**Starvation and Fairness**

* [Causes of Starvation in Java](http://tutorials.jenkov.com/java-concurrency/starvation-and-fairness.html#starvation)
  + [Threads with high priority swallow all CPU time from threads with lower priority](http://tutorials.jenkov.com/java-concurrency/starvation-and-fairness.html#high-priority)
  + [Threads are blocked indefinitely waiting to enter a synchronized block](http://tutorials.jenkov.com/java-concurrency/starvation-and-fairness.html#synchronized)
  + [Threads waiting on an object (called wait() on it) remain waiting indefinitely](http://tutorials.jenkov.com/java-concurrency/starvation-and-fairness.html#wait)
* [Implementing Fairness in Java](http://tutorials.jenkov.com/java-concurrency/starvation-and-fairness.html#fairness)
  + [Using Locks Instead of Synchronized Blocks](http://tutorials.jenkov.com/java-concurrency/starvation-and-fairness.html#locks)
  + [A Fair Lock](http://tutorials.jenkov.com/java-concurrency/starvation-and-fairness.html#fairlock)
  + [A Note on Performance](http://tutorials.jenkov.com/java-concurrency/starvation-and-fairness.html#performance)

|  |  |
| --- | --- |
|  | Jakob Jenkov Last update: 2014-06-23 |

If a thread is not granted CPU time because other threads grab it all, it is called "starvation". The thread is "starved to death" because other threads are allowed the CPU time instead of it. The solution to starvation is called "fairness" - that all threads are fairly granted a chance to execute.

**Causes of Starvation in Java**

The following three common causes can lead to starvation of threads in Java:

1. Threads with high priority swallow all CPU time from threads with lower priority.
2. Threads are blocked indefinately waiting to enter a synchronized block, because other threads are constantly allowed access before it.
3. Threads waiting on an object (called wait() on it) remain waiting indefinitely because other threads are constantly awakened instead of it.

**Threads with high priority swallow all CPU time from threads with lower priority**

You can set the thread priority of each thread individually. The higher the priority the more CPU time the thread is granted. You can set the priority of threads between 1 and 10. Exactly how this is interpreted depends on the operating system your application is running on. For most applications you are better off leaving the priority unchanged.

**Threads are blocked indefinitely waiting to enter a synchronized block**

Java's synchronized code blocks can be another cause of starvation. Java's synchronized code block makes no guarantee about the sequence in which threads waiting to enter the synchronized block are allowed to enter. This means that there is a theoretical risk that a thread remains blocked forever trying to enter the block, because other threads are constantly granted access before it. This problem is called "starvation", that a thread is "starved to death" by because other threads are allowed the CPU time instead of it.

**Threads waiting on an object (called wait() on it) remain waiting indefinitely**

The notify() method makes no guarantee about what thread is awakened if multiple thread have called wait() on the object notify() is called on. It could be any of the threads waiting. Therefore there is a risk that a thread waiting on a certain object is never awakened because other waiting threads are always awakened instead of it.

**Implementing Fairness in Java**

While it is not possible to implement 100% fairness in Java we can still implement our synchronization constructs to increase fairness between threads.

First lets study a simple synchronized code block:

public class Synchronizer{

public synchronized void doSynchronized(){

//do a lot of work which takes a long time

}

}

If more than one thread call the doSynchronized() method, some of them will be blocked until the first thread granted access has left the method. If more than one thread are blocked waiting for access there is no guarantee about which thread is granted access next.

**Using Locks Instead of Synchronized Blocks**

To increase the fairness of waiting threads first we will change the code block to be guarded by a lock rather than a synchronized block:

public class Synchronizer{

Lock lock = new Lock();

public void doSynchronized() throws InterruptedException{

this.lock.lock();

//critical section, do a lot of work which takes a long time

this.lock.unlock();

}

}

Notice how the doSynchronized() method is no longer declared synchronized. Instead the critical section is guarded by the lock.lock() and lock.unlock() calls.

