<http://tutorials.jenkov.com/java-concurrency/semaphores.html>

# Semaphores

* [Simple Semaphore](http://tutorials.jenkov.com/java-concurrency/semaphores.html#simple)
* [Using Semaphores for Signaling](http://tutorials.jenkov.com/java-concurrency/semaphores.html#as-signal)
* [Counting Semaphore](http://tutorials.jenkov.com/java-concurrency/semaphores.html#counting)
* [Bounded Semaphore](http://tutorials.jenkov.com/java-concurrency/semaphores.html#bounded)
* [Using Semaphores as Locks](http://tutorials.jenkov.com/java-concurrency/semaphores.html#aslock)

|  |
| --- |
|  |

A Semaphore is a thread synchronization construct that can be used either to send signals between threads to avoid [**missed signals**](http://tutorials.jenkov.com/java-concurrency/thread-signaling.html#missedsignals), or to guard a [**critical section**](http://tutorials.jenkov.com/java-concurrency/race-conditions-and-critical-sections.html) like you would with a [**lock**](http://tutorials.jenkov.com/java-concurrency/locks.html). Java 5 comes with semaphore implementations in the java.util.concurrent package so you don't have to implement your own semaphores. Still, it can be useful to know the theory behind their implementation and use.

Java 5 comes with a built-in Semaphore so you don't have to implement your own. You can read more about it in the [**java.util.concurrent.Semaphore**](http://tutorials.jenkov.com/java-util-concurrent/semaphore.html) text, in my java.util.concurrent tutorial.

## Simple Semaphore

Here is a simple Semaphore implementation:

public class Semaphore {

private boolean signal = false;

public synchronized void take() {

this.signal = true;

this.notify();

}

public synchronized void release() throws InterruptedException{

while(!this.signal) wait();

this.signal = false;

}

}

The take() method sends a signal which is stored internally in the Semaphore. The release() method waits for a signal. When received the signal flag is cleared again, and the release() method exited.

Using a semaphore like this you can avoid missed signals. You will call take() instead of notify() and release() instead of wait(). If the call to take() happens before the call to release() the thread calling release() will still know that take() was called, because the signal is stored internally in the signal variable. This is not the case with wait() and notify().

The names take() and release() may seem a bit odd when using a semaphore for signaling. The names origin from the use of semaphores as locks, as explained later in this text. In that case the names make more sense.

## Using Semaphores for Signaling

Here is a simplified example of two threads signaling each other using a Semaphore:

Semaphore semaphore = new Semaphore();

SendingThread sender = new SendingThread(semaphore);

ReceivingThread receiver = new ReceivingThread(semaphore);

receiver.start();

sender.start();

public class SendingThread {

Semaphore semaphore = null;

public SendingThread(Semaphore semaphore){

this.semaphore = semaphore;

}

public void run(){

while(true){

//do something, then signal

this.semaphore.take();

}

}

}

public class RecevingThread {

Semaphore semaphore = null;

public ReceivingThread(Semaphore semaphore){

this.semaphore = semaphore;

}

public void run(){

while(true){

this.semaphore.release();

//receive signal, then do something...

}

}

}

## Counting Semaphore

The Semaphore implementation in the previous section does not count the number of signals sent to it by take() method calls. We can change the Semaphore to do so. This is called a counting semaphore. Here is a simple implementation of a counting semaphore:

public class CountingSemaphore {

private int signals = 0;

public synchronized void take() {

this.signals++;

this.notify();

}

public synchronized void release() throws InterruptedException{

while(this.signals == 0) wait();

this.signals--;

}

}

## Bounded Semaphore

The CoutingSemaphore has no upper bound on how many signals it can store. We can change the semaphore implementation to have an upper bound, like this:

public class BoundedSemaphore {

private int signals = 0;

private int bound = 0;

public BoundedSemaphore(int upperBound){

this.bound = upperBound;

}

public synchronized void take() throws InterruptedException{

while(this.signals == bound) wait();

this.signals++;

this.notify();

}

public synchronized void release() throws InterruptedException{

while(this.signals == 0) wait();

this.signals--;

this.notify();

}

}

Notice how the take() method now blocks if the number of signals is equal to the upper bound. Not until a thread has called release() will the thread calling take() be allowed to deliver its signal, if the BoundedSemaphore has reached its upper signal limit.

