Programación Concurrente y de Tiempo Real

Guión de prácticas 9: API Java para la Concurrencia de Alto Nivel

Natalia Partera Jaime
Alumna colaboradora de la asignatura

Índice

1. Introducción

En los guiones anteriores hemos visto varios métodos para controlar la concurrencia utilizando algunas primitivas del lenguaje Java. A partir de Java 5, se incluyen algunos paquetes que implementan otros modos para el control de la concurrencia.

La mayoría de los métodos que describiremos en este guión pertenecen al paquete `java.util.concurrent` o a algún paquete derivado de éste. Entre los métodos que serán explicados se encuentran las variables atómicas, los semáforos, las barreras, los cierres para el control de la exclusión mutua o las clases contenedoras concurrentes de acceso sincronizado.

Estos métodos aportan algunas mejoras respecto a los mecanismos estudiados anteriormente. Entre ellas:

- Generan menos sobrecarga en tiempo de ejecución.
- Permiten controlar la exclusión mutua y la sincronización a un nivel más fino.
- Se implementan primitivas clásicas como los semáforos o las variables de condición en monitores.

Veamos, a continuación, los distintos mecanismos detenidamente.

2. Variables atómicas

Las variables atómicas se encuentran en el paquete `java.util.concurrent.atomic`. Como su propio nombre indica, en este paquete podemos encontrar tipos simples para definir variables que serán "tratadas" de forma atómica. Evidentemente, lo que se gestiona de manera atómica son las operaciones que se ejecutan sobre estas variables. Esto nos permite utilizar variables definidas con estos tipos para programar concurrentemente de forma segura, en algunos casos, con menos esfuerzo.

No todos los métodos que operan sobre estos tipos de datos soportan la concurrencia segura. Ésto sólo lo pueden garantizar los métodos que indican que se realizan de forma atómica. Como no existe una palabra reservada que nos indique si los métodos realizan las operaciones de forma atómica, tendremos que comprobarlo consultando la documentación de la clase correspondiente.

Este paquete incluye 4 clases principales y otras clases basadas en estas 4, como por ejemplo:

- `AtomicBoolean`: clase que representa un valor booleano (`boolean`) que puede ser actualizado atómicamente.
- `AtomicInteger`: clase que representa un valor entero (`int`) que puede ser actualizado atómicamente.
- `AtomicIntegerArray`: clase que representa un *array* de números enteros (`int`) donde los enteros pueden ser actualizados atómicamente.
- `AtomicLong`: clase que representa un valor entero largo (`long`) que puede ser actualizado atómicamente.
- `AtomicLongArray`: clase que representa un *array* de números enteros largos (`long`) donde los enteros largos pueden ser actualizados atómicamente.
- `AtomicReference<V>`: clase que representa una referencia a un objeto, el cual puede ser actualizado atómicamente.

Los métodos más destacados que podemos encontrar en todas o en algunas de estas clases son:

- Constructores con valor inicial por defecto y constructores con valor inicial indicado por el usuario.
- Métodos `get()` y `set()`: devuelve o asigna un valor a la variable de manera atómica. El dato es leído o escrito en memoria principal.
- Métodos `getAndSet()`, `compareAndSet()` y `weakCompareAndSet()`: estos métodos realizan una acción antes de asignar un cierto valor a la variable. En el caso de `getAndSet()`, devuelve el valor que tenía la variable antes de ser actualizada. Los otros dos métodos, comparan el valor que tenía la variable antes de asignarle el nuevo valor. El método `compareAndSet()` sólo actualiza la variable si el valor anterior coincide con el que se especifica en su primer argumento, devolviendo `true` si ha modificado el valor de la variable y `false` en caso contrario. El método `weakCompareAndSet()` puede que no consiga actualizar la variable a pesar de que contenga el valor esperado, en cuyo caso devuelve `false`. En caso contrario, devuelve `true`.
- Métodos `incrementAndGet()`, `decrementAndGet()`, `getAndIncrement()` y `getAndDecrement()`: estos métodos se encuentran en las clases basadas en enteros normales y largos. Incrementan o decrementan en 1 el valor almacenado por la variable y devuelven el valor de la variable, aunque no necesariamente en este orden. Estos 4 métodos equivalen a los operadores de pre-incremento, pre-decremento, post-incremento y post-decremento, respectivamente.
- Métodos `addAndGet()` y `getAndAdd()`: estos métodos se encuentran en las clases basadas en enteros normales y largos. Suman el número indicado como parámetro a la variable y devuelven el valor de la variable, aunque no necesariamente en este orden. Estos métodos equivalen a los operadores de pre-incremento y post-incremento con un valor dado, respectivamente. El valor especificado como argumento puede ser un número negativo.

El uso de estas clases es como el uso de cualquier otra clase:

```
/**
 * Clase VarAtomicas.
 *
 * @author Natalia Partera
 * @version 1.0
 */

import java.util.concurrent.atomic.*;

public class VarAtomicas extends Thread {

  //Atributos privados
  private AtomicBoolean condicion;
  private AtomicInteger entero;
  private AtomicInteger indice;
  private AtomicLong largo;
  private AtomicLongArray multiplicaciones;

  //Constructor de la clase
  public VarAtomicas(AtomicInteger at, AtomicLong al, AtomicLongArray ala) {
    condicion = new AtomicBoolean(true);
    entero = at;
    indice = new AtomicInteger(0);
```

```
    largo = al;
    multiplicaciones = ala;
  }

  //Método run
  public void run() {
    while(condicion.get()) {
        System.out.println(this.getName() + ", multiplicaciones[" + (indice.get()) +
          "] = " + multiplicaciones.getAndSet(indice.get(),
          largo.get() * entero.get()));
      if(indice.get() == (multiplicaciones.length() - 1)) {
        condicion.set(false);
      }
      else {
        System.out.println(this.getName() + ": Se ha multiplicado por " +
          entero.getAndDecrement());
        indice.incrementAndGet();
      }
    }
  }

}
```

Un programa que prueba la clase anterior es este:

```
/**
 * Programa que muestra el funcionamiento de las variables atómicas.
 *
 * @author Natalia Partera
 * @version 1.0
 */

import java.util.concurrent.atomic.*;

public class UsaVarAtomicas {
  public static void main (String[] args) {
    AtomicInteger ent = new AtomicInteger(20);
    AtomicLong lar = new AtomicLong(123456);
    AtomicLongArray arr = new AtomicLongArray(20);

    for(int i= 0; i < 5; ++i) {
      VarAtomicas h = new VarAtomicas(ent, lar, arr);
      h.start();
    }
  }
}
```

Ejercicio 1 Implemente una clase que represente una matriz usando variables atómicas. Programe hilos que multipliquen la matriz por un escalar dado. Cuando termine su ejercicio, compruébelo con la solución del apartado 7.1.

3. Semáforos

Java también proporciona una clase de alto nivel para trabajar con semáforos. Se trata de la clase `Semaphore` del paquete `java.util.concurrent.Semaphore`.

Como ya habrá visto a nivel teórico, un semáforo es un mecanismo de control de la concurrencia que consta de una variable entera y dos operaciones. La variable es la que indica cuántos procesos más pueden ejecutarse concurrentemente. Las operaciones `wait()` y `signal()` interpretan el estado de la variable y controlan la ejecución.

La operación `wait()` comprueba el valor de la variable y permite la ejecución del proceso, modificando después la variable, o pone al proceso en una cola de espera. La operación `signal()` despierta a los procesos en espera en el semáforo, si hay, y si no actualiza el valor de la variable para que se permita la ejecución de procesos.

Recordará también que los semáforos pueden ser de dos tipos: generales o binarios. En los semáforos generales, la variable puede tomar cualquier valor no negativo. Sin embargo, en los semáforos binarios, la variable sólo puede tomar los valores 0 ó 1.