A simple implementation of the Lock class could look like this:

public class Lock{

private boolean isLocked = false;

private Thread lockingThread = null;

public synchronized void lock() throws InterruptedException{

while(isLocked){

wait();

}

isLocked = true;

lockingThread = Thread.currentThread();

}

public synchronized void unlock(){

if(this.lockingThread != Thread.currentThread()){

throw new IllegalMonitorStateException(

"Calling thread has not locked this lock");

}

isLocked = false;

lockingThread = null;

notify();

}

}

If you look at the Synchronizer class above and look into this Lock implementation you will notice that threads are now blocked trying to access the lock() method, if more than one thread calls lock() simultanously. Second, if the lock is locked, the threads are blocked in the wait() call inside the while(isLocked) loop in the lock() method. Remember that a thread calling wait() releases the synchronization lock on the Lock instance, so threads waiting to enter lock() can now do so. The result is that multiple threads can end up having called wait() inside lock().

If you look back at the doSynchronized() method you will notice that the comment between lock() and unlock() states, that the code in between these two calls take a "long" time to execute. Let us further assume that this code takes long time to execute compared to entering the lock() method and calling wait() because the lock is locked. This means that the majority of the time waited to be able to lock the lock and enter the critical section is spent waiting in the wait() call inside the lock() method, not being blocked trying to enter the lock() method.

As stated earlier synchronized blocks makes no guarantees about what thread is being granted access if more than one thread is waiting to enter. Nor does wait() make any guarantees about what thread is awakened when notify() is called. So, the current version of the Lock class makes no different guarantees with respect to fairness than synchronized version of doSynchronized(). But we can change that.

The current version of the Lock class calls its own wait() method. If instead each thread calls wait() on a separate object, so that only one thread has called wait() on each object, the Lock class can decide which of these objects to call notify() on, thereby effectively selecting exactly what thread to awaken.

**A Fair Lock**

Below is shown the previous Lock class turned into a fair lock called FairLock. You will notice that the implementation has changed a bit with respect to synchronization and wait() / notify() compared to the Lock class shown earlier.

Exactly how I arrived at this design beginning from the previous Lock class is a longer story involving several incremental design steps, each fixing the problem of the previous step: [**Nested Monitor Lockout**](http://tutorials.jenkov.com/java-concurrency/nested-monitor-lockout.html), [**Slipped Conditions**](http://tutorials.jenkov.com/java-concurrency/slipped-conditions.html), and [**Missed Signals**](http://tutorials.jenkov.com/java-concurrency/thread-signaling.html#missedsignals). That discussion is left out of this text to keep the text short, but each of the steps are discussed in the appropriate texts on the topic ( see the links above). What is important is, that every thread calling lock() is now queued, and only the first thread in the queue is allowed to lock the FairLock instance, if it is unlocked. All other threads are parked waiting until they reach the top of the queue.

public class FairLock {

private boolean isLocked = false;

private Thread lockingThread = null;

private List<QueueObject> waitingThreads =

new ArrayList<QueueObject>();

public void lock() throws InterruptedException{

QueueObject queueObject = new QueueObject();

boolean isLockedForThisThread = true;

synchronized(this){

waitingThreads.add(queueObject);

}

while(isLockedForThisThread){

synchronized(this){

isLockedForThisThread =

isLocked || waitingThreads.get(0) != queueObject;

if(!isLockedForThisThread){

isLocked = true;

waitingThreads.remove(queueObject);

lockingThread = Thread.currentThread();

return;

}

}

try{

queueObject.doWait();

}catch(InterruptedException e){

synchronized(this) { waitingThreads.remove(queueObject); }

throw e;

}

}

}

public synchronized void unlock(){

if(this.lockingThread != Thread.currentThread()){

throw new IllegalMonitorStateException(

"Calling thread has not locked this lock");

}

isLocked = false;

lockingThread = null;

if(waitingThreads.size() > 0){

waitingThreads.get(0).doNotify();

}

}

}

public class QueueObject {

private boolean isNotified = false;

public synchronized void doWait() throws InterruptedException {

while(!isNotified){

this.wait();

}

this.isNotified = false;

}

public synchronized void doNotify() {

this.isNotified = true;

this.notify();

}

public boolean equals(Object o) {

return this == o;

}

}

First you might notice that the lock() method is no longer declared synchronized. Instead only the blocks necessary to synchronize are nested inside synchronized blocks.