## Using Semaphores as Locks

It is possible to use a bounded semaphore as a lock. To do so, set the upper bound to 1, and have the call to take() and release() guard the critical section. Here is an example:

BoundedSemaphore semaphore = new BoundedSemaphore(1);

...

semaphore.take();

try{

//critical section

} finally {

semaphore.release();

}

In contrast to the signaling use case the methods take() and release() are now called by the same thread. Since only one thread is allowed to take the semaphore, all other threads calling take() will be blocked until release() is called. The call to release() will never block since there has always been a call to take() first.

You can also use a bounded semaphore to limit the number of threads allowed into a section of code. For instance, in the example above, what would happen if you set the limit of the BoundedSemaphore to 5? 5 threads would be allowed to enter the critical section at a time. You would have to make sure though, that the thread operations do not conflict for these 5 threads, or you application will fail.

The relase() method is called from inside a finally-block to make sure it is called even if an exception is thrown from the critical section.

<http://tutorials.jenkov.com/java-util-concurrent/semaphore.html>

# Semaphore

* [Semaphore Usage](http://tutorials.jenkov.com/java-util-concurrent/semaphore.html#semaphore-usage)
  + [Guarding Critical Sections](http://tutorials.jenkov.com/java-util-concurrent/semaphore.html#guarding-critical-sections)
  + [Sending Signals Between Threads](http://tutorials.jenkov.com/java-util-concurrent/semaphore.html#sending-signals-between-threads)
* [Fairness](http://tutorials.jenkov.com/java-util-concurrent/semaphore.html#fairness)
* [More Methods](http://tutorials.jenkov.com/java-util-concurrent/semaphore.html#more-methods)

|  |
| --- |
|  |

The java.util.concurrent.Semaphore class is a [**counting semaphore**](http://tutorials.jenkov.com/java-concurrency/semaphores.html#counting). That means that it has two main methods:

* acquire()
* release()

The counting semaphore is initialized with a given number of "permits". For each call to acquire() a permit is taken by the calling thread. For each call to release() a permit is returned to the semaphore. Thus, at most N threads can pass the acquire() method without any release() calls, where N is the number of permits the semaphore was initialized with. The permits are just a simple counter. Nothing fancy here.

## Semaphore Usage

As semaphore typically has two uses:

1. To guard a critical section against entry by more than N threads at a time.
2. To send signals between two threads.

### **Guarding Critical Sections**

If you use a semaphore to guard a critical section, the thread trying to enter the critical section will typically first try to acquire a permit, enter the critical section, and then release the permit again after. Like this:

Semaphore semaphore = new Semaphore(1);

//critical section

semaphore.acquire();

...

semaphore.release();

### **Sending Signals Between Threads**

If you use a semaphore to send signals between threads, then you would typically have one thread call the acquire() method, and the other thread to call the release() method.

If no permits are available, the acquire() call will block until a permit is released by another thread. Similarly, a release() calls is blocked if no more permits can be released into this semaphore.

Thus it is possible to coordinate threads. For instance, if acquire was called after Thread 1 had inserted an object in a shared list, and Thread 2 had called release() just before taking an object from that list, you had essentially created a blocking queue. The number of permits available in the semaphore would correspond to the maximum number of elements the blocking queue could hold.

e.g Compare below file and understand how Semaphore communicate between 2 threads:  
<https://github.com/brenov/unisex-bathroom/blob/master/UnisexBathroomSemaphore/src/unisexbathroomsemaphore/bathroom/Bathroom.java>

In below example, we have acquire() Semaphore in addUser() method, and release() Semaphore in removeUser() method

public class Bathroom {

// Bathroom capacity

private static final int CAPACITY = 5;

// Singleton

private static Bathroom instance = new Bathroom(CAPACITY);

//Using sex

private Sex currentSex;

// Bathroom capacity

private final int capacity;

// Users list

private LinkedHashSet<Person> users;

// Semaphore

private Semaphore semaphore;

private Semaphore leftMutex;

private Semaphore enterMutex;

public Bathroom(int capacity) {

this.capacity = capacity;

this.currentSex = Sex.NONE;

this.users = new LinkedHashSet<>();

this.semaphore = new Semaphore(this.capacity, true);

this.leftMutex = new Semaphore(1, true);

this.enterMutex = new Semaphore(1, true);

