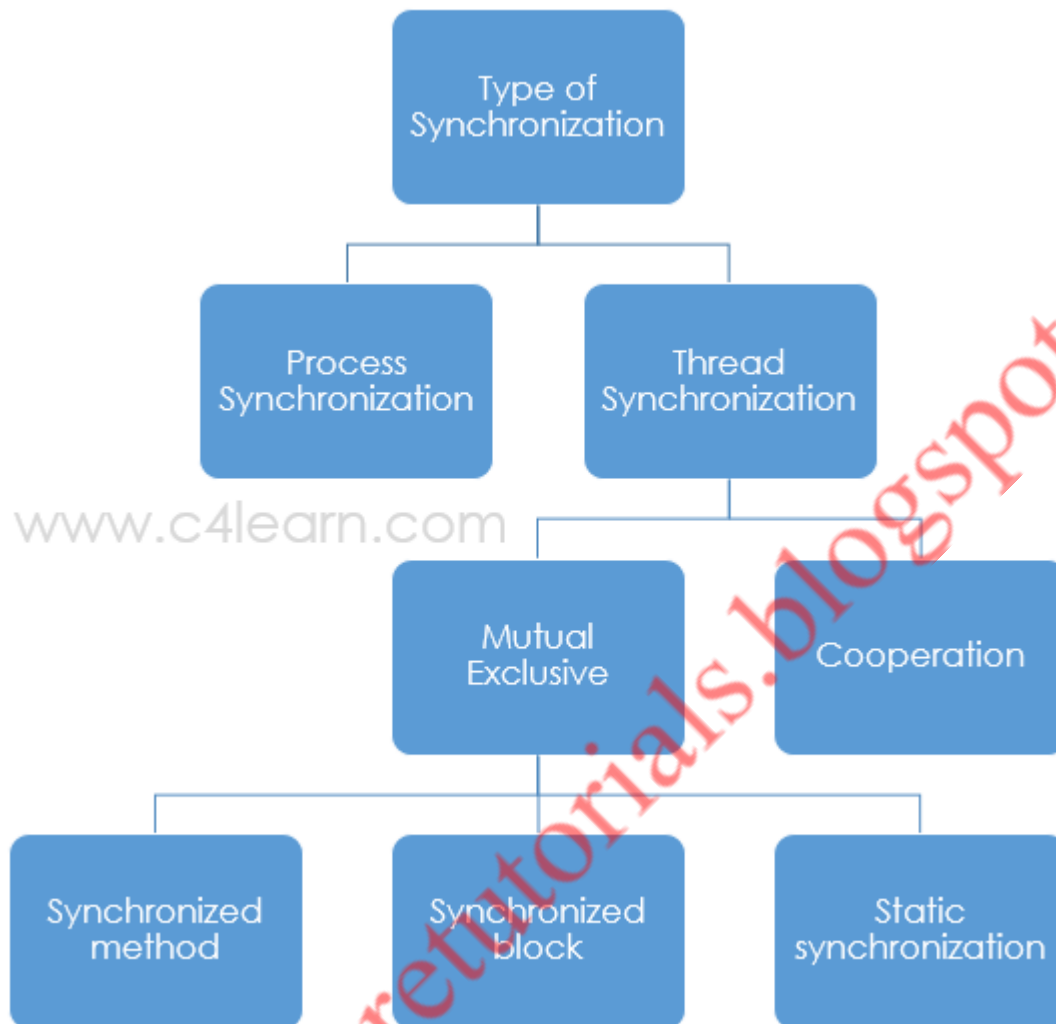
Synchronization in Java

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.
3.      void printTable(int n){//method not synchronized
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.        class TestSynchronization1{
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
       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
```

Example of synchronized method by using annonymous class

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

```
1.      //Program of synchronized method by using annonymous class
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.     public class TestSynchronization3{
15.     public static void main(String args[]){
16.     final Table obj = new Table();//only one object
17.
18.     Thread t1=new Thread(){
19.     public void run(){
20.     obj.printTable(5);
21.     }
22.     };
23.     Thread t2=new Thread(){
24.     public void run(){
25.     obj.printTable(100);
26.     }
27.     };
28.
29.     t1.start();
30.     t2.start();
31.     }
32.     }
```

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
```
1.      synchronized (object reference expression) {
2.        //code block
3.        }
```

Example of synchronized block

Let's see the simple example of synchronized block.

Program of synchronized block