Pues bien, la clase `Semaphore` de Java permite la implementación de semáforos generales. La variable del semáforo será inicializada al llamar al constructor y las operaciones `wait()` y `signal()` se corresponden con los métodos `acquire()` y `release()` respectivamente. Veamos los métodos principales de la clase `Semaphore`:

- `Semaphore(int permits)`: Constructor de la clase semáforo al que se le pasa el número de procesos que se permiten incialmente.
- `acquire()`: Método que adquiere un permiso para este semáforo, bloqueándolo hasta que alguno esté disponible o que el hilo sea interrumpido.
- `acquire(int permits)`: Método que adquiere un número dado de permisos para este semáforo, bloqueándolo hasta que todos estén disponibles o que el hilo sea interrumpido.
- `release()`: Método que cede un permiso, devolviéndolo al semáforo.
- `release(int permits)`: Método que cede un número dado de permisos, devolviéndolos al semáforo.
- `tryAcquire()`: Método que adquiere un permiso de este semáforo, si y sólo si el permiso está disponible en el momento de la invocación.
- `tryAcquire(int permits)`: Método que adquiere un número dado de permisos de este semáforo, si y sólo si los permisos están disponibles en el momento de la invocación.
- `availablePermits()`: Método que devuelve el número de permisos disponibles actualmente en este semáforo.

Para utilizar un semáforo, deberemos crearlo indicando de cuántos permisos dispondrá inicialmente. Luego podremos usar sobre el semáforo cualquiera de los otros métodos descritos anteriormente, con independencia de si se trata de un semáforo general o binario. El tipo de semáforo lo controla el programador, no el API de Java. Veamos un ejemplo de uso de estos métodos:

```
/**
 * Programa que muestra el funcionamiento de la clase Semaphore.
 *
 * @author Natalia Partera
 * @version 1.0
 */

import java.util.concurrent.*;

public class EjemSemaphore {
  public static void main (String[] args) throws InterruptedException{
    Semaphore sem = new Semaphore (3);
    System.out.println("El semáforo empieza con " + sem.availablePermits() +
      " permisos.");
    sem.acquire();
    System.out.println("Tras pedir un permiso, quedan " + sem.availablePermits() +
      " disponibles.");
    sem.tryAcquire(4);
    System.out.println("Intenta pedir 4 permisos. Tras ello quedan " +
      sem.availablePermits() + ".");
    sem.tryAcquire(2);
    System.out.println("Intenta pedir 2 permisos. Tras ello quedan disponibles " +
      sem.availablePermits() + " permisos.");
    sem.release();
    System.out.println("Se cede un permiso. Ahora quedan " + sem.availablePermits()
      + " disponibles.");
    sem.acquire(2);
    System.out.println("Pide dos permisos. Quedan " + sem.availablePermits());
    System.out.println("Fin del programa");
  }
}
```

Ejercicio 2 Cree un programa en el que solicite y ceda permisos alternativamente. Introduzca cuantos permisos debe solicitar o ceder cada vez desde teclado. Cuando termine, compruebe su resultado con el programa del apartado 7.2.

3.1. Protocolo de Control de la Exclusión Mutua con Semáforos

Para controlar la exclusión mutua utilizando semáforos, cada proceso debe intentar adquirir un permiso del semáforo, ejecutar su código y luego devolver el permiso. Además, para que sólo sea posible que se ejecute un proceso al mismo tiempo, el semáforo deberá ser binario, por lo que habrá que crearlo en el programa principal con un solo permiso. Esto se consigue en Java de la siguiente manera:

```
//Thread A
```

```
public void run() {
  try {
    s.acquire();
  } catch(InterruptedException ex) {}
  //Bloque de código a ejecutar en E.M.
  s.release();
}

//Thread B
public void run() {
  try {
    s.acquire();
  } catch(InterruptedException ex) {}
  //Bloque de código a ejecutar en E.M.
  s.release();
}

//En el programa principal, hacer siempre
  Semaphore s = new Semaphore(1);
```

A continuación puede ver un ejemplo del control de la exclusión mutua con semáforos. La siguiente clase es el recurso compartido entre los hilos.

```
/**
 * Clase Recurso.
 *
 * @author Natalia Partera
 * @version 1.0
 */

public class Recurso {

  //Atributos privados
  private int cont;
  private int max = 100;

  //Constructores
  public Recurso() {}

  public Recurso(int n) {
    max = n;
  }

  //Métodos
  public void aumentar(int i) {
    cont = cont + i;
  }

  public int valor() {
    return cont;
  }
```

```
  public int maximo() {
    return max;
  }
}
```

A continuación puede ver el programa que lanza los hilos y los ejecuta usando un semáforo:

```
/**
 * Programa que muestra el control de la exclusión mutua con semáforos.
 *
 * @author Natalia Partera
 * @version 1.0
 */

import java.util.concurrent.*;

class Hilo extends Thread {
  final Recurso r;
  Semaphore s;
  int incremento;

  public Hilo(Recurso rec, Semaphore sem, int suma) {
    r = rec;
    s = sem;
    incremento = suma;
  }

  public void run() {
    while(r.valor() < r.maximo()) {
      try {
        s.acquire();
        if(r.valor() < r.maximo()) {
          r.aumentar(incremento);
          System.out.print(r.valor() + " - ");
        }
        s.release();
      } catch(InterruptedException ex) {}
    }
  }
}

public class UsaRecurso {
  public static void main (String[] args) {
    Recurso r = new Recurso(200);
    Semaphore s = new Semaphore(1);

    for(int i= 1; i <= 5; ++i) {
      Hilo h = new Hilo(r, s, i);
      h.start();
    }

  }
```

```
}
```

Ejercicio 3 Implemente un programa haciendo uso de los semáforos ofrecidos por la clase `Semaphore` que simule la cola de impresión de una impresora. Lanzará varios hilos que mantendrán el recurso impresora ocupado durante un tiempo, indicando en cada caso cuando toma cada hilo el permiso para el recuso y cuando lo cede. Cuando termine, compruebe su solución con la ofrecida en el apartado 7.3.

3.2. Protocolo de Sincronización entre varios hilos con Semáforos

Si lo que queremos es que varios hilos se sincronicen entre sí utilizando semáforos, debemos seguir los siguientes pasos:

- Uno de los hilos deberá intentar adquirir el permiso.
- Otro hilo cederá el permiso cuando quiera sincronizarse con el primero.
- En el programa principal se inicializa el semáforo sin permisos.

Para implementar los pasos anteriores en Java, puede seguir este modelo:

```
//Thread A
public void esperarSeñal(Semaphore s) {
  try {
    s.acquire();
  } catch(InterruptedException ex) {}
}

//Thread B
public void enviarSeñal(Semaphore s) {
  s.release();
}

//En el programa principal, hacer siempre
  Semaphore s = new Semaphore(0);
```

El siguiente ejemplo ilustra la sincronización entre varios hilos usando semáforos:

```
/**
 * Programa que sincroniza dos hilos entre sí usando semáforos.
 *
 * @author Natalia Partera
 * @version 1.0
 */

import java.util.concurrent.*;

class HiloAumentador extends Thread {
  Semaphore s;
  int cont;
```

```
  public HiloAumentador(Semaphore semaforo) {
    s = semaforo;
  }

  public void cederPermiso(Semaphore s) {
    s.release();
  }

  public void run() {
    while(cont < 100) {
      ++cont;
      System.out.print(cont + " - ");
    }
    System.out.println();
    cederPermiso(s);
  }
}

class HiloDecrementador extends Thread {
  Semaphore s;
  int cont = 100;

  public HiloDecrementador(Semaphore semaforo) {
    s = semaforo;
  }

  public void pedirPermiso(Semaphore s) {
    try {
      s.acquire();
    } catch(InterruptedException ex) {}
  }

  public void run() {
    while(cont != 0) {
      pedirPermiso(s);
      while(cont > 0) {
        --cont;
        System.out.print(cont + " - ");
      }
      System.out.println();
    }
  }
}

public class UsaSemaforoSincro {
  public static void main (String[] args) {
    Semaphore s = new Semaphore(0);

    HiloAumentador hAumen = new HiloAumentador(s);
    HiloDecrementador hDecre = new HiloDecrementador(s);
    hDecre.start();
```

```
    hAumen.start();
  }
}
```

Ejercicio 4 Realice un programa que haga uso de varios hilos para calcular la inversa de una matriz de $3x3$. Un hilo deberá calcular la matriz de adjuntos, otro calculará la traspuesta de otra matriz y otro hilo calculará el determinante de una matriz dada. Sincronice los hilos con semáforos según sea necesario. Cuando haya terminado, puede comprobar su solución con la del apartado 7.4.