}

public static Bathroom getInstance() {

return instance;

}

public void addUser(Person person) {

try {

this.semaphore.acquire();

} catch (InterruptedException ex) {

Logger.getLogger(Bathroom.class.getName())

.log(Level.SEVERE, null, ex);

}

//if it is the first person to enter the bathroom

if (this.isEmpty()) {

this.currentSex = person.getSex();

}

// Check if the bathroom isn't full

if (!this.isFull() && !this.users.contains(person)

&& getCurrentSex().equals(person.getSex())) {

try {

this.enterMutex.acquire();

} catch (InterruptedException ex) {

Logger.getLogger(Bathroom.class.getName())

.log(Level.SEVERE, null, ex);

}

// Add the person

if (this.users.add(person)) {

System.out.println(person.getName() + " entered the bathroom");

}

this.enterMutex.release();

// Check if the bathroom is full

if (this.isFull()) {

System.out.println("The bathroom is full");

}

}

}

public void removeUser(Person person) {

this.semaphore.release();

// Check if the bathroom in't empty

if (!this.isEmpty()) {

try {

this.leftMutex.acquire();

} catch (InterruptedException ex) {

Logger.getLogger(Bathroom.class.getName())

.log(Level.SEVERE, null, ex);

}

if (this.users.remove(person)) {

System.out.println(person.getName() + " left the bathroom");

}

this.leftMutex.release();

// Check if the bathroom is empty

if (this.isEmpty()) {

System.out.println("The bathroon is empty");

this.currentSex = Sex.NONE;

}

}

}

public boolean isInTheBathroom(Person person) {

return this.users.contains(person);

}

public boolean isFull() {

return this.capacity == this.users.size();

}

public boolean isEmpty() {

return this.users.isEmpty();

}

public Sex getCurrentSex() {

return this.currentSex;

}

@Override

public String toString() {

return "Bathroom{" + "currentSex = " + this.currentSex

+ ", capacity = " + this.capacity

+ ", numberOfUsers = " + this.users.size() + '}';

}

}

## Fairness

No guarantees are made about [**fairness**](http://tutorials.jenkov.com/java-concurrency/starvation-and-fairness.html) of the threads acquiring permits from the Semaphore. That is, there is no guarantee that the first thread to call acquire() is also the first thread to obtain a permit. If the first thread is blocked waiting for a permit, then a second thread checking for a permit just as a permit is released, may actually obtain the permit ahead of thread 1.

If you want to enforce fairness, the Semaphore class has a constructor that takes a boolean telling if the semaphore should enforce fairness. Enforcing fairness comes at a performance / concurrency penalty, so don't enable it unless you need it.

Here is how to create a Semaphore in fair mode:

Semaphore semaphore = new Semaphore(1, true);

## More Methods

The java.util.concurrent.Semaphore class has lots more methods. For instance:

* availablePermits()
* acquireUninterruptibly()
* drainPermits()
* hasQueuedThreads()
* getQueuedThreads()
* tryAcquire()
* etc.

Check out the JavaDoc for more details on these methods.

<https://techvidvan.com/tutorials/semaphore-in-java/>

# Semaphore in Java – Working, Types and Implementation

In this article, we will discuss Semaphores in Java. We use Semaphores in the Thread Synchronization in Java. We will study the working of semaphores and types of Semaphores with examples.

There is a Semaphore class that has many constructors and methods that control access over shared resources or concurrent threads that want to access the specified resource.

Let’s begin studying the Semaphore in Java.

### **Semaphore in Java**

A Semaphore in Java is a Thread Synchronization construct that controls access to the shared resource with the help of counters.

A semaphore also sends the signals between the threads so that the missed signals are not avoided. A semaphore is a kind of variable that manages the concurrent processes and synchronizes them.

We can use a semaphore to avoid race conditions. A semaphore can restrict the number of threads to access a shared resource.

For example, a semaphore can limit a maximum of 10 connections to access a shared file simultaneously.

### **Working of a Semaphore**

We know that a semaphore controls access over the shared resource with the help of counters. The counter has some value.