```
1.      class Table{
2.
```

```java
3.          void printTable(int n){
4.           synchronized(this){//synchronized block
5.            for(int i=1;i<=5;i++){
6.             System.out.println(n*i);
7.             try{
8.              Thread.sleep(400);
9.             }catch(Exception e){System.out.println(e);}
10.           }
11.          }
12.        }//end of the method
13.       }
14.
15.       class MyThread1 extends Thread{
16.       Table t;
17.       MyThread1(Table t){
18.       this.t=t;
19.       }
20.       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
```

Same Example of synchronized block by using annonymous class:

//Program of synchronized block by using annonymous class

```
1.        class Table{
2.
3.        void printTable(int n){
4.          synchronized(this){//synchronized block
5.            for(int i=1;i<=5;i++){
6.             System.out.println(n*i);
7.             try{
8.              Thread.sleep(400);
9.             }catch(Exception e){System.out.println(e);}
10.           }
11.          }
12.        }//end of the method
13.        }
14.
15.        public class TestSynchronizedBlock2{
16.        public static void main(String args[]){
17.        final Table obj = new Table();//only one object
18.
19.        Thread t1=new Thread(){
20.        public void run(){
21.        obj.printTable(5);
22.        }
23.        };
24.        Thread t2=new Thread(){
25.        public void run(){
26.        obj.printTable(100);
27.        }
28.        };
29.
30.        t1.start();
31.        t2.start();
32.        }
```
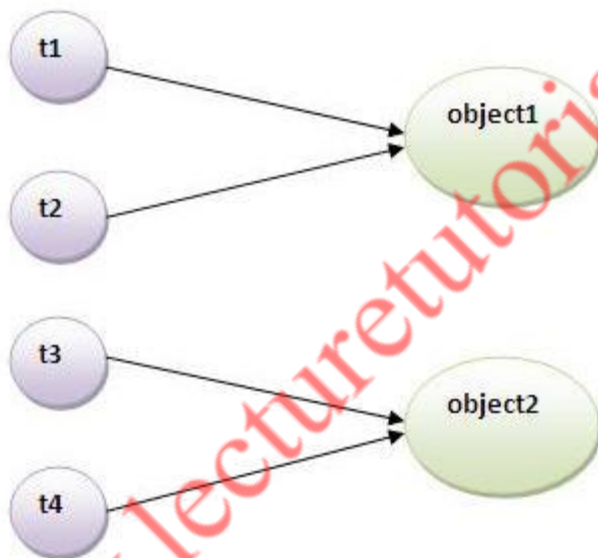
33.   }
      Output:5
          10
          15
          20
          25
          100
          200
          300
          400
          500

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.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.

```
1.        class Table{
2.
3.         synchronized static void printTable(int n){
4.          for(int i=1;i<=10;i++){
5.            System.out.println(n*i);
6.            try{
7.             Thread.sleep(400);
8.            }catch(Exception e){}
9.          }
10.       }
11.      }
12.
13.       class MyThread1 extends Thread{
14.       public void run(){
15.       Table.printTable(1);
16.       }
17.       }
18.
19.       class MyThread2 extends Thread{
20.       public void run(){
21.       Table.printTable(10);
22.       }
23.       }
24.
25.       class MyThread3 extends Thread{
26.       public void run(){
27.       Table.printTable(100);
28.       }
29.       }
30.       class MyThread4 extends Thread{
31.       public void run(){
32.       Table.printTable(1000);
33.       }
34.       }
35.
36.       public class TestSynchronization4{
37.       public static void main(String t[]){
38.       MyThread1 t1=new MyThread1();
39.       MyThread2 t2=new MyThread2();
40.       MyThread3 t3=new MyThread3();
```

```
41.        MyThread4 t4=new MyThread4();
42.        t1.start();
43.        t2.start();
44.        t3.start();
45.        t4.start();
46.        }
47.        }
```

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
```

Same example of static synchronization by annonymous class

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

```
1.      class Table{
2.
3.       synchronized static  void printTable(int n){
4.        for(int i=1;i<=10;i++){
5.         System.out.println(n*i);
6.         try{
7.          Thread.sleep(400);
8.         }catch(Exception e){}
9.        }
10.       }
11.      }
12.
13.      public class TestSynchronization5 {
14.      public static void main(String[] args) {
15.
16.       Thread t1=new Thread(){
17.        public void run(){
18.         Table.printTable(1);
19.        }
20.       };
21.
22.       Thread t2=new Thread(){
23.        public void run(){
24.         Table.printTable(10);
25.        }
26.       };
27.
```

```
28.          Thread t3=new Thread(){
29.            public void run(){
30.               Table.printTable(100);
31.            }
32.          };
33.
34.          Thread t4=new Thread(){
35.            public void run(){
36.               Table.printTable(1000);
37.            }
38.          };
39.          t1.start();
40.          t2.start();
41.          t3.start();
42.          t4.start();
43.
44.       }
45.    }
```