FairLock creates a new instance of QueueObject and enqueue it for each thread calling lock(). The thread calling unlock() will take the top QueueObject in the queue and call doNotify() on it, to awaken the thread waiting on that object. This way only one waiting thread is awakened at a time, rather than all waiting threads. This part is what governs the fairness of the FairLock.

Notice how the state of the lock is still tested and set within the same synchronized block to avoid slipped conditions.

Also notice that the QueueObject is really a semaphore. The doWait() and doNotify() methods store the signal internally in the QueueObject. This is done to avoid missed signals caused by a thread being preempted just before calling queueObject.doWait(), by another thread which calls unlock() and thereby queueObject.doNotify(). The queueObject.doWait() call is placed outside the synchronized(this) block to avoid nested monitor lockout, so another thread can actually call unlock() when no thread is executing inside the synchronized(this) block in lock() method.

Finally, notice how the queueObject.doWait() is called inside a try - catch block. In case an InterruptedException is thrown the thread leaves the lock() method, and we need to dequeue it.

**A Note on Performance**

If you compare the Lock and FairLock classes you will notice that there is somewhat more going on inside the lock() and unlock() in the FairLock class. This extra code will cause the FairLock to be a sligtly slower synchronization mechanism than Lock. How much impact this will have on your application depends on how long time the code in the critical section guarded by the FairLock takes to execute. The longer this takes to execute, the less significant the added overhead of the synchronizer is. It does of course also depend on how often this code is called.

**Understanding Deadlock, Livelock and Starvation with Code Examples in Java**

Written by  [Nam Ha Minh](https://www.codejava.net/nam-ha-minh)

Last Updated on 13 August 2019   |  [Print](https://www.codejava.net/java-core/concurrency/understanding-deadlock-livelock-and-starvation-with-code-examples-in-java?tmpl=component&print=1&page=)[Email](https://www.codejava.net/component/mailto/?tmpl=component&template=protostar&link=f3e643f277cf0c79b70ff7f3ea3e3963c3428c36)

This Java concurrency tutorial helps you understand the 3 problems that may happen in multi-threaded applications:  deadlock, livelock and starvation. You will be able to identify each kind of problem so you can know to avoid them.

**1. Understanding Deadlock**

***Deadlock describes a situation where two more threads are blocked because of waiting for each other forever***. When deadlock occurs, the program hangs forever and the only thing you can do is to kill the program.

Let’s consider the account transaction example in [this tutorial](https://www.codejava.net/java-core/concurrency/java-synchronization-tutorial-part-3-using-synchronized-keyword-intrinsic-locking). Modify the maximum amount can be transferred from 10 to 200 in the Bank class as follows:

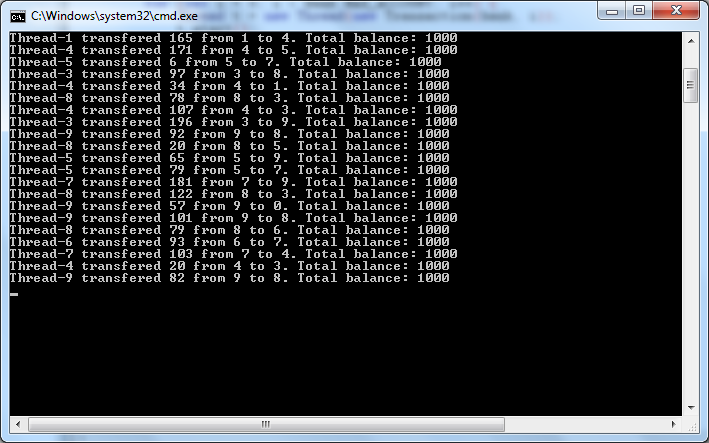
|  |  |
| --- | --- |
| 1 | public static final int MAX\_AMOUNT = 200; |

Look at the Transaction class you see the amount is chosen randomly by this statement:

|  |  |
| --- | --- |
| 1 | int amount = (int) (Math.random() \* Bank.MAX\_AMOUNT); |

Now, recompile the Bank and Transaction classes, and then run the TransactionTest program. Guess what will happen?