3.8M

303

machine learning project spam detector

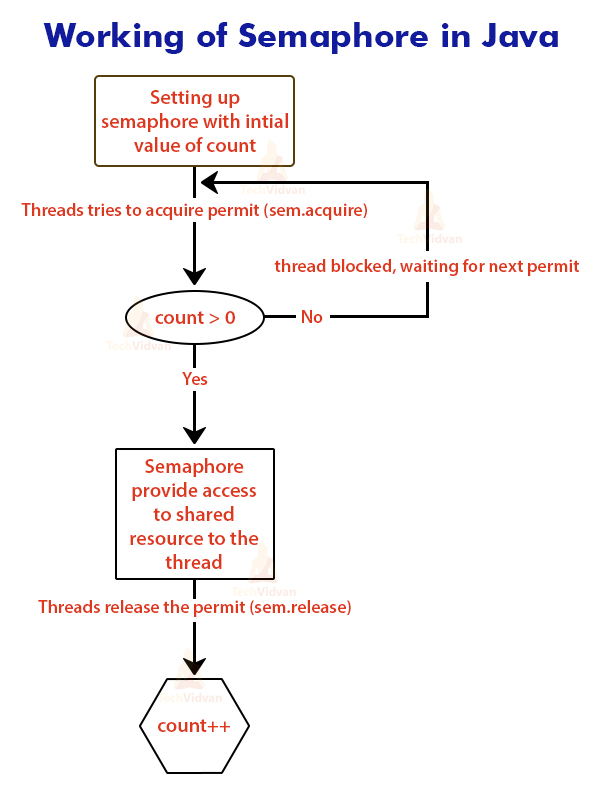
If the value of the counter is greater than zero, then the thread gets permission to access the resource and then the counter value decrements.

And if the value of the counter is zero, then the thread does not get access to the resource.

When the thread completes its execution inside the resources or no longer needs the resource then it releases the permit and the value of the counter gets incremented.

**Flow chart of Semaphore:**

The following figure shows the working of the semaphore using a flow chart:

[](https://i0.wp.com/techvidvan.com/tutorials/wp-content/uploads/sites/2/2020/06/Working-of-Semaphore-in-Java.jpg?ssl=1)

### **The Semaphore class in Java**

Java provides the Semaphore class to implement the above mechanism of semaphore. This Semaphore class is present in the java.util.concurrent package.

The Semaphore class provides all the functionalities on its own, so there is no need to implement it manually.

**Constructors of Semaphore class:**

There are two constructors of the Semaphore class. They are:

**1.** Semaphore(int num)  
**2.** Semaphore(int num, boolean how)

Here, num represents the number of threads that can access the shared resource. The boolean value is set to true so that it permits waiting threads in the order they requested the access.

**Methods of Semaphore class:**

There are following methods in Semaphore class:

**1. acquire():** This method returns true if a permit is available immediately and acquires it, otherwise, returns false, but acquire() acquires a permit and blocking until one is available.

**2. release():** The release() method releases a permit.

**3. availablePermits():** This method returns the number of current permits available.

package com.techvidvan.semaphore;

import java.util.concurrent. \* ;

//A shared resource/class.

class SharedResource {

static int count = 0;

}

class MyThread extends Thread {

Semaphore semaphore;

String threadName;

public MyThread(Semaphore semaphore, String threadName) {

super(threadName);

this.semaphore = semaphore;

this.threadName = threadName;

}@Override

public void run() {

if (this.getName().equals("Thread1")) {

System.out.println("Starting " + threadName);

try {

// First, get a permit.

System.out.println(threadName + " is waiting for a permit.");

//Acquiring the lock

semaphore.acquire();

System.out.println(threadName + " gets a permit.");

// Accessing the shared resource.

// other waiting threads will wait, until this thread releases the lock

for (int i = 0; i < 5; i++) {

SharedResource.count++;

System.out.println(threadName + ": " + SharedResource.count);

//Allowing a context switch if possible.for thread B to execute

Thread.sleep(10);

}

}

catch(InterruptedException exc) {

System.out.println(exc);

}

// Release the permit.

