

Basic Thread Synchronization

**Programación Concurrente
2017**

Ing. Ventre, Luis O.

Synchronizing data access with read/write locks

- Una gran ventaja de los LOCKS es la interface READWRITELOCK y la clase asociada.
- Esta clase ReentrantReadWriteLock tiene asociado dos LOCKS
 - Uno para Read
 - Uno para Write
 - Implementados de manera que:
 - Puede haber más de un lector.
 - Solo puede haber 1 escritor.
 - Cuando hay 1 escritor no pueden haber lectores ejecutando.

Synchronizing data access with read/write locks

- A continuación se verá un ejemplo:
- Clases:
 - PricesInfo: Almacenará la información de precio de dos productos. Sus métodos:
 - Constructor
 - getPrice1 (hará uso de un lock tipo read)
 - getPrice2 (hará uso de un lock tipo read)
 - setPrices (hará uso de un lock tipo write)

Synchronizing data access with read/write locks

- PricesInfo

```
Main.java *PricesInfo.java
1 package com.packtpub.java7.concurrency.chapter2.recipe4.task;
2 import java.util.concurrent.locks.ReadWriteLock;
3 * This class simulates the store of two prices. We will
10 public class PricesInfo {
11
12     private double price1;
13     private double price2;
14
15     private ReadWriteLock lock;
16
17
18     * Constructor of the class. Initializes the prices and the Lock
20     public PricesInfo(){
21         price1=1.0;
22         price2=2.0;
23         lock=new ReentrantReadWriteLock();
24     }
25
26
27     * Returns the first price
30     public double getPrice1() {
31         lock.readLock().lock();
32         double value=price1;
33         lock.readLock().unlock();
34         return value;
35     }
36
37
38     * Returns the second price
41     public double getPrice2() {
42         lock.readLock().lock();
43         double value=price2;
44         lock.readLock().unlock();
45         return value;
46     }
47
48
49     * Establish the prices
53     public void setPrices(double price1, double price2) {
54         lock.writeLock().lock();
55         this.price1=price1;
56         this.price2=price2;
57         lock.writeLock().unlock();
58     }
59 }
```

Synchronizing data access with read/write locks

- Clase Reader (runnable). Implementa un lector. Sus métodos:
 - Constructor con atributo clase PricesInfo
 - Run. Bucle con 10 accesos a precio1 y precio2.

```
1 package com.packtpub.java7.concurrency.chapter2.recipe4.task;
2
3
4 * This class implements a reader that consults the prices
5
6 public class Reader implements Runnable {
7
8
9
10 * Class that stores the prices
11 private PricesInfo pricesInfo;
12
13
14
15 * Constructor of the class
16 public Reader (PricesInfo pricesInfo){
17     this.pricesInfo=pricesInfo;
18 }
19
20
21
22
23 * Core method of the reader. Consults the two prices and prints them
24 @Override
25 public void run() {
26     for (int i=0; i<10; i++){
27         System.out.printf("%s: Price 1: %f\n",Thread.currentThread().getName(),pricesInfo.getPrice1());
28         System.out.printf("%s: Price 2: %f\n",Thread.currentThread().getName(),pricesInfo.getPrice2());
29     }
30 }
31
32 }
33
34 }
```