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:

```
1.        static void printTable(int n) {
2.          synchronized (Table.class) {      // Synchronized block on class A
3.            // ...
4.          }
5.        }
```

Introduction to Suspend Resume Thread

When the **sleep()** method time is over, the thread becomes implicitly active. **sleep()** method is preferable when the inactive time is known earlier. Sometimes, the inactive time or blocked time may not be known to the programmer earlier; to come to the task here comes **suspend()** method. The suspended thread will be in blocked state until **resume()** method is called on it. These methods are deprecated, as when not used

with precautions, the thread locks, if held, are kept in inconsistent state or may lead to deadlocks.

Note: You must have noticed, in the earlier **sleep()** method, that the thread in blocked state retains all its state. That is, attribute values remains unchanged by the time it comes into runnable state.

Suspend Resume Thread: Program explaining the usage of suspend() and resume() methods

```
1  public class SRDemo extends Thread
2  {
3    public static void main( String args[ ] )
4    {
5      SRDemo srd1 = new SRDemo();
6      SRDemo srd2 = new SRDemo();
7      srd1.setName("First");
8      srd2.setName("Second");
9      srd1.start();
10     srd2.start();
11     try
12     {
13       Thread.sleep( 1000 );
14       srd1.suspend();
15       System.out.println("Suspending thread First");
16       Thread.sleep( 1000 );
17       srd1.resume();
18       System.out.println("Resuming thread First");
```
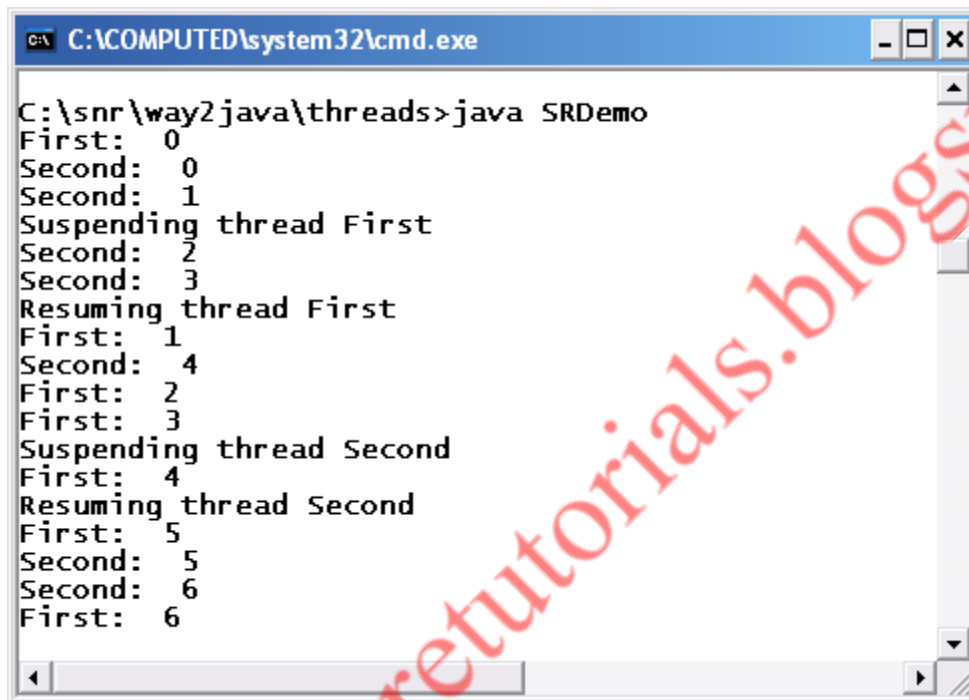
```
19
20    Thread.sleep(1000);
21    srd2.suspend();
22    System.out.println("Suspending thread Second");
23    Thread.sleep(1000);
24    srd2.resume();
25    System.out.println("Resuming thread Second");
26  }
27  catch(InterruptedException e)
28  {
29    e.printStackTrace();
30  }
31 }
32 public void run()
33 {
34  try
35  {
36   for(int i=0; i<7; i++)
37   {
38    Thread.sleep(500);
39    System.out.println( this.getName() + ":  " + i );
40   }
41  }
42  catch(InterruptedException e)
```

43 {

44   e.printStackTrace();

45 }

46 }

47 }

```
C:\COMPUTED\system32\cmd.exe                    _ □ ×

C:\snr\way2java\threads>java SRDemo
First:   0
Second:   0
Second:   1
Suspending thread First
Second:   2
Second:   3
Resuming thread First
First:   1
Second:   4
First:   2
First:   3
Suspending thread Second
First:   4
Resuming thread Second
First:   5
Second:   5
Second:   6
First:   6
```

Output screen on Suspend Resume Thread Java

Observe the screenshot, when no thread is suspended both threads are under execution. When**First** thread goes into suspended state, the **Second** thread goes into action. Similarly, when the**Second** goes to suspended state, the **First** is executed.