---

4. Barreras

A veces es necesario que varios hilos que se ejecutan concurrentemente, se paren antes de empezar la siguiente iteración hasta que todos terminen la ejecución de una parte dada. En estos casos podemos usar un mecanismo llamado barrera cíclica.

Una barrera es un punto de espera para hilos. Ningún hilo puede pasar de la barrera hasta que todos los hilos esperados llegan a ella. Por tanto, al pasar la barrera todos los hilos se encuentran sincronizados. La barrera cíclica se usa cuando es necesario sincronizar dichos hilos repetidamente.

Normalmente se utiliza una barrera cíclica para:

- Unificar resultados parciales.
- Iniciar la siguiente fase de ejecución simultánea.

Java proporciona barreras cíclicas con la clase `CyclicBarrier` implementada en el paquete `java.util.concurrent.CyclicBarrier`. A continuación puede ver cómo se implementa el protocolo de barrera:

```
//Código de Thread
public Hilo extends Thread {
  public Hilo(CyclicBarrier bar) {...}

  public void run() {
    try {
      int i = bar.await();
    }
    catch(BrokenBarrierException e) {}
    catch(InterruptedException e) {}
    //Código a ejecutar cuando se abre la barrera ...
  }
}

//En el programa principal, hacer siempre:
  int numHilos = n;    //Número de hilos que abren la barrera
  CyclicBarrier barrera = new CyclicBarrier(numHilos);
  new Hilo(barrera).start();
```

Un ejemplo de uso de este protocolo es el siguiente. Tenemos una piedra en el camino y necesitamos mucha gente para moverla. La clase que implementa los hilos a la espera de pasar por la barrera es la siguiente:

```
/**
 * Clase que se queda a la espera de poder pasar por la barrera cíclica.
 *
 * @author Natalia Partera
 * @version 1.0
 */

import java.util.concurrent.*;

public class Persona extends Thread {
  final CyclicBarrier piedra;
  boolean bloqueado = true;

  public Persona(CyclicBarrier barrera) {
    piedra = barrera;
  }

  public void run() {
    while(bloqueado) {
      try {
        piedra.await();
      } catch(BrokenBarrierException ex) {}
        catch(InterruptedException ex) {}
      bloqueado = false;
    }
    System.out.println("¡Hemos pasado!");
  }
}
```

Y este es el programa principal donde se encuentra la barrera:

```
/**
 * Programa que muestra el uso de una barrera cíclica para sincronizar.
 *
 * @author Natalia Partera
 * @version 1.0
 */

import java.util.concurrent.*;

public class UsaBarrera {
  public static void main (String[] args) {
    int numHilos = 10;

    System.out.println("¡Oh! Hay una enorme piedra en el camino. Debemos moverla " +
      "para continuar, pero pesa demasiado. Tenemos que esperar a que más gente " +
      "nos ayude a moverla.");
    CyclicBarrier piedra = new CyclicBarrier(numHilos);
```

```
    for (int i = 0; i <  numHilos; ++i) {
      new Persona(piedra).start();
      System.out.println("Ha llegado una persona más.");
    }
  }
}
```

Ejercicio 5 Realice un programa que calcule el resultado de elevar a n una matriz cuadrada. Controle con una barrera que no se empieza a calcular M^{i+1} hasta que no se haya terminado de calcular M^i. Compruebe su solución con la que se muestra en el apartado 7.5.

---

5. Cerrojos

Otra forma de conseguir la sincronización entre los hilos es bloqueándolos con cerrojos o haciendo que esperen a que se cumpla una condición antes de seguir. El paquete `java.util.concurrent.locks` contiene elementos que nos ayudan a controlar la ejecución de los hilos sin tener que utilizar bloques sincronizados ni monitores.

Los cerrojos permiten, o no, que un código concreto pueda ser ejecutado concurrentemente a otras partes de código o no. En esta clase se proporciona la interfaz `Lock` que nos permite controlar el acceso a recursos compartidos por múltiples hilos. Algunas de las clases de este paquete implementan esta interfaz.

Algunas de las clases e interfaces más destacadas de este paquete son:

- `ReentrantLock`: clase que proporciona cerrojos de exclusióm mutua de semántica equivalente a `synchronized`, pero con un manejo más sencillo.
- `LockSupport`: proporciona primitivas de bloqueo de hilos que permiten al programador diseñar sus clases de sincronización y cerrojos propios.
- `Condition`: interfaz cuyas instancias se usan asociadas a `locks`. Implementan variables de condición y proporcionan una alternativa de sincronización a los métodos `wait`, `notify` y `notifyAll` de la clase `Object`.

Con la clase `LockSupport` podemos sincronizar los hilos como si de un semáforo binario se tratase. Además, permite parar y reanudar la ejecución de hilos, como lo hacemos con `Thread.suspend()` y `Thread.resume()`, pero sin los problemas que estos métodos derogados nos causan.

Veamos un poco más a fondo en qué consisten la clase `ReentrantLock` y la interfaz `Condition`.

5.1. Clase `ReentrantLock`

La clase `ReentrantLock` proporciona cerrojos "reentrantes". Estos cerrojos pueden invocar a un método sincronizado y adquirir su cerrojo durante la ejecución de otro método sincronizado que se controla con el mismo cerrojo y que ya posee. De este modo podemos conseguir, por ejemplo, la recursión de métodos sincronizados.

Algunos de los métodos prinicpales para el manejo de esta clase son:

- `lock()`: método heredado de la interfaz `Lock` para adquirir el cerrojo. Si el cerrojo está libre o ya está adquirido por ese hilo, el objeto adquiere el cerrojo.
- `unlock()`: método heredado de la interfaz `Lock` para liberar el cerrojo.
- `isLocked()`: método que indica si ese cerrojo está actualmente adquirido por algún hilo.
- `newCondition()`: método heredado de la interfaz `Lock` para establecer una condición, objeto de la clase `Condition`, para el uso del cerrojo. Devuelve la variable de condición asociada al cerrojo.

Con `ReentrantLock` se puede conseguir la exclusión mutua sobre un recurso compartido, para ello debemos seguir los siguientes pasos:

- Definir el recurso compartido donde sea necesario.
- Definir un objeto `c` de clase `ReentrantLock` para el control de la exclusión mutua.
- Acotar la sección crítica con entre las llamadas a `c.lock()` y a `c.unlock()`.

El siguiente código ilustra el uso de los pasos anteriores:

```
class Recurso {
  private final ReentrantLock cerrojo = new ReentrantLock();
  // ...

  public void metodo() {
    cerrojo.lock();
    try {
      //... cuerpo del metodo
    } finally {
      cerrojo.unlock();
    }
  }
}
```

El siguiente ejemplo es una modificación del programa de ejemplo del uso de semáforos para controlar la exclusión mutua. Observe las diferencias en el código que existen entre este programa y el del control de la exclusión mutua con semáforos.

```
/**
 * Clase Recurso.
 *
 * @author Natalia Partera
 * @version 1.0
 */

import java.util.concurrent.locks.*;

public class Recurso {

  //Atributos privados
  private int cont;
```

```
  private int max = 100;
  private final ReentrantLock cerrojo = new ReentrantLock();

  //Constructores
  public Recurso() {}

  public Recurso(int n) {
    max = n;
  }

  //Métodos
  public void aumentar(int i) {
    cerrojo.lock();
    try {
      cont = cont + i;
      System.out.print(cont + " - ");
    } finally {
      cerrojo.unlock();
    }
  }

  public int valor() {
    int v = -1;
    cerrojo.lock();
    try {
      v = cont;
    } finally {
      cerrojo.unlock();
      return v;
    }
  }

  public int maximo() {
    return max;
  }
}
```

Y la clase que implementa los hilos y el programa principal:

```
/**
 * Programa que muestra el uso de los cerrojos.
 *
 * @author Natalia Partera
 * @version 1.0
 */

import java.util.concurrent.locks.*;

class Hilo extends Thread {
  Recurso r;
  int incremento;
```

```
  public Hilo() {}

  public Hilo(Recurso rec, int suma) {
    r = rec;
    incremento = suma;
  }

  public void run() {
    while(r.valor() < r.maximo()) {
      r.aumentar(incremento);
    }
  }
}

public class UsaRecurso {
  public static void main (String[] args) {
    Recurso r = new Recurso(200);

    for(int i= 1; i <= 5; ++i) {
      Hilo h = new Hilo(r, i);
      h.start();
    }
  }
}
```

Ejercicio 6 Modifique el ejercicio 3 controlando esta vez la cola de impresión con el uso de cerrojos. Cuando acabe, compruebe su solución con la ofrecida en el apartado 7.6.