Synchronizing data access with read/write locks

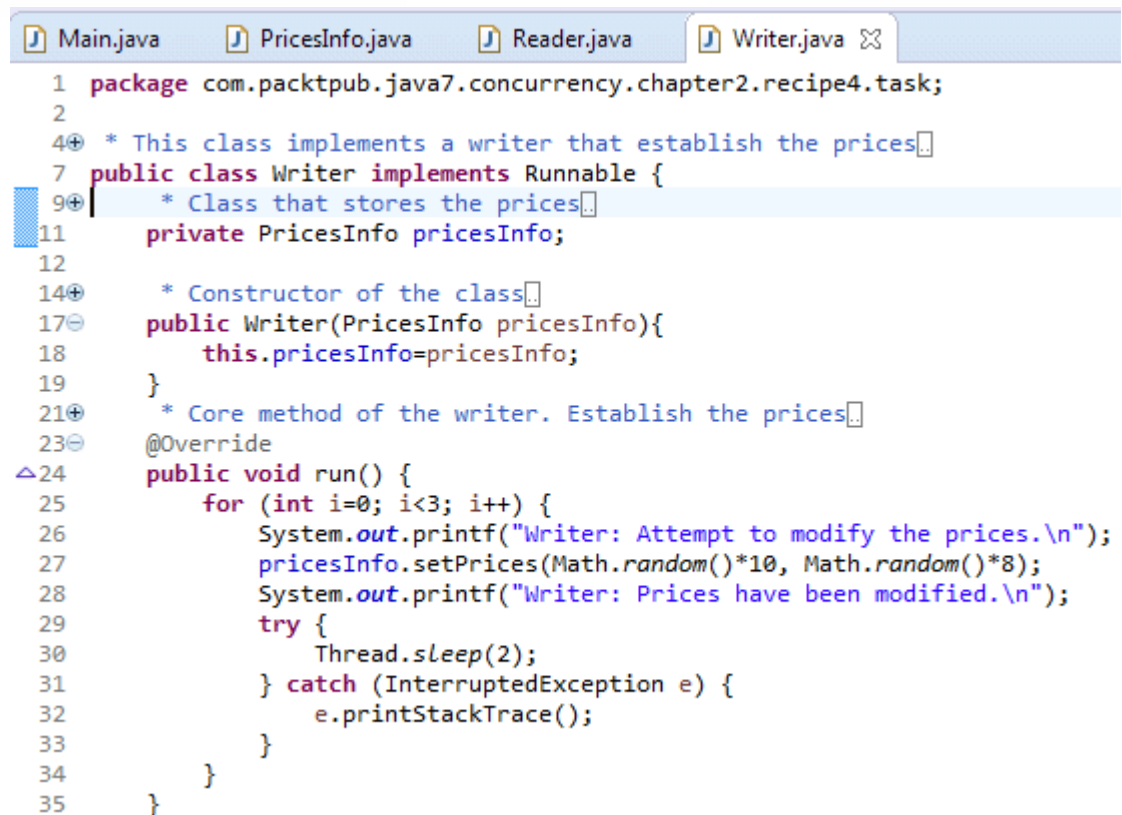
– Clase Writer (runnable). Implementa un escritor. Sus métodos:

- Constructor con atributo clase PricesInfo
- Run.

Impresión de intento de modificación .

Acceso a setPrices

Sleep.



```
1 package com.packtpub.java7.concurrency.chapter2.recipe4.task;
2
3
4 * This class implements a writer that establish the prices.
5
6 public class Writer implements Runnable {
7     * Class that stores the prices.
8
9     private PricesInfo pricesInfo;
10
11     * Constructor of the class.
12
13     public Writer(PricesInfo pricesInfo){
14         this.pricesInfo=pricesInfo;
15     }
16
17     * Core method of the writer. Establish the prices.
18
19     @Override
20     public void run() {
21         for (int i=0; i<3; i++) {
22             System.out.printf("Writer: Attempt to modify the prices.\n");
23             pricesInfo.setPrices(Math.random()*10, Math.random()*8);
24             System.out.printf("Writer: Prices have been modified.\n");
25             try {
26                 Thread.sleep(2);
27             } catch (InterruptedException e) {
28                 e.printStackTrace();
29             }
30         }
31     }
32 }
```

Synchronizing data access with read/write locks

– Clase Main.

- Crea objeto PricesInfo.
- Crea 5 Threads y 5 objetos Reader
- Crea 1 Thread y 1 objeto escritor.
- Lanza ejecución de todos los hilos.

```

Main.java PricesInfo.java Reader.java Writer.java
1 package com.packtpub.java7.concurrency.chapter2.recipe4.core;
2 import com.packtpub.java7.concurrency.chapter2.recipe4.task.PricesInfo;
3
4
5
6 * Main class of the example
7 public class Main {
8     * Main class of the example
9     public static void main(String[] args) {
10
11         // Creates an object to store the prices
12         PricesInfo pricesInfo=new PricesInfo();
13
14         Reader readers[]=new Reader[5];
15         Thread threadsReader[]=new Thread[5];
16
17         // Creates five readers and threads to run them
18         for (int i=0; i<5; i++){
19             readers[i]=new Reader(pricesInfo);
20             threadsReader[i]=new Thread(readers[i]);
21         }
22
23         // Creates a writer and a thread to run it
24         Writer writer=new Writer(pricesInfo);
25         Thread threadWriter=new Thread(writer);
26
27         // Starts the threads
28         for (int i=0; i<5; i++){
29             threadsReader[i].start();
30         }
31         threadWriter.start();
32     }
33 }