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

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

2) 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:
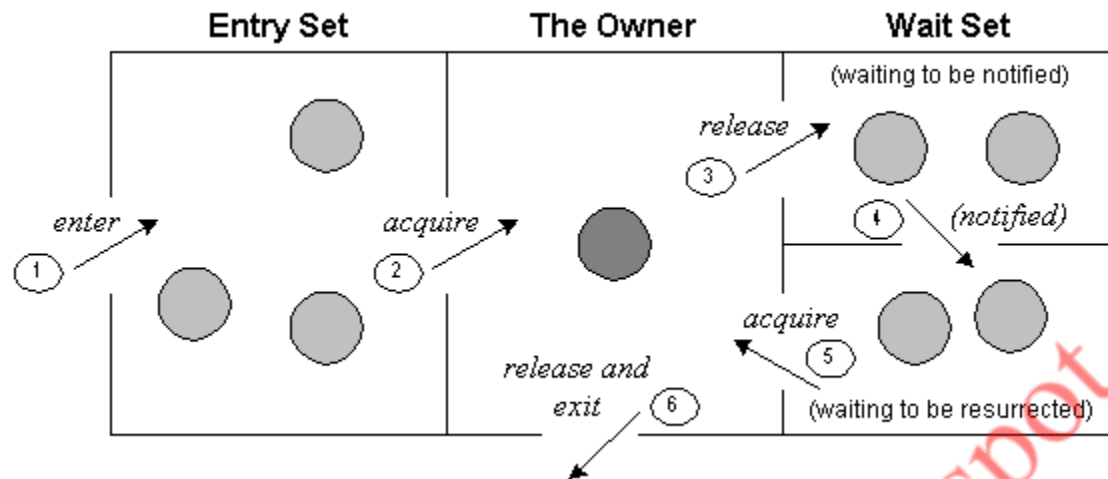
public final void notify()

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()
wait() method releases the lock	sleep() method doesn't release the lock.

is the method of Object class	is the method of Thread class
is the non-static method	is the static method
is the non-static method	is the static method
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.

```
1.        class Customer{
2.        int amount=10000;
3.
4.        synchronized void withdraw(int amount){
5.        System.out.println("going to withdraw...");
6.
7.        if(this.amount<amount){
8.        System.out.println("Less balance; waiting for deposit...");
9.        try{wait();}catch(Exception e){}
10.       }
11.       this.amount-=amount;
12.       System.out.println("withdraw completed...");
13.       }
14.
15.       synchronized void deposit(int amount){
16.       System.out.println("going to deposit...");
17.       this.amount+=amount;
18.       System.out.println("deposit completed... ");
19.       notify();
20.       }
21.       }
22.
23.       class Test{
24.       public static void main(String args[]){
25.       final Customer c=new Customer();
26.       new Thread(){
27.       public void run(){c.withdraw(15000);}
28.       }.start();
29.       new Thread(){
30.       public void run(){c.deposit(10000);}
```

```
31.         }.start();
32.
33.         }}
```
Output: going to withdraw...

Less balance; waiting for deposit...

going to deposit...

deposit completed...

withdraw completed

IO Stream

Java performs I/O through **Streams**. A Stream is linked to a physical layer by java I/O system to make input and output operation in java. In general, a stream means continuous flow of data. Streams are clean way to deal with input/output without having every part of your code understand the physical.

Java encapsulates Stream under **java.io** package. Java defines two types of streams. They are,

1. **Byte Stream :** It provides a convenient means for handling input and output of byte.

2. **Character Stream :** It provides a convenient means for handling input and output of characters. Character stream uses Unicode and therefore can be internationalized.

