Synchronization

At times when more than one thread try to access a shared resource, we need to ensure that resource will be used by only one thread at a time. The process by which this is achieved is called synchronization. The synchronization keyword in java creates a block of code referred to as critical section.

General Syntax:

synchronized (object)

{
//statement to be synchronized
}

Every Java object with a critical section of code gets a lock associated with the object. To enter critical section a thread need to obtain the corresponding object's lock.

Why we use Synchronization?

If we do not use synchronization, and let two or more threads access a shared resource at the same time, it will lead to distorted results.

Consider an example, suppose we have two different threads T1 and T2, T1 starts execution and save certain values in a file *temporary.txt* which will be used to calculate some result when T1 returns. Meanwhile, T2 starts and before T1 returns, T2 change the values saved by T1 in the file temporary.txt (temporary.txt is the shared resource). Now obviously T1 will return wrong result.

To prevent such problems, synchronization was introduced. With synchronization in above case, once T1 starts using *temporary.txt* file, this file will be locked (LOCK mode), and no other thread will be able to access or modify it until T1 returns.

Using Synchronized Methods

Using Synchronized methods is a way to accomplish synchronization. But lets first see what happens when we do not use synchronization in our program.

Example with no Synchronization

```
class First
{
 public void display(String msg)
 {
  System.out.print ("["+msg);
  try
  {
   Thread.sleep(1000);
  }
  catch(InterruptedException e)
  {
   e.printStackTrace();
  }
  System.out.println ("]");
 }
}

class Second extends Thread
{
```

```
 String msg;
 First fobj;
 Second (First fp,String str)
 {
  fobj = fp;
  msg = str;
  start();
 }
 public void run()
 {
  fobj.display(msg);
 }
}

public class Syncro
{
 public static void main (String[] args)
 {
  First fnew = new First();
  Second ss = new Second(fnew, "welcome");
  Second ss1= new Second (fnew,"new");
  Second ss2 = new Second(fnew, "programmer");
 }
}
```

```
[welcome [ new [ programmer]
]
]
```

In the above program, object fnew of class First is shared by all the three running threads(ss, ss1 and ss2) to call the shared method(*void* display). Hence the result is unsynchronized and such situation is called Race condition.

Synchronized Keyword
To synchronize above program, we must *synchronize* access to the shared display() method, making it available to only one thread at a time. This is done by using
keyword synchronized with display() method.

synchronized void display (String msg)

Using Synchronized block
If you have to synchronize access to an object of a class or you only want a part of a method to be synchronized to an object then you can use synchronized block for it.

```
class First
{
 public void display(String msg)
 {
  System.out.print ("["+msg);
  try
  {
   Thread.sleep(1000);
```

```
  }
  catch(InterruptedException e)
  {
   e.printStackTrace();
  }
  System.out.println ("]");
 }
}

class Second extends Thread
{
 String msg;
 First fobj;
 Second (First fp,String str)
 {
  fobj  =  fp;
  msg = str;
  start();
 }
 public void run()
 {
synchronized(fobj)       //Synchronized block
{
   fobj.display(msg);
}
 }
}


public class Syncro
{
 public static void main (String[] args)
 {
  First fnew = new First();
  Second ss = new Second(fnew, "welcome");
  Second ss1= new Second (fnew,"new");
  Second ss2 = new Second(fnew, "programmer");
 }
}
```

```
[welcome]
[new]
[programmer]
```

Because of synchronized block this program gives the expected output.

Difference between synchronized keyword and synchronized block

When we use synchronized keyword with a method, it acquires a lock in the object for the whole method. It means that no other thread can use any synchronized method until the current thread, which has invoked it's synchronized method, has finished its execution. Synchronized block acquires a lock in the object only between parentheses after the synchronized keyword. This means that no other thread can acquire a lock on the locked object until the synchronized block exits. But other threads can access the rest of the code of the method.

Which is more preferred - Synchronized method or synchronized block?
In Java, synchronized keyword causes a performance cost. A synchronized method in Java is very slow and can degrade performance. So we must use synchronization keyword in java when it is necessary else, we should use Java synchronized block that is used for synchronizing critical section only.

Inter Thread Communication

Java provides benefits of avoiding thread pooling using inter-thread communication.
The wait(), notify(), and notifyAll() methods of Object class are used for this purpose. These methods are implemented as final methods in Object, so that all classes have them. All the three method can be called only from within a synchronized context.