System.out.println(threadName + " releases the permit.");

semaphore.release();

}

// Run by thread B

else {

System.out.println("Starting " + threadName);

try {

// First, get a permit.

System.out.println(threadName + " is waiting for a permit.");

// acquiring the lock

semaphore.acquire();

System.out.println(threadName + " gets a permit.");

// Now, accessing the shared resource.

// other waiting threads will wait, until this thread release the lock

for (int i = 0; i < 5; i++) {

SharedResource.count--;

System.out.println(threadName + ": " + SharedResource.count);

Thread.sleep(10);

}

}

catch(InterruptedException exc) {

System.out.println(exc);

}

// Release the permit.

System.out.println(threadName + " releases the permit.");

semaphore.release();

}

}

}

//Main class

public class SemaphoreDemo {

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

// Creating a Semaphore object with number of permits = 1

Semaphore semaphore = new Semaphore(1);

// Creating two threads with name t1 and t2

// Note that thread A will increment the count and thread B will decrement the count

MyThread t1 = new MyThread(semaphore, "Thread1");

MyThread t2 = new MyThread(semaphore, "Thread2");

t1.start();

t2.start();

// waiting for threads t1 and t2

t1.join();

t2.join();

//count will always be 0 after both threads complete their execution

System.out.println("count: " + SharedResource.count);

}

}

**Output:**

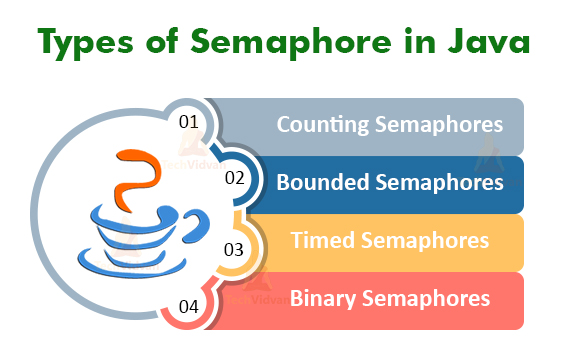
Starting Thread1  
Starting Thread2  
Thread1 is waiting for a permit.  
Thread2 is waiting for a permit.  
Thread1 gets a permit.  
Thread1: 1  
Thread1: 2  
Thread1: 3  
Thread1: 4  
Thread1: 5  
Thread1 releases the permit.  
Thread2 gets a permit.  
Thread2: 4  
Thread2: 3  
Thread2: 2  
Thread2: 1  
Thread2: 0  
Thread2 releases the permit.  
count: 0

***Note:****The output can be different in different executions of the above program, but the final value of the count variable will always remain 0.*

**Explanation of the above program :**

* The program uses a semaphore to control access to the count variable, which is a static variable within the Shared class. Shared.count increments five times by thread t1 and decrements five times by thread t2. To prevent these two threads from accessing Shared.count at the same time, access is allowed only after getting a permit from the controlling semaphore. After access is complete, the permit releases. Using this way, only one thread at a time will access Shared.count, as we can see in the output.
* Notice the call to sleep( ) within run( ) method inside MyThread class. We use it to “prove” that the semaphore synchronizes the accesses to Shared.count. In run() method, the call to sleep() method invokes the thread to pause between each access to Shared.count. This would enable the second thread to run. However, because of the semaphore, the second thread must wait until the first has released the permit, which happens only after all accesses by the first thread are complete. Thus, Shared.count is first incremented five times by thread t1 and then decremented five times by thread t2. The increments and decrements are not intermixed at assembly code.
* Without the use of the semaphore, accesses to Shared.count by both threads would occur simultaneously, and there could be a mixture of increments and decrements.To confirm this, try commenting out the calls to acquire( ) and release( ). When you run the program, you will see that access to Shared.count is no longer synchronized, thus you will not always get count value 0.

### **Types of Semaphore**

[](https://i0.wp.com/techvidvan.com/tutorials/wp-content/uploads/sites/2/2020/06/Types-of-Semaphore-in-Java.jpg?ssl=1)

There are 4 types of Semaphores:

#### **1. Counting Semaphores**

There are some cases when more than one process wants to execute in a critical section simultaneously, so the counting semaphores are useful to achieve this.

The Semaphore implementation takes place through the take() method.