```

Synchronizing data access with read/write locks

- Ejecución:

```
Thread-2: Price 2: 1.346672
Thread-1: Price 2: 1.346672
Thread-4: Price 2: 1.346672
Writer: Attempt to modify the prices.
Writer: Prices have been modified.
Thread-1: Price 1: 9.464604
Thread-4: Price 1: 9.464604
Thread-1: Price 2: 5.623313
Thread-4: Price 2: 5.623313
Thread-1: Price 1: 4.348840
Thread-4: Price 1: 4.348840
Thread-1: Price 2: 5.623313
Thread-4: Price 2: 5.623313
Writer: Attempt to modify the prices
```

- Es responsabilidad del programador asegurar el correcto uso de los locks. Error modificar precio por un lector.

Synchronizing data access with read/write locks

- Cuando será óptimo la utilización de ReadWriteLock?.
 - A) Un sistema de base de datos que muy esporádicamente se modifican los valores y se accede y se consulta continuamente por muchos lectores?
 - B) Un sistema de base de datos en la cual los precios cambian continuamente y hay muy poco acceso de consultas a la misma.
- Ojo, la implementación del ReadWriteLock si no es el escenario correcto, puede ser más caro que una sección crítica clásica.

Modifying Lock fairness

- Los constructores de las clases:

- ReentrantLock
- ReentrantReadWriteLock

Admiten un parámetro boolean llamado FAIR que permite controlar el comportamiento de ambas.

- El valor FALSO es por defecto y es llamado NON-FAIR.
- En éste modo cuando hay varios hilos esperando por el lock, se libera el lock y debe elegir uno, no hay criterio de selección.

Modifying Lock fairness

- El valor TRUE es llamado FAIR MODE; en la situación anterior, la elección tomada es el hilo que estába esperando hace más tiempo.
- Tener en cuenta que esto NO funciona con el método tryLock(), ya que el mismo no duerme ni bloquea el hilo.
- Se modificará un ej. antes visto para observar el resultado de un modo y otro.

Modifying Lock fairness

- Clases:
 - **PrintQueue:** Se modifica el constructor con un lock con parámetro TRUE.
 - **PrintJob:** Simular la impresión pero ahora en dos etapas, liberando el lock y retomándolo.
 - **Main:** Lanzar los hilos con 100 mseg. de diferencia de creación.

Modifying Lock fairness

– PrintQueue

```
Main.java  Job.java  PrintQueue.java ✕
1 package com.packtpub.java7.concurrency.chapter2.recipe5.task;
2
3 import java.util.concurrent.locks.Lock;
4
5
6 * This class simulates a print queue.
7
10 public class PrintQueue {
11
12     * Creates a lock to control the access to the queue.
13     private final Lock queueLock=new ReentrantLock(true);
14
15
16     * Method that prints the Job. The printing is divided in two phase two.
17     public void printJob(Object document){
18
19         queueLock.lock();
20         try {
21             Long duration=(long)(Math.random()*10000);
22             System.out.printf("%s: PrintQueue: Printing a Job during %d seconds\n",Thread.currentThread().getName(),(duration/1000));
23             Thread.sleep(duration);
24         } catch (InterruptedException e) {
25             e.printStackTrace();
26         } finally {
27             queueLock.unlock();
28         }
29
30         queueLock.lock();
31         try {
32             Long duration=(long)(Math.random()*10000);
33             System.out.printf("%s: PrintQueue: Printing a Job during %d seconds\n",Thread.currentThread().getName(),(duration/1000));
34             Thread.sleep(duration);
35         } catch (InterruptedException e) {
36             e.printStackTrace();
37         } finally {
38             queueLock.unlock();
39         }
40     }
41 }
```