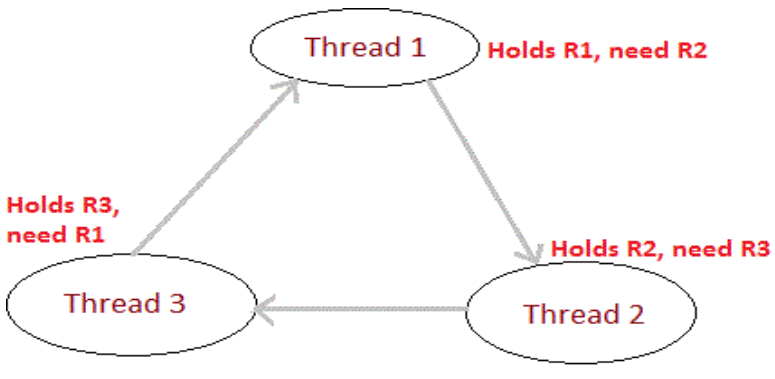
- wait() tells calling thread to give up monitor and go to sleep until some other thread enters the same monitor and call notify.
- notify() wakes up a thread that called wait() on same object.
- notifyAll() wakes up all the thread that called wait() on same object.

Difference between wait() and sleep()

wait()	sleep()
called from synchronized block	no such requirement
monitor is released	monitor is not released
gets awake when notify() or notifyAll() method is called.	does not get awake when notify() or notifyAll() method is called
not a static method	static method
wait() is generaly used on condition	sleep() method is simply used to put your thread on sleep.

Thread Pooling
Pooling is usually implemented by loop i.e. to check some condition repeatedly. Once condition is true appropriate action is taken. This wastes CPU time.

Deadlock



Deadlock Condition

Deadlock is a situation of complete Lock, when no thread can complete its execution because lack of resources. In the above picture, Thread 1 is holding a resource R1, and need another resource R2 to finish execution, but R2 is locked by Thread 2, which needs R3, which in turn

is locked by Thread 3. Hence none of them can finish and are stuck in a deadlock.

ThreadGroup in Java

Java provides a convenient way to group multiple threads in a single object. In such way, we can suspend, resume or interrupt group of threads by a single method call.

Note: Now suspend(), resume() and stop() methods are deprecated.

Java thread group is implemented by *java.lang.ThreadGroup* class.

Constructors of ThreadGroup class

There are only two constructors of ThreadGroup class.

No.	Constructor	Description
1)	ThreadGroup(String name)	creates a thread group with given name.
2)	ThreadGroup(ThreadGroup parent, String name)	creates a thread group with given parent group and name.

Important methods of ThreadGroup class

There are many methods in ThreadGroup class. A list of important methods are given below.

No.	Method	Description
1)	int activeCount()	returns no. of threads running in current group.
2)	int activeGroupCount()	returns a no. of active group in this thread group.
3)	void destroy()	destroys this thread group and all its sub groups.
4)	String getName()	returns the name of this group.
5)	ThreadGroup getParent()	returns the parent of this group.
6)	void interrupt()	interrupts all threads of this group.
7)	void list()	prints information of this group to standard console.

Let's see a code to group multiple threads.

```
1. ThreadGroup tg1 = new ThreadGroup("Group A");
2. Thread t1 = new Thread(tg1,new MyRunnable(),"one");
3. Thread t2 = new Thread(tg1,new MyRunnable(),"two");
4. Thread t3 = new Thread(tg1,new MyRunnable(),"three");
```

Now all 3 threads belong to one group. Here, tg1 is the thread group name, MyRunnable is the class that implements Runnable interface and "one", "two" and "three" are the thread names.

Now we can interrupt all threads by a single line of code only.

```
1. Thread.currentThread().getThreadGroup().interrupt();
```

ThreadGroup Example

File: ThreadGroupDemo.java

```
public class ThreadGroupDemo implements Runnable{
    public void run() {
        System.out.println(Thread.currentThread().getName());
    }
    public static void main(String[] args) {
      ThreadGroupDemo runnable = new ThreadGroupDemo();
        ThreadGroup tg1 = new ThreadGroup("Parent ThreadGroup");

        Thread t1 = new Thread(tg1, runnable,"one");
        t1.start();
        Thread t2 = new Thread(tg1, runnable,"two");
        t2.start();
        Thread t3 = new Thread(tg1, runnable,"three");
        t3.start();

        System.out.println("Thread Group Name: "+tg1.getName());
```

```
            tg1.list();

        }
       }
```
Output:
```
one
two
three
Thread Group Name: Parent ThreadGroup
java.lang.ThreadGroup[name=Parent ThreadGroup,maxpri=10]
   Thread[one,5,Parent  ThreadGroup]
   Thread[two,5,Parent  ThreadGroup]
   Thread[three,5,Parent ThreadGroup]
```