**Counting Semaphore in Java Example:**

public **class** CountingSemaphore

{

private int signals = 0;

public synchronized **void** take()

{

**this**.signals++;

**this**.notify();

}

public synchronized **void** release() throws InterruptedException

{

**while**(**this**.signals == 0)

wait();

**This**.signals--;

}

}

#### **2. Bounded Semaphores**

The counting semaphores do not have any upper bound values that how many signals it can store. So the use of bounded semaphores is to set the upper bound.

public **class** BoundedSemaphore

{

private int signal = 0;

private int bound = 0;

public BoundedSemaphore(int upperBound)

{

**this**.bound = upperBound;

}

public **void** synchronized take() throws InterruptedException

{

**while**(**this**.signal == bound)

wait();

**this**.signal++;

**this**.notify++;

}

public **void** synchronized release() throws InterruptedException

{

**while**(**this**.signal == 0)

wait();

**this**.signal--;

}

}

#### **3. Timed Semaphores**

The Timed Semaphore is a semaphore that allows a thread to run for a particular duration or a period of time. After this period of time, the timer resets, and all the permits release.

#### **4. Binary Semaphores**

The Binary semaphores are quite similar to the counting semaphores, but they can take only binary values i.e., either 0 or 1.

It is easier to implement the binary semaphore than the counting semaphore. If the value = 1, then the signal operation succeeds, otherwise it fails, if the value is 0.

### **Using Semaphores as Locks**

We can also use a bounded semaphore as a lock. For this, we have to set the upper bound to 1 and need to call the **take()** and **release()** methods to guard the critical section.

Here is an example:

BoundedSemaphore semaphore = **new** BoundedSemaphore(1);

...

semaphore.take();

**try** {

//critical section

}

finally {

semaphore.release();

}

### **Conclusion**

Here we come to the end of the article on Java Semaphores. We explored the basics of Semaphores with their working and types with the examples.

We hope that you would now have understood the concept of semaphores with implementation.

<https://stackoverflow.com/questions/17683575/binary-semaphore-vs-a-reentrantlock>

# [Binary Semaphore vs a ReentrantLock](https://stackoverflow.com/questions/17683575/binary-semaphore-vs-a-reentrantlock)

Q: I've been trying to understand Reentrant locks and Semaphores ( the nesting of Reentrant locks vs release/unlock mechanism ).

It seems that having a Semaphore requires you to write a more thoroughly tested application because the release() method does not check if the thread releasing the permit is actually holding it. When I tested my test code, I found out that this may subsequently increase the number of permits beyond the initial limit. On the other hand, if a thread is not holding a reentrant lock when it invokes the unlock method, we get an IllegalMonitorException.

So would it be right to say that there is no real reason ever to have a binary semaphore as everything that a binary semaphore can do can also be done by a ReentrantLock. If we use binary semaphores we would have to check the entire method call stack to see if a permit was acquired before ( also was it released too if there is a possibility of a subsequent acquire - which might block if a release does not proceed it and so on ). Also since reentrant locks also provide one lock per object, isn't it always a better idea to prefer a reentrant lock to a binary semaphore?

I have checked a post here that talks about difference between a binary semaphore and a mutex but is there a thing like a mutex in Java?

Thanks, Chan.

P.S - I had posted this question in another forum ( <http://www.coderanch.com/t/615796/threads/java/reason-prefer-binary-Semaphore-Reentrant> ) and I haven't received a response yet. I thought I'd post it here as well to see what I can get.

A:

there is no real reason ever to have a binary semaphore as everything that a binary semaphore can do can also be done by a ReentrantLock

If all you need is reentrant mutual exclusion, then yes, there is no reason to use a binary semaphore over a ReentrantLock. If for any reason you need non-ownership-release semantics then obviously semaphore is your only choice.

Also since reentrant locks also provide one lock per object, isn't it always a better idea to prefer a reentrant lock to a binary semaphore?

It depends on the need. Like previously explained, if you need a simple mutex, then don't choose a semaphore. If more than one thread (but a limited number) can enter a critical section you can do this through either thread-confinement or a semaphore.

I have checked a post here that talks about difference between a binary semaphore and a mutex but is there a thing like a mutex in Java?

ReentrantLock and synchronized are examples of mutexes in Java.