You will see that the program runs for a few transactions and hangs forever, as shown in the following screenshot:



The program encounters a deadlock and cannot continue. Why can deadlock happen when we increase the maximum amount of money can be transferred among accounts?

Let’s analyze the code to understand why.

In the Bank class you will each account is initialized with an amount of 100. Now the maximum amount can be transferred is 200, so there will be some threads trying to transfer an amount which is greater than the account’s balance, for example:

      Thread 1 tries to transfer 150 from account 1 to account 2

      Thread 2 tries to transfer 170 from account 3 to account 1

Account 1 has only 100 in balance so thread 1 has to wait for other threads to deposit more funds to this account. Similarly, thread 2 also has to wait because account 3 doesn’t have sufficient fund. Other threads may add funds to accounts 1 and 3, but if all threads are trying to transfer an amount greater than the account’s balance, they are waiting for each other forever. Hence deadlock occurs.

That’s why you see the program quickly runs into deadlock after few transactions have been done. It hangs and you have to press Ctrl + C to terminate the program.

You can ask why the previous version of the example runs fine. It’s because the maximum account is smaller (10) than the balance (100), so all accounts have enough fund to transfer.

**Another Deadlock Example:**

Another common reason for deadlock problem is two or more threads attempt to acquire two locks simultaneously, but in different order. Consider the following class:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | /\*\*   \* Business.java   \* This class is used to illustrate a deadlock situtation.   \* @author www.codejava.net   \*/  public class Business {        private Object lock1 = new Object();      private Object lock2 = new Object();        public void foo() {          synchronized (lock1) {              synchronized (lock2) {                  System.out.println("foo");              }          }      }        public void bar() {          synchronized (lock2) {              synchronized (lock1) {                  System.out.println("bar");              }          }      }  } |

As you can see, both the methods foo() and bar() try to acquire two lock objects lock1 and lock2 but in different order.

And consider the following test program:

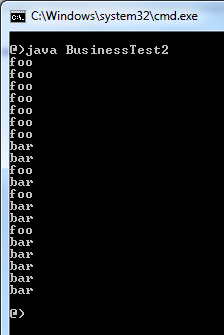
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | /\*\*   \* BusinessTest1.java   \* This program tests for deadlock situtation.   \* @author www.codejava.net   \*/  public class BusinessTest1 {      public static void main(String[] args) {          Business business = new Business();            Thread t1 = new Thread(new Runnable() {              public void run() {                  business.foo();              }          });            t1.start();            Thread t2 = new Thread(new Runnable() {              public void run() {                  business.bar();              }          });            t2.start();      }  } |

This program creates two threads, one executes the foo() method and another executes the bar() method on a shared instance of the Business class. But deadlock is likely never to occur because one thread can execute and exit a method very quickly so the other thread have chance to acquire the locks.

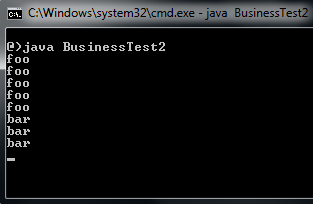
Let’s modify this test program in order to create 10 threads for executing foo() and other 10 threads for executing bar() as follows:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | /\*\*   \* BusinessTest2.java   \* This program tests for deadlock situtation.   \* @author www.codejava.net   \*/  public class BusinessTest2 {      public static void main(String[] args) {          Business business = new Business();            for (int i = 0; i < 10; i++) {              new Thread(new Runnable() {                  public void run() {                      business.foo();                  }              }).start();          }            for (int i = 0; i < 10; i++) {              new Thread(new Runnable() {                  public void run() {                      business.bar();                  }              }).start();          }      }  } |

Run this program several times (4-10 times), you will see that sometimes the program runs fine:



But sometimes it hangs like this:



Why? It’s because deadlock happens. Let me explain how:

- Thread 1 enters foo() method and it acquires lock1. At the same time, thread 2 enters bar() method and it acquires lock2.