5.2. Interfaz `Condition`

La interfaz `Condition` proporciona variables de condición que pueden usarse asociadas a un cerrojo. Por otra parte, si una clase implementa esta interfaz puede proporcionar un comportamiento distinto.

Normalmente el uso de está interfaz está asociado a un cerrojo derivado de la interfaz `Lock`. La forma de usar una condición es creando una instancia de una clase como `ReentrantLock` y llamando al método `newCondition()` desde el objeto cerrojo ya creado. Este método devolverá un objeto condición, que es específico para ese cerrojo.

Las operaciones más destacables de esta interfaz son:

- `await()`: Pone en espera al hilo actual hasta que sea despertado o interrumpido.
- `signal()`: Despierta un hilo que esté en espera.
- `signalAll()`: Despierta a todos los hilos en espera.

Un ejemplo del uso de la interfaz `Condition` es el siguiente. Simula la cola de una caja en un supermercado o similar. El cajero/a atiende a los clientes si hay clientes en la cola.

```
/**
 * Clase LineaCaja.
 *
 * @author Natalia Partera
 * @version 1.0
 */

import java.util.concurrent.locks.*;

public class LineaCaja {

  //Atributos privados
  private final ReentrantLock cerrojo = new ReentrantLock();
  private final Condition noVacia = cerrojo.newCondition();
  private final Condition noLlena = cerrojo.newCondition();

  private final int[] colaClientes = new int[50];
  private int clientesEnCola = 0;
  private int ultimoCliente = 0;

  //Constructores
  public LineaCaja() {}

  //Métodos
  public void llegaCliente() throws InterruptedException {
    cerrojo.lock();
    try {
      while(clientesEnCola == (colaClientes.length - 1))
        noLlena.await();
      colaClientes[clientesEnCola] = ultimoCliente;
      System.out.println("Ha llegado un nuevo cliente: " + ultimoCliente);
      ++ultimoCliente;
      ++clientesEnCola;
      noVacia.signal();
    } finally {
      cerrojo.unlock();
    }
  }

  public void atenderCliente() throws InterruptedException {
    cerrojo.lock();
    try {
      while(clientesEnCola == 0)
        noVacia.await();
      int cliente = colaClientes[0];
      System.out.println("Se atiende al cliente " + cliente);
      //Los clientes avanzan una posición en la cola.
      int i = 1;
      for(; colaClientes[i] != 0; ++i) {
```

```
        colaClientes[i-1] = colaClientes[i];
      }
      colaClientes[i] = 0;
      --clientesEnCola;
      noLlena.signal();
    } finally{
      cerrojo.unlock();
    }
  }
}
```

El programa que usa esta clase es el siguiente:

```
/**
 * Programa que muestra el uso de los cerrojos.
 *
 * @author Natalia Partera
 * @version 1.0
 */

import java.util.concurrent.locks.*;

class Cliente extends Thread {
  LineaCaja linea;

  public Cliente(LineaCaja lc) {
    linea = lc;
  }

  public void run() {
    for(int i = 1; i <= 10; ++i) {
      try {
        linea.llegaCliente();
      } catch (InterruptedException ex) {}
    }
  }
}

class Cajero extends Thread {
  LineaCaja linea;

  public Cajero(LineaCaja lc) {
    linea = lc;
  }

  public void run() {
    while(true) {
      try {
        linea.atenderCliente();
      } catch (InterruptedException ex) {}
    }
  }
```

```
}

public class UsaLineaCaja {
  public static void main (String[] args) {
    LineaCaja lc = new LineaCaja();
    Cajero cajero = new Cajero(lc);
    cajero.start();

    for(int i= 0; i < 10; ++i) {
      Cliente cliente = new Cliente(lc);
      cliente.start();
    }
  }
}
```

Ejercicio 7 Modifique el ejercicio anterior para que tenga en cuenta si hay elementos en la cola de impresión o si está completa y no puede añadir más tareas. Suponga que el límite de tareas de la cola de impresión es 10. Cuando acabe, comprueba su solución con la del apartado 7.7.

---

6. Contenedores sincronizados y concurrentes

Hasta ahora hemos conseguido la sincronización y la exclusión mutua de tipos sencillos. Sin embargo, a veces esto no es suficiente. Cuando nos interese garantizar un acceso concurrente seguro desde múltiples hilos a elementos de una clase contenedora, utilizaremos las collecciones concurrentes. Podemos encontrarlas en el paquete `java.util.concurrent`. En este paquete podemos encontrar clases sincronizadas, que implementan la interfaz `Serializable`, y clases concurrentes.

Los contenedores concurrentes se encuentran en Java desde la versión 5.0 y proporcionan sincronización de un modo diferente a como lo hacen los contenedores sincronizados. Los contenedores concurrentes usan un mecanismo de bloqueo de grano más fino que el usado en los contenedores sincronizados. Mientras que los contenedores sincronizados bloquean cada método bajo un mismo cerrojo, por lo que sólo permite a un hilo que acceda al contenedor. Los contenedores concurrentes permiten el acceso de más o menos hilos al contenedor según sean observadores (lectores) o modificadores (escritores). De este modo:

- Cualquier número de hilos observadores pueden acceder al contenedor concurrentemente.
- Hilos lectores pueden acceder al contenedor concurrentemente con hilos modificadores.
- Sólo un número limitado de hilos modificadores pueden acceder al contenedor concurrentemente.

Algunas de las clases contenedoras que podemos encontrar en el paquete son las siguientes:

- `ConcurrentHashMap`: conjunto de elementos con clave de acceso concurrente.
- `ConcurrentSkipListMap`: conjunto de elementos con clave, ordenados según su clave y de acceso concurrente.
- `ConcurrentSkipListSet`: lista de elementos ordenados y de acceso concurrente.

- `CopyOnWriteArrayList`: lista de elementos copiados desde una lista ya existente en una nueva lista.
- `CopyOnWriteArraySet`: conjunto de elementos copiados desde un conjunto ya existente en un nuevo conjunto.

Para un uso correcto de las colecciones, siga estas recomendaciones:

- Usar colecciones a través de interfaces (usabilidad del código).
- Usar colecciones no sincronizadas no mejora mucho el rendimiento.
- En los problemas del tipo "Productor-Consumidor" utilice colas.
- Minimice la sincronización explícita.
- Si con una colección retarda sus algoritmos, distribuya los datos y use varias colecciones.

Este paquete también proporciona colas, tanto sincronizadas como concurrentes. Veamos sus detalles.

6.1. Colas sincronizadas

El paquete `java.util.concurrent` proporciona multitud de colas distintas. Esto se debe a que las colas suelen ser usadas en los problemas del tipo "Productor-Consumidor". Todas las colas del citado paquete son sincronizadas, y algunas son, además, concurrentes.

Las colas son estructuras del tipo FIFO, en las que el elemento que lleva más tiempo en la cola es el primero en salir, y los elementos nuevos son colocados al final. Las colas que se incluyen en este paquete pueden tener un tamaño limitado a cierto número de elementos ó ilimitado. Proporcionan, también, notificación a hilos según cambia su contenido.