Modifying Lock fairness

– Job

```
Main.java  Job.java  PrintQueue.java
1  package com.packtpub.java7.concurrency.chapter2.recipe5.task;
2
3  4+ * This class simulates a job that send a document to print..
4  7  public class Job implements Runnable {
5  8
6  10+ * The queue to send the documents..
7  12  private PrintQueue printQueue;
8  13
9  15+ * Constructor of the class. Initializes the print queue..
10 18- public Job(PrintQueue printQueue){
11 19     this.printQueue=printQueue;
12 20 }
13 21
14 23+ * Core method of the Job. Sends the document to the queue..
15 25- @Override
16 26 public void run() {
17 27     System.out.printf("%s: Going to print a job\n",Thread.currentThread().getName());
18 28     printQueue.printJob(new Object());
19 29     System.out.printf("%s: The document has been printed\n",Thread.currentThread().getName());
20 30 }
21 31 |
22 32 }
```

Modifying Lock fairness

– Main

```

Main.java  Job.java  PrintQueue.java
1 package com.packtpub.java7.concurrency.chapter2.recipe5.core;
2 import com.packtpub.java7.concurrency.chapter2.recipe5.task.Job;
5 * Main class of the example
8 public class Main {
10     * Main method of the example
13     public static void main (String args[]){
14         // Creates the print queue
15         PrintQueue printQueue=new PrintQueue();
16
17         // Creates ten jobs and the Threads to run them
18         Thread thread[]=new Thread[10];
19         for (int i=0; i<10; i++){
20             thread[i]=new Thread(new Job(printQueue),"Thread "+i);
21         }
22         // Launch a thread ever 0.1 seconds
23         for (int i=0; i<10; i++){
24             thread[i].start();
25             try {
26                 Thread.sleep(100);
27             } catch (InterruptedException e) {
28                 e.printStackTrace();
29             }
30         }
31     }
32
33 }
```

Modifying Lock fairness

- Ejecución del proyecto FAIR MODE
 - Los threads son creados con 100 ms de diferencia. El primer thread accede a la impresion con el LOCK.
 - Luego los restantes threads se bloquean.
 - Cuando el primer Thread, termina su primera parte del trabajo, libera el LOCK y lo intenta adquirir nuevamente de inmediato.
 - En ese momento hay 10 Threads esperando el LOCK.
 - Como se pasó parametro Fair TRUE, la JVM elige el hilo que hace más tiempo espera, en concreto el segundo creado.

Modifying Lock fairness

- Ejecución del proyecto. FAIR MODE

```
Console X
<terminated> Main (41) [Java Application] C:\Program Files\Java\jre7\bin\
Thread 0: Going to print a job
Thread 0: PrintQueue: Printing a Job during 7 seconds
Thread 1: Going to print a job
Thread 2: Going to print a job
Thread 3: Going to print a job
Thread 4: Going to print a job
Thread 5: Going to print a job
Thread 6: Going to print a job
Thread 7: Going to print a job
Thread 8: Going to print a job
Thread 9: Going to print a job
Thread 1: PrintQueue: Printing a Job during 9 seconds
Thread 2: PrintQueue: Printing a Job during 0 seconds
Thread 3: PrintQueue: Printing a Job during 9 seconds
Thread 4: PrintQueue: Printing a Job during 5 seconds
Thread 5: PrintQueue: Printing a Job during 0 seconds
Thread 6: PrintQueue: Printing a Job during 6 seconds
Thread 7: PrintQueue: Printing a Job during 7 seconds
Thread 8: PrintQueue: Printing a Job during 9 seconds
Thread 9: PrintQueue: Printing a Job during 1 seconds
Thread 0: PrintQueue: Printing a Job during 8 seconds
Thread 0: The document has been printed
Thread 1: PrintQueue: Printing a Job during 7 seconds
```

Modifying Lock fairness

- Ejecución del proyecto NON FAIR MODE
 - Los threads son creados con 100 ms de diferencia. El primer thread accede a la impresion con el LOCK.
 - Luego los restantes threads se bloquean.
 - Cuando el primer Thread, termina su primera parte del trabajo libera el LOCK y lo intenta adquirir nuevamente de inmediato.
 - En ese momento hay 10 Threads esperando el LOCK.
 - Como se paso parametro Fair FALSE, la ejecucion varía pueden terminar exactamente en el orden solicitado, o no tal como se explico, al estar en false no hay criterio para selección del siguiente.
 - En esta ejecución puede observarse que vuelve a adquirir el LOCK el mismo hilo que lo soltó.

