Synchronized in Java

[Multi-threaded](http://quiz.geeksforgeeks.org/multithreading-in-java/)programs may often come to a situation where multiple threads try to access the same resources and finally produce erroneous and unforeseen results.

So it needs to be made sure by some synchronization method that only one thread can access the resource at a given point of time.

Java provides a way of creating threads and synchronizing their task by using synchronized blocks. Synchronized blocks in Java are marked with the synchronized keyword. A synchronized block in Java is synchronized on some object. All synchronized blocks synchronized on the same object can only have one thread executing inside them at a time. All other threads attempting to enter the synchronized block are blocked until the thread inside the synchronized block exits the block.

Following is the general form of a synchronized block:

// Only one thread can execute at a time.

// sync\_object is a reference to an object

// whose lock associates with the [monitor](http://quiz.geeksforgeeks.org/monitors/).

// The code is said to be synchronized on

// the monitor object

synchronized(sync\_object)

{

// Access shared variables and other

// shared resources

}

This synchronization is implemented in Java with a concept called monitors. Only one thread can own a monitor at a given time. When a thread acquires a lock, it is said to have entered the monitor. All other threads attempting to enter the locked monitor will be suspended until the first thread exits the monitor.

Following is an example of multi threading with synchronized.

|  |
| --- |
| // A Java program to demonstrate working of  // synchronized.  import java.io.\*;  import java.util.\*;    // A Class used to send a message  class Sender  {      public void send(String msg)      {          System.out.println("Sending\t"  + msg );          try          {              Thread.sleep(1000);          }          catch (Exception e)          {              System.out.println("Thread  interrupted.");          }          System.out.println("\n" + msg + "Sent");      }  }    // Class for send a message using Threads  class ThreadedSend extends Thread  {      private String msg;      private Thread t;      Sender  sender;        // Recieves a message object and a string      // message to be sent      ThreadedSend(String m,  Sender obj)      {          msg = m;          sender = obj;      }        public void run()      {          // Only one thread can send a message          // at a time.          synchronized(sender)          {              // synchronizing the snd object              sender.send(msg);          }      }  }    // Driver class  class SyncDemo  {      public static void main(String args[])      {          Sender snd = new Sender();          ThreadedSend S1 =              new ThreadedSend( " Hi " , snd );          ThreadedSend S2 =              new ThreadedSend( " Bye " , snd );            // Start two threads of ThreadedSend type          S1.start();          S2.start();            // wait for threads to end          try          {              S1.join();              S2.join();          }          catch(Exception e)          {              System.out.println("Interrupted");          }      }  } |

Run on IDE

Output:

Sending Hi

Hi Sent

Sending Bye

Bye Sent

The output is same every-time we run the program.

In the above example, we chose to synchronize the Sender object inside the run() method of the ThreadedSend class. Alternately, we could define the **whole send() block as synchronized**and it would produce the same result. Then we don’t have to synchronize the Message object inside the run() method in ThreadedSend class.

|  |
| --- |
| // An alternate implementation to demonstrate  // that we can use synchronized with method also.  class Sender  {      public synchronized void send(String msg)      {          System.out.println("Sending\t" + msg );          try          {              Thread.sleep(1000);          }          catch (Exception e)          {              System.out.println("Thread interrupted.");          }          System.out.println("\n" + msg + "Sent");      }  } |

Run on IDE

We do not always have to synchronize a whole method. Sometimes it is preferable to **synchronize only part of a method**. Java synchronized blocks inside methods makes this possible.

|  |
| --- |
| // One more alternate implementation to demonstrate  // that synchronized can be used with only a part of  // method  class Sender  {      public void send(String msg)      {          synchronized(this)          {              System.out.println("Sending\t" + msg );              try              {                  Thread.sleep(1000);              }              catch (Exception e)              {                  System.out.println("Thread interrupted.");              }              System.out.println("\n" + msg + "Sent");          }      }  } |

**Java synchronized keyword**

By Lokesh Gupta | Filed Under: [Multi Threading](https://howtodoinjava.com/java/multi-threading/)

**Java synchronized keyword** marks a block or method a critical section. A critical section is where one and only one thread is executing at a time, and the thread holds the lock for the synchronized section.