Las principales colas que se incluyen en este paquete son:

- `LinkedBlockingQueue`: cola de bloqueo basada en nodos enlazados, lo cual permite más flexibilidad. Se trata de una cola de cuya capacidad máxima es opcional, es decir, se puede especificar al construir la cola. Si no se especifica en la llamada al constructor, se considera infinita, pudiendo tener hasta $2^{31} - 1$ elementos.
- `ArrayBlockingQueue`: cola de bloqueo almacenada en un vector de tamaño fijo, que se especifica al construir la cola. Una vez creada, no es posible redimensionarla.
- `SynchronousQueue`: cola de bloqueo sin capacidad interna. La peculiaridad de esta cola reside en que para realizar una inserción en la cola, es necesario que también se quiera eliminar un elemento, y viceversa.
- `PriorityBlockingQueue`: cola de bloqueo infinita basada en el funcionamiento de las colas de prioridad de la clase `PriorityQueue`. No permite insertar elementos que no sigan el orden natural o que sean no comparables.
- `DelayQueue`: cola de bloqueo infinita de elementos retrasados. En esta cola los primeros elementos en salir son los primeros cuyo retraso haya expirado antes.

En las colas mencionadas arriba podemos encontrar los siguientes métodos para su manejo. Tenga en cuenta que el comportamiento de estos elementos puede variar un poco de una cola a otra según sea su funcionamiento.

- `offer()`: inserta el elemento inmediatamente en la cola siempre que sea posible. Si no queda espacio, no inserta el elemento.
- `poll()`: devuelve y elimina un elemento del principio de la cola.
- `put()`: inserta el elemento en la cola, esperando a que quede algún hueco libre si fuera necesario.
- `take()`: devuelve y elimina un elemento del principio de la cola, esperando si es necesario a que el elemento esté disponible.

Los métodos `put()` y `take()` pueden bloquear la cola si no se dan las condiciones necesarias para su ejecución.

Para ver el modo de implementar una cola, retomaremos el ejemplo anterior y lo implementamos con un objeto de la clase `LinkedBlockingQueue`:

```
/**
 * Clase LineaCaja.
 *
 * @author Natalia Partera
 * @version 1.0
 */

import java.util.concurrent.*;
import java.util.concurrent.locks.*;

public class LineaCaja {

  //Atributos privados
  private final ReentrantLock cerrojo = new ReentrantLock();
  private final Condition noVacia = cerrojo.newCondition();
  private final Condition noLlena = cerrojo.newCondition();

  private final LinkedBlockingQueue<Integer> colaClientes;
  private int clientesEnCola = 0;
  private Integer ultimoCliente = 0;

  //Constructores
  public LineaCaja() {
    colaClientes = new LinkedBlockingQueue<Integer>(50);
  }

  //Métodos
  public void llegaCliente() throws InterruptedException {
    cerrojo.lock();
    try {
      while(colaClientes.size() == 50)
        noLlena.await();
      colaClientes.put(ultimoCliente);
```

```
      System.out.println("Ha llegado un nuevo cliente: " +
        ultimoCliente.toString());
      ++ultimoCliente;
      ++clientesEnCola;
      noVacia.signal();
    } finally {
      cerrojo.unlock();
    }
  }

  public void atenderCliente() throws InterruptedException {
    cerrojo.lock();
    try {
      while(clientesEnCola == 0)
        noVacia.await();
      int cliente = colaClientes.take();
      System.out.println("Se atiende al cliente " + cliente);
      --clientesEnCola;
      noLlena.signal();
    } finally{
      cerrojo.unlock();
    }
  }
}
```

El programa principal, no sufre cambios:

```
/**
 * Programa que muestra el uso de los cerrojos.
 *
 * @author Natalia Partera
 * @version 1.0
 */

import java.util.concurrent.locks.*;

class Cliente extends Thread {
  LineaCaja linea;

  public Cliente(LineaCaja lc) {
    linea = lc;
  }

  public void run() {
    for(int i = 1; i <= 10; ++i) {
      try {
        linea.llegaCliente();
      } catch (InterruptedException ex) {}
    }
  }
}
```

```
class Cajero extends Thread {
  LineaCaja linea;

  public Cajero(LineaCaja lc) {
    linea = lc;
  }

  public void run() {
    while(true) {
      try {
        linea.atenderCliente();
      } catch (InterruptedException ex) {}
    }
  }
}

public class UsaLineaCaja {
  public static void main (String[] args) {
    LineaCaja lc = new LineaCaja();
    Cajero cajero = new Cajero(lc);
    cajero.start();

    for(int i= 0; i < 10; ++i) {
      Cliente cliente = new Cliente(lc);
      cliente.start();
    }
  }
}
```

Ejercicio 8 Rehaga el ejercicio anterior sobre una cola de impresión e impleméntelo usando una cola sincronizada a su elección. Puede comprobar su solución con la posibilidad ofertada en el apartado 7.8.

---

7. Soluciones de los ejercicios

Compruebe los resultados de sus ejercicios con estas soluciones.

7.1. Ejercicio 1

Esta es la clase que implementa la matriz:

```
/**
 * Clase Matriz.
 *
 * @author Natalia Partera
 * @version 1.0
 */

import java.util.*;
import java.util.concurrent.atomic.*;

public class Matriz extends Thread {
  //Atributos privados
  private AtomicReferenceArray<AtomicLongArray> matriz;
  private int dim;

  //Constructor de la clase
  public Matriz(int dimension) {
    matriz = new AtomicReferenceArray(dimension);
    dim = dimension;

    for(int i = 0; i < dim; ++i) {
      AtomicLongArray fila = new AtomicLongArray(dim);
      matriz.set(i, fila);
    }
  }

  public void inicializarMatriz() {
    for(int i = 0; i < dim; ++i) {
      AtomicLongArray fila = new AtomicLongArray(dim);
      for(int j = 0; j < dim; ++j){
        fila.set(j, i+j);
      }
      matriz.set(i, fila);
    }
  }

  public void inicializarMatrizAleatoria() {
    Random aleat = new Random();
    for(int i = 0; i < dim; ++i) {
      AtomicLongArray fila = new AtomicLongArray(dim);
      for(int j = 0; j < dim; ++j){
        fila.set(j, aleat.nextInt(100));
      }
      matriz.set(i, fila);
```

```
    }
  }

  public void imprimirMatriz() {
    for(int i = 0; i < dim; ++i) {
      for(int j = 0; j < dim; ++j){
        System.out.print(" " + (matriz.get(i)).get(j) + " ");
      }
      System.out.println();
    }
  }

  public void multiplicarFilaPorEscalar(int f, int k) {
    if(k >=0 && k < dim) {
      AtomicLongArray fila =new AtomicLongArray(dim);
      for(int i = 0; i < dim; ++i) {
        fila.set(i, matriz.get(f).get(i));
      }
      for(int j = 0; j < dim; ++j) {
        fila.set(j, fila.get(j) * k);
      }
      matriz.set(f, fila);
    }
  }

  public int dimension() {
    return dim;
  }
}
```

Y este es el programa que multiplica la matriz por un escalar:

```
/**
 * Programa que multiplica una matriz por un escalar.
 *
 * @author Natalia Partera
 * @version 1.0
 */

import java.util.concurrent.atomic.*;

class Hilo extends Thread {
  final Matriz matriz;
  int fila;
  int cte;

  public Hilo(Matriz m, int f, int c) {
    matriz = m;
    fila = f;
    cte = c;
  }
```

```
  public void run() {
    matriz.multiplicarFilaPorEscalar(fila, cte);
  }
}

public class MatrizPorEscalar {
  public static void main (String[] args) throws InterruptedException {
    int dimension = 4;
    Matriz m = new Matriz(dimension);

    m.inicializarMatrizAleatoria();
    m.imprimirMatriz();

    int constante = 2;
    Hilo h1 = new Hilo(m, 0, constante);
    h1.start();
    Hilo h2 = new Hilo(m, 1, constante);
    h2.start();
    Hilo h3 = new Hilo(m, 2, constante);
    h3.start();
    Hilo h4 = new Hilo(m, 3, constante);
    h4.start();
    h1.join();
    h2.join();
    h3.join();
    h4.join();
    System.out.println("Matriz m por escalar " + constante + ":");
    m.imprimirMatriz();
  }
}
```