Modifying Lock fairness

- Ejecución del proyecto. NON-FAIR MODE

Console

```
<terminated> Main (41) [Java Application] C:\Program Files\Java\jre7\b  
Thread 0: Going to print a job  
Thread 0: PrintQueue: Printing a Job during 6 seconds  
Thread 1: Going to print a job  
Thread 2: Going to print a job  
Thread 3: Going to print a job  
Thread 4: Going to print a job  
Thread 5: Going to print a job  
Thread 6: Going to print a job  
Thread 7: Going to print a job  
Thread 8: Going to print a job  
Thread 9: Going to print a job  
Thread 1: PrintQueue: Printing a Job during 9 seconds  
Thread 1: PrintQueue: Printing a Job during 1 seconds  
Thread 1: The document has been printed  
Thread 2: PrintQueue: Printing a Job during 3 seconds  
Thread 2: PrintQueue: Printing a Job during 6 seconds  
Thread 2: The document has been printed  
Thread 3: PrintQueue: Printing a Job during 2 seconds  
Thread 3: PrintQueue: Printing a Job during 4 seconds  
Thread 3: The document has been printed  
Thread 4: PrintQueue: Printing a Job during 5 seconds  
Thread 4: PrintQueue: Printing a Job during 5 seconds  
Thread 4: The document has been printed
```

Controlling concurrent access to a resource

- Sincronización: más de una tarea concurrente comparte un recurso.
 - El bloque de código que accede ese recurso es llamado sección crítica.
 - Hemos analizado:
 - Synchronized
 - Locks.
 - Veremos mecanismos de más alto nivel.
 - SEMAFOROS.

Controlling concurrent access to a resource

- Un semáforo puede comprenderse como un contador que protege el acceso a uno o más recursos compartidos. Edsger Dijkstra 1965.
- Cuando un hilo quiere acceder a un recurso compartido primero debe adquirir el semáforo.
 - Si el valor del contador es mayor que 0, se decrementa y se permite el acceso al recurso.
 - Si el valor del contador es igual a 0, el semáforo pone el hilo a dormir en una cola.

Controlling concurrent access to a resource

- Cuando un hilo finaliza el uso del recurso debe liberar el semáforo. Esta operación incrementa el valor del contador del semáforo.
- A continuación veremos un ej. de uso de la clase semáforo, pero con un semáforo con posibles valores 0 y 1, llamado “semáforo binario”.
- Este semáforo protegerá el acceso a una impresora. Con una cola de impresión.

Controlling concurrent access to a resource

- Clases: **PrintQueue.**

Declare un objeto Semáforo.

Implemente constructor y el método printJob. Adentro de este método primero, trate de adquirir el semáforo.

Luego

Imprime

Y final//

libera

semáforo

```
Main.java  Job.java  PrintQueue.java  ✕
1  package com.packtpub.java7.concurrency.chapter3.recipe1.task;
2  import java.util.concurrent.Semaphore;
3  * This class implements the PrintQueue using a Semaphore to control the
4
5  public class PrintQueue {
6      * Semaphore to control the access to the queue
7      private final Semaphore semaphore;
8      * Constructor of the class. Initializes the semaphore
9      public PrintQueue(){
10         semaphore=new Semaphore(1);
11     }
12     * Method that simulates printing a document
13     public void printJob (Object document){
14         try {
15             // Get the access to the semaphore. If other job is printing, this
16             // thread sleep until get the access to the semaphore
17             semaphore.acquire();
18
19             Long duration=(long)(Math.random()*10);
20             System.out.printf("%s: PrintQueue: Printing a Job during %d seconds\n",Thread.currentThread().getName(),duration);
21             Thread.sleep(duration);
22             TimeUnit.SECONDS.sleep(duration);
23         } catch (InterruptedException e) {
24             e.printStackTrace();
25         } finally {
26             // Free the semaphore. If there are other threads waiting for this semaphore,
27             // the JVM selects one of this threads and give it the access.
28             semaphore.release();
29         }
30     }
31 }
32
33 }
```

Controlling concurrent access to a resource

- Clases: **Job (runnable)**

Implemente el constructor y el método run.