**synchronized** keyword helps in writing [concurrent](https://howtodoinjava.com/java-concurrency-tutorial/) parts of the applications, to protect shared resources within this block.

The synchronized keyword can be use with –

* a code block
* a method

**1. Java synchronized block**

**1.1. Syntax**

The general syntax for writing a synchronized block is as follows. Here **lockObject** is a reference to an object whose lock associates with the monitor that the synchronized statements represent.

|  |
| --- |
| Syntax |
| synchronized( lockObject )  {     // synchronized statements  } |

**1.2. Internal working**

When a thread wants to execute synchronized statements inside the synchronized block, it MUST acquire the lock on lockObject‘s monitor. At a time, only one thread can acquire the monitor of a lock object. So all other threads must wait till this thread, currently acquired the lock, finish it’s execution.

In this way, synchronized keyword guarantees that only one thread will be executing the synchronized block statements at a time, and thus prevent multiple threads from corrupting the shared data inside the block.

Keep in mind that if a thread is put on sleep (using sleep() method) then it does not release the lock. At this sleeping time, no thread will be executing the synchronized block statements.

Java synchronization will throw **NullPointerException** if lock object used in 'synchronized (lock)' is null.

**1.3. Java synchronized block example**

Java program to demonstrate the usage of synchronized block. In given example, we have a MathClass with a method printNumbers(). This method will print the numbers starting from 1 to the argument number N.

Notice that the code in printNumbers() method is inside synchronized block.

|  |
| --- |
| MathClass.java |
| public class MathClass  {      void printNumbers(int n) throws InterruptedException      {          synchronized (this)          {              for (int i = 1; i <= n; i++)              {                  System.out.println(Thread.currentThread().getName() + " :: "+  i);                  Thread.sleep(500);              }          }      }  } |

I have created two threads which start executing the printNumbers() method exactly at same time. Due to block being synchronized, only one thread is allowed access and other thread has to wait until first thread is finished.

|  |
| --- |
| Main.java |
| public class Main  {      public static void main(String args[])      {          final MathClass mathClass = new MathClass();            //first thread          Runnable r = new Runnable()          {              public void run()              {                  try {                      mathClass.printNumbers(3);                  } catch (InterruptedException e) {                      e.printStackTrace();                  }              }          };            new Thread(r, "ONE").start();          new Thread(r, "TWO").start();      }  } |

Program output.

|  |
| --- |
| Console |
| ONE :: 1  ONE :: 2  ONE :: 3    TWO :: 1  TWO :: 2  TWO :: 3 |

**2. Java synchronized method**

**2.1. Syntax**

The general syntax for writing a synchronized method is as follows. Here **lockObject** is a reference to an object whose lock associates with the monitor that the synchronized statements represent.

|  |
| --- |
| Syntax |
| <access modifier> synchronized method( parameters )  {      // synchronized code  } |

**2.2. Internal working**

Similar to synchronized block, a thread MUST acquire the lock on the associated monitor object with synchronized method. In case of synchronized method, the lock object is –

* **‘.class’ object** – if the method is static.
* **‘this’ object** – if the method is not static. ‘this’ refer to reference to current object in which synchronized method is invoked.

Read More : [Object level lock vs Class level lock in Java](https://howtodoinjava.com/java/multi-threading/object-vs-class-level-locking/)

Java synchronized keyword is **re-entrant** in nature it means if a synchronized method calls another synchronized method which requires same lock then current thread which is holding lock can enter into that method without acquiring lock.

**2.3. Java synchronized method example**

Similar to synchronized block example, we can apply synchronized keyword at printNumber()method and it will make the method as synchronized. Now if we again run the example, we will get the similar output.

|  |
| --- |
| MathClass.java |
| public class MathClass  {      synchronized void printNumbers(int n) throws InterruptedException      {          for (int i = 1; i <= n; i++)          {              System.out.println(Thread.currentThread().getName() + " :: "+  i);              Thread.sleep(500);          }      }  } |

Program output.

|  |
| --- |
| Console |
| ONE :: 1  ONE :: 2  ONE :: 3    TWO :: 1  TWO :: 2  TWO :: 3 |

Drop me your questions in comments.

Happy Learning !!