7.2. Ejercicio 2

Esta es una posible implementación del ejercicio 2:

```
/**
 * Programa que muestra el funcionamiento de la clase Semaphore.
 *
 * @author Natalia Partera
 * @version 1.0
 */

import java.util.*;
import java.util.concurrent.*;

public class EjerSemaphore {
  public static void main (String[] args) throws InterruptedException{
    Scanner teclado = new Scanner(System.in);
    String seguir = "s";
```

```
    System.out.println("Introduzca con cuántos permisos comenzará el semáforo:");
    int perm = teclado.nextInt();
    Semaphore sem = new Semaphore(perm);

    while(seguir.contentEquals("s") || seguir.contentEquals("S")) {
      System.out.println("El semáforo tiene " + sem.availablePermits() +
      " permisos.");
      System.out.println("Indique cuántos permisos solicitar: ");
      perm = teclado.nextInt();
      sem.tryAcquire(perm);
      System.out.println("El semáforo tiene " + sem.availablePermits() +
      " permisos.");
      System.out.println("Indique cuántos permisos ceder: ");
      perm = teclado.nextInt();
      sem.release(perm);
      System.out.println("El semáforo tiene " + sem.availablePermits() +
      " permisos.");
      System.out.println("Si quiere continuar, introduzca ''S'' ó ''s'': ");
      seguir = teclado.next();
    }
  }
}
```

7.3. Ejercicio 3

Esta es la clase que representa la impresora:

```
/**
 * Clase Impresora.
 * [Ejercicio]
 * @author Natalia Partera
 * @version 1.0
 */

public class Impresora {
  //Constructores
  public Impresora() {}

  //Métodos
  public void imprimir() {
    for(int i = 0; i < 100000; ++i) {}
  }
}
```

A continuación puede ver el programa que manda a ejecutar las tareas a la impresora:

```
/**
 * Programa que controla una cola de impresión usando semáforos.
 *
 * @author Natalia Partera
```

```
 * @version 1.0
 */

import java.util.concurrent.*;

class Hilo extends Thread {
  final Impresora impresora;
  Semaphore semaforo;
  int numTarea;

  public Hilo(Impresora i, Semaphore sem, int num) {
    impresora = i;
    semaforo = sem;
    numTarea = num;
  }

  public void run() {
    try {
      semaforo.acquire();
      System.out.println("La tarea " + numTarea + " obtiene el permiso.");
      impresora.imprimir();
      System.out.println("La tarea " + numTarea + " ha sido impresa por completo.");
      semaforo.release();
    } catch(InterruptedException ex) {}
  }
}

public class UsaImpresora {
  public static void main (String[] args) {
    Impresora imp = new Impresora();
    Semaphore s = new Semaphore(1);

    for(int i= 1; i <= 10; ++i) {
      Hilo h = new Hilo(imp, s, i);
      h.start();
    }
  }
}
```

7.4. Ejercicio 4

Para el cálculo de la matriz inversa, hemos cambiado la clase `Matriz` para que trabaje con números reales. Esta es la nueva versión de la clase `Matriz`:

```
/**
 * Clase Matriz.
 *
 * @author Natalia Partera
 * @version 1.0
 */
```

```
import java.util.*;
import java.util.concurrent.atomic.*;

public class Matriz extends Thread {
  //Atributos privados
  private AtomicReferenceArray<AtomicReferenceArray<Float>> matriz;
  private int dim;

  //Constructor de la clase
  public Matriz(int dimension) {
    matriz = new AtomicReferenceArray<AtomicReferenceArray<Float>>(dimension);
    dim = dimension;

    for(int i = 0; i < dim; ++i) {
      AtomicReferenceArray<Float> fila = new AtomicReferenceArray<Float>(dim);
      matriz.set(i, fila);
    }
  }

  public void inicializarMatriz() {
    for(int i = 0; i < dim; ++i) {
      AtomicReferenceArray<Float> fila = new AtomicReferenceArray<Float>(dim);
      for(int j = 0; j < dim; ++j){
        Integer num = new Integer(i+j);
        fila.set(j, num.floatValue());
      }
      matriz.set(i, fila);
    }
  }

  public void inicializarFila(int f, AtomicReferenceArray<Float> fila) {
    matriz.set(f, fila);
  }

  public void inicializarMatrizAleatoria() {
    Random aleat = new Random();
    for(int i = 0; i < dim; ++i) {
      AtomicReferenceArray<Float> fila = new AtomicReferenceArray<Float>(dim);
      for(int j = 0; j < dim; ++j){
        Integer num = new Integer(aleat.nextInt(20));
        fila.set(j, num.floatValue());
      }
      matriz.set(i, fila);
    }
  }

  public void imprimirMatriz() {
    for(int i = 0; i < dim; ++i) {
      for(int j = 0; j < dim; ++j){
        System.out.print(" " + matriz.get(i).get(j) + " ");
      }
      System.out.println();
```

```
  }
}

public void multiplicarFilaPorEscalar(int f, float k) {
  if(k >=0 && k < dim) {
    AtomicReferenceArray<Float> fila = new AtomicReferenceArray<Float>(dim);
    for(int i = 0; i < dim; ++i) {
      fila.set(i, matriz.get(f).get(i));
    }
    for(int j = 0; j < dim; ++j) {
      fila.set(j, fila.get(j) * k);
    }
    matriz.set(f, fila);
  }
}

public void dividirMatriz(Matriz mat, float k) {
  for(int i = 0; i < dim; ++i) {
    AtomicReferenceArray<Float> fila = new AtomicReferenceArray<Float>(dim);
    for(int j = 0; j < dim; ++j) {
      fila.set(j, matriz.get(i).get(j) / k);
    }
    mat.inicializarFila(i, fila);
  }
}

public int dimension() {
  return dim;
}

public Matriz componente2(int f, int c) {
  Matriz mat = new Matriz(2);

  int indexI = 0;
  for(int i = 0; i < dim; ++i) {
    if(i != f) {
      AtomicReferenceArray<Float> fila = new AtomicReferenceArray<Float>(dim-1);
      int indexJ = 0;
      for(int j = 0; j < dim; ++j) {
        if(j != c) {
          fila.set(indexJ, matriz.get(i).get(j));
          ++indexJ;
        }
      }
      mat.inicializarFila(indexI, fila);
      ++indexI;
    }
  }

  return mat;
}
```

```
  public float determinante2() {
    float det = matriz.get(0).get(0) * matriz.get(1).get(1) -
      matriz.get(0).get(1) * matriz.get(1).get(0);
    return det;
  }

  public float determinante3() {
    float det = matriz.get(0).get(0) * matriz.get(1).get(1) * matriz.get(2).get(2) +
      matriz.get(1).get(0) *matriz.get(2).get(1) * matriz.get(0).get(2) +
      matriz.get(0).get(1) * matriz.get(1).get(2) * matriz.get(2).get(0) - (
      matriz.get(0).get(2) * matriz.get(1).get(1) * matriz.get(2).get(0) +
      matriz.get(1).get(2) * matriz.get(2).get(1) * matriz.get(0).get(0) +
      matriz.get(1).get(0) * matriz.get(0).get(1) * matriz.get(2).get(2));
    return det;
  }

  public void trasponer(Matriz tras) {
    for(int j = 0; j < dim; ++j) {
      AtomicReferenceArray<Float> fila = new AtomicReferenceArray<Float>(dim);
      for(int i = 0; i < dim; ++i) {
        fila.set(i, matriz.get(i).get(j));
      }
      tras.inicializarFila(j, fila);
    }
  }
}
```