- Thread 1 tries to acquire lock2 which is currently held by thread 2, hence thread 1 blocks.

- Thread 2 tries to acquire lock1 which is currently held by thread 1, hence thread 2 blocks.

Both threads block each other forever, deadlock occurs and the program hangs.

**So how to avoid deadlock?**

Java doesn’t have anything to escape deadlock state when it occurs, so you have to design your program to avoid deadlock situation. Avoid acquiring more than one lock at a time. If not, make sure that you acquire multiple locks in consistent order. In the above example, you can avoid deadlock by synchronize two locks in the same order in both methods:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | public void foo() {      synchronized (lock1) {          synchronized (lock2) {              System.out.println("foo");          }      }  }    public void bar() {      synchronized (lock1) {          synchronized (lock2) {              System.out.println("bar");          }      }  } |

Also try to shrink the synchronized blocks as small as possible to avoid unnecessary locking on code that doesn’t need to be synchronized.

**2. Understanding Livelock**

***Livelock describes situation where two threads are busy responding to actions of each other***. They keep repeating a particular code so the program is unable to make further progress:

Thread 1 acts as a response to action of thread 2

Thread 2 acts as a response to action of thread 1

Unlike deadlock, threads are not blocked when livelock occurs. They are simply too busy responding to each other to resume work. In other words, the program runs into an infinite loop and cannot proceed further.

**A Livelock Example:**

Let’s see an example: a criminal kidnaps a hostage and he asks for ransom in order to release the hostage. A police agrees to give the criminal the money he wants once the hostage is released. The criminal releases the hostage only when he gets the money. Both are waiting for each other to act first, hence livelock.

Here’s the code of this example.

Criminal class:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | /\*\*   \* Criminal.java   \* This class is used to demonstrate livelock situation   \* @author www.codejava.net   \*/  public class Criminal {      private boolean hostageReleased = false;        public void releaseHostage(Police police) {          while (!police.isMoneySent()) {                System.out.println("Criminal: waiting police to give ransom");                try {                  Thread.sleep(1000);              } catch (InterruptedException ex) {                  ex.printStackTrace();              }          }            System.out.println("Criminal: released hostage");            this.hostageReleased = true;      }        public boolean isHostageReleased() {          return this.hostageReleased;      }  } |

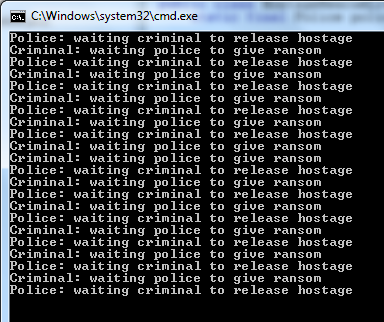
Police class:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | /\*\*   \* Police.java   \* This class is used to demonstrate livelock situation   \* @author www.codejava.net   \*/  public class Police {      private boolean moneySent = false;        public void giveRansom(Criminal criminal) {            while (!criminal.isHostageReleased()) {                System.out.println("Police: waiting criminal to release hostage");                try {                  Thread.sleep(1000);              } catch (InterruptedException ex) {                  ex.printStackTrace();              }          }            System.out.println("Police: sent money");            this.moneySent = true;      }        public boolean isMoneySent() {          return this.moneySent;      }    } |

Test class:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | /\*\*   \* HostageRescueLivelock.java   \* This class is used to demonstrate livelock situation   \* @author www.codejava.net   \*/  public class HostageRescueLivelock {      static final Police police = new Police();        static final Criminal criminal = new Criminal();        public static void main(String[] args) {              Thread t1 = new Thread(new Runnable() {              public void run() {                  police.giveRansom(criminal);              }          });          t1.start();            Thread t2 = new Thread(new Runnable() {              public void run() {                  criminal.releaseHostage(police);              }          });          t2.start();      }    } |

Run this program and you will see that it runs into a loop which never terminates:



So how to avoid livelock? There’s no general guideline, you have to design your program to avoid livelock situation.