Byte Stream Classes

Byte stream is defined by using two abstract class at the top of hierarchy, they are InputStream and OutputStream.
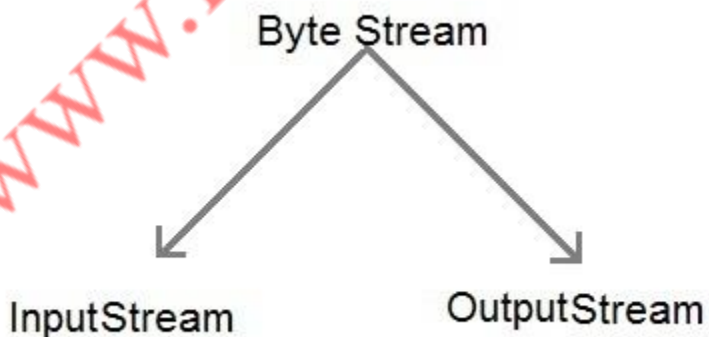
These two abstract classes have several concrete classes that handle various devices such as disk files, network connection etc.
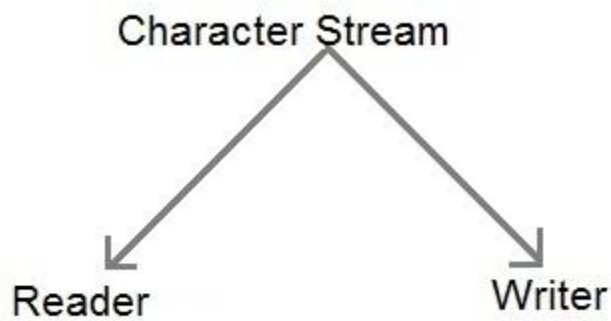

Some important Byte stream classes.

Stream class	Description
BufferedInputStream	Used for Buffered Input Stream.
BufferedOutputStream	Used for Buffered Output Stream.
DataInputStream	Contains method for reading java standard datatype
DataOutputStream	An output stream that contain method for writing java standard data type
FileInputStream	Input stream that reads from a file
FileOutputStream	Output stream that write to a file.
InputStream	Abstract class that describe stream input.
OutputStream	Abstract class that describe stream output.
PrintStream	Output Stream that contain print() and println() method

These classes define several key methods. Two most important are

1. read() : reads byte of data.
2. write() : Writes byte of data.

Character Stream Classes

Character stream is also defined by using two abstract class at the top of hierarchy, they are Reader and Writer.

These two abstract classes have several concrete classes that handle unicode character.

Some important Charcter stream classes.

Stream class	Description
BufferedReader	Handles buffered input stream.
BufferedWriter	Handles buffered output stream.
FileReader	Input stream that reads from file.
FileWriter	Output stream that writes to file.
InputStreamReader	Input stream that translate byte to character

OutputStreamReader	Output stream that translate character to byte.
PrintWriter	Output Stream that contain print() and println() method.
Reader	Abstract class that define character stream input
Writer	Abstract class that define character stream output

Reading Console Input

We use the object of BufferedReader class to take inputs from the keyboard.

Reading Characters

read() method is used with BufferedReader object to read characters. As this function returns integer type value has we need to use typecasting to convert it into **char** type.

int **read()** throws **IOException**

Below is a simple example explaining character input.

class CharRead

```
{
 public static void main( String args[])
 {
  BufferedReader br = new Bufferedreader(new InputstreamReader(System.in));
  char c = (char)br.read();      //Reading character
 }
}
```

Reading Strings

To read string we have to use readLine() function with BufferedReader class's object.

String **readLine()** throws **IOException**

Program to take String input from Keyboard in Java

```
import java.io.*;
class MyInput
{
 public static void main(String[] args)
 {
  String text;
  InputStreamReader isr = new InputStreamReader(System.in);
  BufferedReader br = new BufferedReader(isr);
  text = br.readLine();        //Reading String
  System.out.println(text);
 }
}
```

Program to read from a file using BufferedReader class

```
import java. Io *;
class ReadTest
{
 public static void main(String[] args)
 {
  try
  {
   File fl = new File("d:/myfile.txt");
   BufferedReader br = new BufferedReader(new FileReader(fl)) ;
   String str;
   while ((str=br.readLine())!=null)
   {
    System.out.println(str);
   }
   br.close();
   fl.close();
  }
  catch (IOException  e)
  { e.printStackTrace(); }
 }
}
```

Program to write to a File using FileWriter class

```
import java. Io *;
class WriteTest
{
 public static void main(String[] args)
```

```
{
 try
 {
  File fl = new File("d:/myfile.txt");
  String str="Write this string to my file";
  FileWriter fw = new FileWriter(fl) ;
  fw.write(str);
  fw.close();
  fl.close();
 }
 catch (IOException  e)
 { e.printStackTrace(); }
 }
}
```