Y este es el programa principal y las clases que representan a los hilos:

```
/**
 * Programa que sincroniza dos hilos entre sí usando semáforos.
 *
 * @author Natalia Partera
 * @version 1.0
 */

import java.util.concurrent.*;
import java.util.concurrent.atomic.*;

class HiloAdjunto extends Thread {
  final Matriz matriz;
  Semaphore semAdj;
  final Matriz adjuntos;

  public HiloAdjunto(Matriz m, Semaphore sem, Matriz adj) {
    matriz = m;
    semAdj = sem;
    adjuntos = adj;
  }

  public void cederPermiso(Semaphore semAdj) {
```

```
    semAdj.release();
  }

  public void run() {
    for(int i = 0; i < matriz.dimension(); ++i) {
      AtomicReferenceArray<Float> fila = new AtomicReferenceArray<Float>(
        matriz.dimension());
      for(int j = 0; j < matriz.dimension(); ++j) {
        fila.set(j, matriz.componente2(i, j).determinante2());
      }
      adjuntos.inicializarFila(i, fila);
    }
    cederPermiso(semAdj);
  }
}

class HiloTraspuesta extends Thread {
  Semaphore semAdj;
  Semaphore semDet;
  final Matriz adjuntos;
  final Matriz traspuesta;

  public HiloTraspuesta(Matriz adj, Matriz trasp, Semaphore semA, Semaphore semD) {
    adjuntos = adj;
    traspuesta = trasp;
    semAdj = semA;
    semDet = semD;
  }

  public void pedirPermiso(Semaphore semAdj) {
    try {
      semAdj.acquire();
    } catch(InterruptedException ex) {}
  }

  public void cederPermiso(Semaphore semDet) {
    semDet.release();
  }

  public void run() {
    pedirPermiso(semAdj);
    adjuntos.trasponer(traspuesta);
    cederPermiso(semDet);
  }
}

class HiloDeterminante extends Thread {
  final Matriz matriz;
  final Matriz traspuesta;
  final Matriz inversa;
  Semaphore semDet;
  float determinante;
```

```
  public HiloDeterminante(Matriz m, Matriz tras, Matriz inv, Semaphore sem) {
    matriz = m;
    traspuesta = tras;
    inversa = inv;
    semDet = sem;
  }

  public void pedirPermiso(Semaphore semDet) {
    try {
      semDet.acquire();
    } catch(InterruptedException ex) {}
  }

  public void run() {
    pedirPermiso(semDet);
    determinante = matriz.determinante3();
    traspuesta.dividirMatriz(inversa, determinante);
  }
}

public class MatrizInversa {
  public static void main (String[] args) throws InterruptedException {
    Semaphore semAdj = new Semaphore(0);
    Semaphore semDet = new Semaphore(0);
    Matriz matriz = new Matriz(3);
    Matriz adjuntos = new Matriz(3);
    Matriz traspuesta = new Matriz(3);
    float determinante;
    Matriz inversa = new Matriz(3);
    matriz.inicializarMatrizAleatoria();

    System.out.println("La matriz inicial es: ");
    matriz.imprimirMatriz();
    HiloAdjunto hAdj = new HiloAdjunto(matriz, semAdj, adjuntos);
    HiloTraspuesta hTras = new HiloTraspuesta(adjuntos, traspuesta, semAdj, semDet);
    HiloDeterminante hDet = new HiloDeterminante(matriz, traspuesta, inversa,
      semDet);
    hAdj.start();
    hTras.start();
    hDet.start();
    hDet.join();
    System.out.println("La matriz inversa es: ");
    inversa.imprimirMatriz();
  }
}
```

7.5. Ejercicio 5

Para la resolución de este ejercicio, le hemos añadido algunos métodos a la clase `Matriz`:

```
/**
 * Clase Matriz.
 *
 * @author Natalia Partera
 * @version 1.0
 */

import java.util.*;
import java.util.concurrent.atomic.*;

public class Matriz extends Thread {
  //Atributos privados
  private AtomicReferenceArray<AtomicReferenceArray<Float>> matriz;
  private int dim;

  //Constructor de la clase
  public Matriz(int dimension) {
    matriz = new AtomicReferenceArray<AtomicReferenceArray<Float>>(dimension);
    dim = dimension;

    for(int i = 0; i < dim; ++i) {
      AtomicReferenceArray<Float> fila = new AtomicReferenceArray<Float>(dim);
      matriz.set(i, fila);
    }
  }

  public void inicializarMatriz() {
    for(int i = 0; i < dim; ++i) {
      AtomicReferenceArray<Float> fila = new AtomicReferenceArray<Float>(dim);
      for(int j = 0; j < dim; ++j){
        Integer num = new Integer(i+j);
        fila.set(j, num.floatValue());
      }
      matriz.set(i, fila);
    }
  }

  public void copiarMatriz(Matriz m) {
    for(int i = 0; i < dim; ++i) {
      AtomicReferenceArray<Float> fila = new AtomicReferenceArray<Float>(dim);
      for(int j = 0; j < dim; ++j) {
        fila.set(j, m.obtener(i, j));
      }
      matriz.set(i, fila);
    }
  }

  public void inicializarMatrizDe1() {
    for(int i = 0; i < dim; ++i) {
      AtomicReferenceArray<Float> fila = new AtomicReferenceArray<Float>(dim);
      for(int j = 0; j < dim; ++j){
        Integer num = new Integer(1);
```

```
      fila.set(j, num.floatValue());
    }
    matriz.set(i, fila);
  }
}

public void inicializarFila(int f, AtomicReferenceArray<Float> fila) {
  matriz.set(f, fila);
}

public void inicializarMatrizAleatoria() {
  Random aleat = new Random();
  for(int i = 0; i < dim; ++i) {
    AtomicReferenceArray<Float> fila = new AtomicReferenceArray<Float>(dim);
    for(int j = 0; j < dim; ++j){
      Integer num = new Integer(aleat.nextInt(10));
      fila.set(j, num.floatValue());
    }
    matriz.set(i, fila);
  }
}

public void asignar(AtomicReferenceArray<Float> filaElem, int f) {
  matriz.set(f, filaElem);
}

public float obtener(int f, int c) {
  return matriz.get(f).get(c);
}

public void imprimirMatriz() {
  for(int i = 0; i < dim; ++i) {
    for(int j = 0; j < dim; ++j){
      System.out.print(" " + matriz.get(i).get(j) + " ");
    }
    System.out.println();
  }
}

public void multiplicarFilaPorEscalar(int f, float k) {
  if(k >=0 && k < dim) {
    AtomicReferenceArray<Float> fila = new AtomicReferenceArray<Float>(dim);
    for(int i = 0; i < dim; ++i) {
      fila.set(i, matriz.get(f).get(i));
    }
    for(int j = 0; j < dim; ++j) {
      fila.set(j, fila.get(j) * k);
    }
    matriz.set(f, fila);
  }
}
```

```
public AtomicReferenceArray<Float> multiplicaFilaPorColumna(Matriz original,
int fila) {
  AtomicReferenceArray<Float> elems = new AtomicReferenceArray<Float>(dim);
  float n = 0;
  for(int k = 0; k < dim; ++k) {
    n = 0;
    for(int i = 0; i < dim; ++i) {
      n = n + matriz.get(fila).get(i) * original.obtener(i, k);
    }
    elems.set(k, n);
  }
  return elems;
}

public void dividirMatriz(Matriz mat, float k) {
  for(int i = 0; i < dim; ++i) {
    AtomicReferenceArray<Float> fila = new AtomicReferenceArray<Float>(dim);
    for(int j = 0; j < dim; ++j) {
      fila.set(j, matriz.get(i).get(j) / k);
    }
    mat.inicializarFila(i, fila);
  }
}

public int dimension() {
  return dim;
}

public Matriz componente2(int f, int c) {
  Matriz mat = new Matriz(2);

  int indexI = 0;
  for(int i = 0; i < dim; ++i) {
    if(i != f) {
      AtomicReferenceArray<Float> fila = new AtomicReferenceArray<Float>(dim-1);
      int indexJ = 0;
      for(int j = 0; j < dim; ++j) {
        if(j != c) {
          fila.set(indexJ, matriz.get(i).get(j));
          ++indexJ;
        }
      }
      mat.inicializarFila(indexI, fila);
      ++indexI;
    }
  }

  return mat;
}

public float determinante2() {
  float det = matriz.get(0).get(0) * matriz.get(1).get(1) -
```

```
      matriz.get(0).get(1) * matriz.get(1).get(0);
    return det;
  }

  public float determinante3() {
    float det = matriz.get(0).get(0) * matriz.get(1).get(1) * matriz.get(2).get(2) +
      matriz.get(1).get(0) *matriz.get(2).get(1) * matriz.get(0).get(2) +
      matriz.get(0).get(1) * matriz.get(1).get(2) * matriz.get(2).get(0) - (
      matriz.get(0).get(2) * matriz.get(1).get(1) * matriz.get(2).get(0) +
      matriz.get(1).get(2) * matriz.get(2).get(1) * matriz.get(0).get(0) +
      matriz.get(1).get(0) * matriz.get(0).get(1) * matriz.get(2).get(2));
    return det;
  }

  public void trasponer(Matriz tras) {
    for(int j = 0; j < dim; ++j) {
      AtomicReferenceArray<Float> fila = new AtomicReferenceArray<Float>(dim);
      for(int i = 0; i < dim; ++i) {
        fila.set(i, matriz.get(i).get(j));
      }
      tras.inicializarFila(j, fila);
    }
  }
}
```