**3. Understanding Starvation**

***Starvation describes a situation where a greedy thread holds a resource for a long time so other threads are blocked forever***. The blocked threads are waiting to acquire the resource but they never get a chance. Thus they starve to death.

Starvation can occur due to the following reasons:

- Threads are blocked infinitely because a thread takes long time to execute some synchronized code (e.g. heavy I/O operations or infinite loop).

- A thread doesn’t get CPU’s time for execution because it has low priority as compared to other threads which have higher priority.

- Threads are waiting on a resource forever but they remain waiting forever because other threads are constantly notified instead of the hungry ones.

When a starvation situation occurs, the program is still running but doesn’t run to completion because some threads are not executed.

**A Starvation Example:**

Let’s see an example. Suppose we have a Worker class like this:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | import java.io.\*;    /\*\*   \* Worker.java   \* This class is used to demonstrate starvation situation.   \* @author www.codejava.net   \*/  public class Worker {        public synchronized void work() {          String name = Thread.currentThread().getName();          String fileName = name + ".txt";            try (              BufferedWriter writer = new BufferedWriter(new FileWriter(fileName));          ) {              writer.write("Thread " + name + " wrote this mesasge");          } catch (IOException ex) {              ex.printStackTrace();          }            while (true) {              System.out.println(name + " is working");          }      }  } |

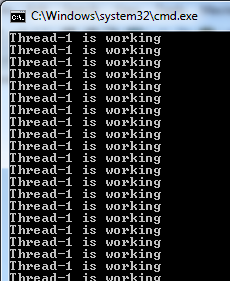
This class has a synchronized method work() that creates a text file <thread-name>.txt and writes a message to it. Then it repeatedly prints a message:

|  |  |
| --- | --- |
| 1 | <thread-name> is working |

And the following program creates 10 threads that call the work() method on a shared instance of the Worker class:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | /\*\*   \* StarvationExample.java   \* This class is used to demonstrate starvation situation.   \* @author www.codejava.net   \*/  public class StarvationExample {        public static void main(String[] args) {          Worker worker = new Worker();            for (int i = 0; i < 10; i++) {              new Thread(new Runnable() {                  public void run() {                      worker.work();                  }              }).start();          }      }  } |

Compile and run this program and you will see that there’s only one thread gets executed:



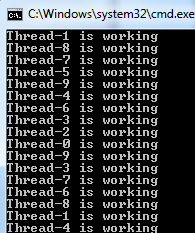
According to the code logic, each thread should create a text file with the name of <thread-name>.txt but you see only one gets created, e.g. thread-1.txt. That means other threads are unable to execute the work() method.

Why does this happen? It’s because the while loop runs forever so that the first executed thread never release the lock, causing other threads to wait forever.

A solution to solve this starvation problem is to make the current thread waits for a specified amount of time so other threads have chance to acquire the lock on the Worker object:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | while (true) {      System.out.println(name + " is working");        try {          wait(1000);      } catch (InterruptedException ex) {          ex.printStackTrace();      }    } |

Recompile and run this program again and you will see that all threads get executed, proven by 10 text files created and in the output:



In general, you should design your program to avoid starvation situation.

**4. Conclusion**

So far I have helped you identify the 3 problems which can happen in multi-threading Java programs: deadlock, livelock and starvation.  Livelock and starvation are less common than deadlock but they still can occur. To summarize, the following points help you understand the key differences of these problems:

- **Deadlock**:  All threads are blocked, the program hangs forever.

- **Livelock**: No threads blocked but they run into infinite loops. The program is still running but unable to make further progress.

- **Starvation**: Only one thread is running, and other threads are waiting forever.

You should be aware of these problems which can occur with multiple threads and synchronization, and design your programs to avoid them.

https://www.codejava.net/java-core/concurrency/understanding-deadlock-livelock-and-starvation-with-code-examples-in-java