Este método imprime mensaje, llama al método printJob y finaliza con otra impresión

```

Main.java  Job.java  PrintQueue.java
1 package com.packtpub.java7.concurrency.chapter3.recipe1.task;
2
3
4+ * This class simulates a job that send a document to print.
5
6
7 public class Job implements Runnable {
8
9
10+ * Queue to print the documents
11
12 private PrintQueue printQueue;
13
14
15+ * Constructor of the class. Initializes the queue
16
17
18 public Job(PrintQueue printQueue){
19     this.printQueue=printQueue;
20 }
21
22
23+ * Core method of the Job. Sends the document to the print queue and waits
24
25
26 @Override
27 public void run() {
28     System.out.printf("%s: Going to print a job\n",Thread.currentThread().getName());
29     printQueue.printJob(new Object());
30     System.out.printf("%s: The document has been printed\n",Thread.currentThread().getName());
31 }
32 }
33
```


Controlling concurrent access to a resource

- Clases: **Main.**

Crea objeto del tipo PrintQueue.

Crea 10 hilos y
objetos Job.

Lanza ejecución de
los 10 hilos

```

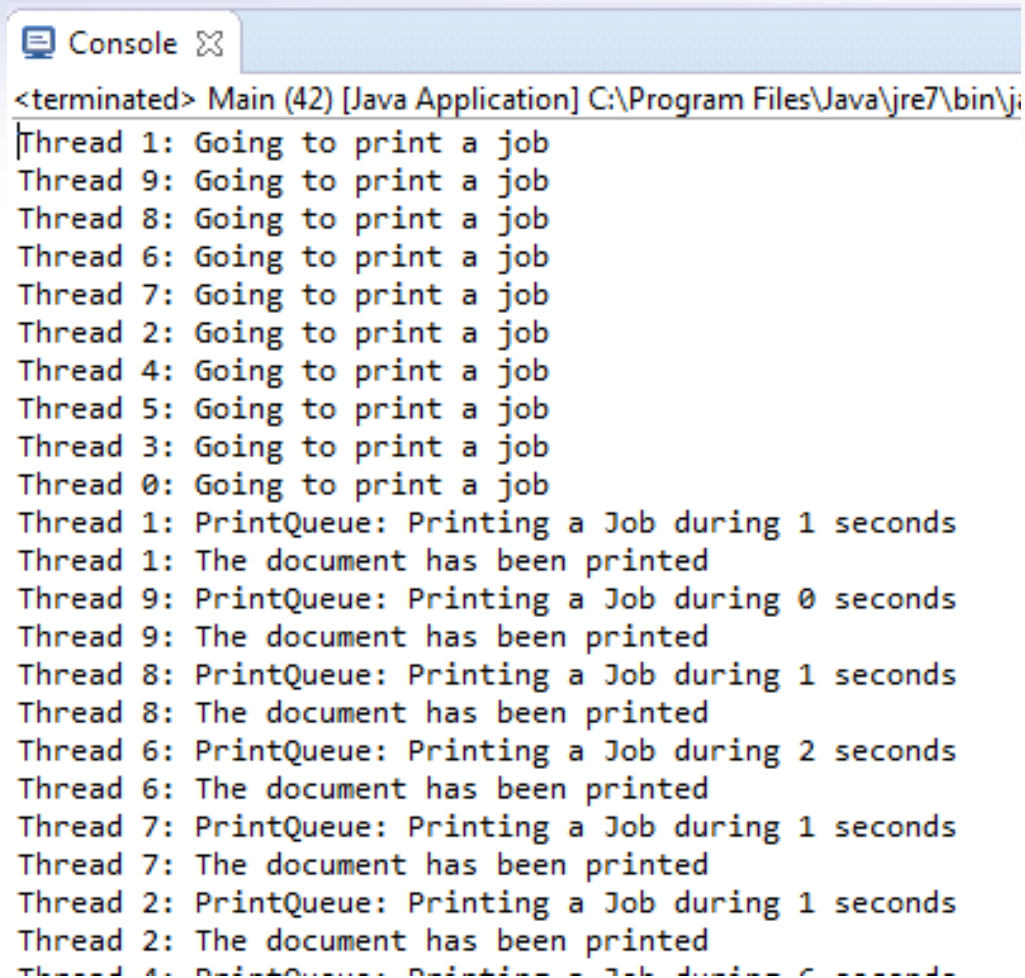
Main.java  Job.java  PrintQueue.java
1 package com.packtpub.java7.concurrency.chapter3.recipe1.core;
2
3 import com.packtpub.java7.concurrency.chapter3.recipe1.task.Job;
4
5
6 * Main class of the example.
7 public class Main {
8
9     * Main method of the class. Run ten jobs in parallel that
10    public static void main (String args[]){
11
12        // Creates the print queue
13        PrintQueue printQueue=new PrintQueue();
14
15        // Creates ten Threads
16        Thread thread[]=new Thread[10];
17        for (int i=0; i<10; i++){
18            thread[i]=new Thread(new Job(printQueue),"Thread "+i);
19        }
20
21        // Starts the Threads
22        for (int i=0; i<10; i++){
23            thread[i].start();
24        }
25    }
26 }
27
28
29
30
31
32
33
34
```

