**Binary Semaphore Tutorial and Example**

A semaphore is a counter that protects the access to one or more shared resources. In this tutorial, we will learn **how to use the binary semaphore** for controlling access to a shared resource by multiple threads.

**How Semaphores Work?**

You can visualize a semaphore as **counter which can be incremented or decremented**. You initialize the semaphore with a number i.e. 5. Now this semaphore can be decremented maximum five times in a row until counter reaches to 0. Once counter is zero, you can increment it to maximum five times to make it 5. The counter value of semaphore MUST always be inside limit **0 >= n >= 5** (in our case).

Obviously, semaphores are more than just being counters. They are able to make threads wait when counter value is zero i.e. they act as Locks with counter functionality.

Talking in terms of multi-threading, when a thread wants to access one of shared resources (guarded by semaphore), first, it must acquire the semaphore. If the internal counter of the semaphore is greater than 0, the semaphore decrements the counter and allows access to the shared resource. Otherwise, if the counter of the semaphore is 0, the semaphore puts the thread to sleep until the counter is greater than 0. A value of 0 in the counter means all the shared resources are used by other threads, so the thread that wants to use one of them must wait until one is free.

When a thread has finished the use of the shared resource, it must release the semaphore so that the other threads can access the shared resource. That operation increases the internal counter of the semaphore.

**When to use Binary Semaphore?**

Quite obvious,**binary semaphore can have a value either 0 or 1**. It means **binary semaphore protect the access to a SINGLE shared resource**, so the internal counter of the semaphore can only take the values 1 or 0.

So whenever you have a requirement for protecting the access to a SINGLE resource accessed by multiple threads, you can use Binary Semaphore.

**How to use Binary Semaphore?**

To show the usage of binary semaphore, we are going to implement a print queue that can be used by concurrent tasks to print their jobs. This print queue will be protected by a binary semaphore, so only one thread can print at a time.  
 **PrintingJob.java**

This class represents an independent printing which could be submitted to printer. This class implements Runnable interface, so that printer can execute it when it’s turn come.

|  |
| --- |
| class PrintingJob implements Runnable  {     private PrinterQueue printerQueue;       public PrintingJob(PrinterQueue printerQueue)     {        this.printerQueue = printerQueue;     }       @Override     public void run()     {        System.out.printf("%s: Going to print a document\n", Thread.currentThread().getName());        printerQueue.printJob(new Object());     }  } |

**PrinterQueue.java**

This class represent the printer queue/ printer. **Please note that we pass the value 1 as the parameter of this Semaphore’s constructor, so you are creating a binary semaphore.**

|  |
| --- |
| class PrinterQueue  {     private final Semaphore semaphore;       public PrinterQueue()     {        semaphore = new Semaphore(1);     }       public void printJob(Object document)     {        try        {           semaphore.acquire();             Long duration = (long) (Math.random() \* 10000);           System.out.println(Thread.currentThread().getName() + ": PrintQueue: Printing a Job during " + (duration / 1000) + " seconds :: Time - " + new Date());           Thread.sleep(duration);        }        catch (InterruptedException e)        {           e.printStackTrace();        }        finally        {           System.out.printf("%s: The document has been printed\n", Thread.currentThread().getName());             semaphore.release();        }     }  } |

Let’s test our printer program:

|  |
| --- |
| public class SemaphoreExample  {     public static void main(String[] args)     {        PrinterQueue printerQueue = new PrinterQueue();        Thread thread[] = new Thread[10];        for (int i = 0; i < 10; i++)        {           thread[i] = new Thread(new PrintingJob(printerQueue), "Thread " + i);        }        for (int i = 0; i < 10; i++)        {           thread[i].start();        }     }  }    Output:    Thread 0: Going to print a document  Thread 9: Going to print a document  Thread 8: Going to print a document  Thread 5: Going to print a document  Thread 7: Going to print a document  Thread 6: Going to print a document  Thread 3: Going to print a document  Thread 4: Going to print a document  Thread 2: Going to print a document  Thread 1: Going to print a document  Thread 0: PrintQueue: Printing a Job during 3 seconds :: Time - Tue Jan 06 18:00:12 IST 2015  Thread 0: The document has been printed  Thread 9: PrintQueue: Printing a Job during 0 seconds :: Time - Tue Jan 06 18:00:16 IST 2015  Thread 9: The document has been printed  Thread 8: PrintQueue: Printing a Job during 7 seconds :: Time - Tue Jan 06 18:00:16 IST 2015  Thread 8: The document has been printed  Thread 5: PrintQueue: Printing a Job during 0 seconds :: Time - Tue Jan 06 18:00:24 IST 2015  Thread 5: The document has been printed  Thread 7: PrintQueue: Printing a Job during 4 seconds :: Time - Tue Jan 06 18:00:24 IST 2015  Thread 7: The document has been printed  Thread 6: PrintQueue: Printing a Job during 3 seconds :: Time - Tue Jan 06 18:00:29 IST 2015  Thread 6: The document has been printed  Thread 3: PrintQueue: Printing a Job during 8 seconds :: Time - Tue Jan 06 18:00:33 IST 2015  Thread 3: The document has been printed  Thread 4: PrintQueue: Printing a Job during 0 seconds :: Time - Tue Jan 06 18:00:41 IST 2015  Thread 4: The document has been printed  Thread 2: PrintQueue: Printing a Job during 4 seconds :: Time - Tue Jan 06 18:00:42 IST 2015  Thread 2: The document has been printed  Thread 1: PrintQueue: Printing a Job during 3 seconds :: Time - Tue Jan 06 18:00:46 IST 2015  Thread 1: The document has been printed |

Look at the printJob() method. This method shows the three steps you must follow when you use a semaphore to implement a critical section, and protect the access to a shared resource:

1. First, you acquire the semaphore, with the acquire() method.
2. Then, you do the necessary operations with the shared resource.