Controlling concurrent access to a resource

- Ejecución del proyecto y conclusión.
 - Lo importante de este ejemplo esta en el método `printJob` de la clase `PrintQueue`. Allí hay 3 pasos que se deben seguir para trabajar con semáforos.
 1. Adquirir el semáforo con el método **`acquire()`**.
 2. Operar con el recurso compartido.
 3. Finalmente, liberar el semáforo con el método **`release()`**.
- Otro punto importante es la INICIALIZACION del semáforo en el constructor. Valor inicial 1 binario.

Controlling concurrent access to a resource

- Ejecución



The screenshot shows a Java console window titled "Console" with a close button. The text inside the console is as follows:

```
<terminated> Main (42) [Java Application] C:\Program Files\Java\jre7\bin\j  
Thread 1: Going to print a job  
Thread 9: Going to print a job  
Thread 8: Going to print a job  
Thread 6: Going to print a job  
Thread 7: Going to print a job  
Thread 2: Going to print a job  
Thread 4: Going to print a job  
Thread 5: Going to print a job  
Thread 3: Going to print a job  
Thread 0: Going to print a job  
Thread 1: PrintQueue: Printing a Job during 1 seconds  
Thread 1: The document has been printed  
Thread 9: PrintQueue: Printing a Job during 0 seconds  
Thread 9: The document has been printed  
Thread 8: PrintQueue: Printing a Job during 1 seconds  
Thread 8: The document has been printed  
Thread 6: PrintQueue: Printing a Job during 2 seconds  
Thread 6: The document has been printed  
Thread 7: PrintQueue: Printing a Job during 1 seconds  
Thread 7: The document has been printed  
Thread 2: PrintQueue: Printing a Job during 1 seconds  
Thread 2: The document has been printed  
Thread 4: PrintQueue: Printing a Job during 0 seconds
```

Controlling concurrent access to a resource

- Otros métodos interesantes:
 - `acquireUninterruptibly()`: en el método `acquire()` cuando el contador interno está en 0, el hilo se bloquea. Durante este tiempo el hilo puede ser interrumpido y lanzar una excepción, con éste método el hilo ignora la interrupción y no lanza excepciones.
 - `tryAcquire()`: este método intenta adquirir el semáforo. Si puede el mismo devuelve `true` y lo toma. Caso contrario devuelve `false` en vez de bloquearse.

Controlling concurrent access to a resource

- SEMAPHORE Fairness:

- El concepto de Fairness es usado en java, en todas las clases que pueden tener varios hilos bloqueados esperando por la liberación de un recurso de sincronización (ej un semaforo).
- Como ocurre con otras primitivas, por defecto false, no hay criterio para selección de hilo, true el que más espero.
- Por lo tanto el constructor del semáforo puede aceptar 2 parámetros, el valor de inicialización y el boolean de fairness.