A continuación puede ver el programa principal:

```
/**
 * Programa que calcula la n-ésima potencia de una matriz.
 *
 * @author Natalia Partera
 * @version 1.0
 */

import java.util.concurrent.*;
import java.util.concurrent.atomic.*;

class Hilo extends Thread {
  final CyclicBarrier potencia;
  final Matriz matriz;
  final Matriz nEsima;
  int n, fila, columna;

  public Hilo(CyclicBarrier barrera, Matriz m, Matriz esima, int pot, int f) {
    potencia = barrera;
    matriz = m;
    nEsima = esima;
    n = pot;
    fila = f;
  }
```

```
  public void run() {
    if(n == 0)
    {
      nEsima.inicializarMatrizDe1();
    }
    else if (n > 1) {
      for(int i = 1; i < n; ++i) {
        try {
          potencia.await();
        } catch(BrokenBarrierException ex) {}
          catch(InterruptedException ex) {}
        AtomicReferenceArray<Float> filaElem = nEsima.multiplicaFilaPorColumna(
          matriz, fila);
        try {
          potencia.await();
        } catch(BrokenBarrierException ex) {}
          catch(InterruptedException ex) {}
        nEsima.asignar(filaElem, fila);
      }
    }
  }
}

public class MatrizNesima {
  public static void main (String[] args) throws InterruptedException {
    int dimension = 3;
    Matriz matriz = new Matriz(dimension);
    Matriz m_esima = new Matriz(dimension);
    int n = 3;
    int numHilos = dimension;
    matriz.inicializarMatrizAleatoria();
    m_esima.copiarMatriz(matriz);

    System.out.println("La matriz M es:");
    matriz.imprimirMatriz();

    CyclicBarrier barrera = new CyclicBarrier(numHilos);

    Hilo h1 = new Hilo(barrera, matriz, m_esima, n, 0);
    Hilo h2 = new Hilo(barrera, matriz, m_esima, n, 1);
    Hilo h3 = new Hilo(barrera, matriz, m_esima, n, 2);

    h1.start();
    h2.start();
    h3.start();
    h1.join();
    h2.join();
    h3.join();
    System.out.println("La matriz elevada a " + n + " es:");
    m_esima.imprimirMatriz();
  }
}
```

7.6. Ejercicio 6

La clase `Impresora` debe modificarse para incorporar el cerrojo, quedando de la siguiente manera:

```
/**
 * Clase Impresora.
 *
 * @author Natalia Partera
 * @version 1.0
 */

import java.util.concurrent.locks.*;

public class Impresora {
  //Atributos
  private final ReentrantLock cerrojo = new ReentrantLock();

  //Constructores
  public Impresora() {}

  //Métodos
  public void imprimir(String documento) {
    cerrojo.lock();
    try {
      System.out.println("Comienza la impresión de ’’" + documento + "’’.");
      for(int i = 0; i < 100000; ++i) {}
      System.out.println("Documento ’’" + documento + "’’ impreso.");
    } finally {
      cerrojo.unlock();
    }
  }
}
```

En el programa principal ahora usamos un cerrojo en lugar de un semáforo, y modificamos un poco la clase que se ejecuta como hilos:

```
/**
 * Programa que controla una cola de impresión usando cerrojos.
 *
 * @author Natalia Partera
 * @version 1.0
 */

import java.util.concurrent.locks.*;

class Tarea extends Thread {
  final Impresora impresora;
  String nombre;

  public Tarea(Impresora i, String nom) {
    impresora = i;
```

```
    nombre = nom;
  }

  public void run() {
    impresora.imprimir(nombre);
  }
}

public class UsaImpresora {
  public static void main (String[] args) {
    Impresora imp = new Impresora();
    int numTareas = 10;
    String[] tareas = new String[numTareas];

    for(int i = 0; i < numTareas; ++i) {
      tareas[i] = "Apuntes de PCTR: guión " + (i+1);
    }

    for(int j= 0; j < numTareas; ++j) {
      Tarea t = new Tarea(imp, tareas[j]);
      t.start();
    }
  }
}
```

7.7. Ejercicio 7

En este ejercicio, añadimos una cola (no concurrente ni sincronizada) de impresión a la clase `Impresora`. Sobre esta estructura se evaluarán las condiciones que permiten o no que se siga la ejecución de los hilos.

```
/**
 * Clase Impresora.
 *
 * @author Natalia Partera
 * @version 1.0
 */

import java.util.concurrent.locks.*;

public class Impresora {
  //Atributos
  private final ReentrantLock cerrojo = new ReentrantLock();
  private final Condition noVacia = cerrojo.newCondition();
  private final Condition noLlena = cerrojo.newCondition();

  private final String[] colaImpresion = new String[10];
  private int tareasEnCola = 0;

  //Constructores
  public Impresora() {}
```

```
  //Métodos
  public void mandarImprimir(String documento) throws InterruptedException {
    cerrojo.lock();
    try {
      while(tareasEnCola == (colaImpresion.length - 1))
        noLlena.await();
      colaImpresion[tareasEnCola] = documento;
      System.out.println("Se pone en cola un nuevo documento: " + documento);
      ++tareasEnCola;
      noVacia.signal();
    } finally {
      cerrojo.unlock();
    }
  }

  public void imprimir() throws InterruptedException {
    cerrojo.lock();
    try {
      while(tareasEnCola == 0)
        noVacia.await();
      String doc = colaImpresion[0];
      System.out.println("Comienza la impresión de ''" + doc + "''.");
      for(int i = 0; i < 100000; ++i) {}
      System.out.println("Documento ''" + doc + "'' impreso.");
      //Se elimina el documento impreso de la cola de impresión
      int j = 1;
      for(; colaImpresion[j] != null; ++j) {
        colaImpresion[j-1] = colaImpresion[j];
      }
      colaImpresion[j] = null;
      --tareasEnCola;
      noLlena.signal();
    } finally {
      cerrojo.unlock();
    }
  }
}
```

En el programa principal ampliamos la lista de documentos a imprimir. También creamos un hilo que funcionará como un hilo demonio, mandando imprimir documentos cuando exista alguno en la cola de impresión:

```
/**
 * Programa que controla una cola de impresión usando cerrojos y condiciones.
 *
 * @author Natalia Partera
 * @version 1.0
 */

import java.util.concurrent.locks.*;
```

```
class Tarea extends Thread {
  final Impresora impresora;
  String nombre;

  public Tarea(Impresora i, String nom) {
    impresora = i;
    nombre = nom;
  }

  public void run() {
    try {
      impresora.mandarImprimir(nombre);
    } catch (InterruptedException ex) {}
  }
}

class Imprimiendo extends Thread {
  final Impresora impresora;

  public Imprimiendo(Impresora i) {
    impresora = i;
  }

  public void run() {
    while(true) {
      try {
        impresora.imprimir();
      } catch (InterruptedException ex) {}
    }
  }
}

public class UsaImpresora {
  public static void main (String[] args) {
    Impresora imp = new Impresora();
    Imprimiendo daemonion = new Imprimiendo(imp);
    daemonion.start();

    int numTareas = 20;
    String[] tareas = new String[numTareas];

    int i = 0;
    for(; i < numTareas/2; ++i) {
      tareas[i] = "Apuntes de PCTR: guión " + (i+1);
    }
    for(; i < numTareas; ++i) {
      tareas[i] = "Apuntes de POO: guión " + (i+1-10);
    }

    for(int j= 0; j < numTareas; ++j) {
      Tarea t = new Tarea(imp, tareas[j]);
      t.start();
```

```
    }
  }
}
```

7.8. Ejercicio 8

En la clase `Impresora` hemos cambiado la cola de impresión por una cola del tipo `ArrayBlockingQueue`:

```
/**
 * Clase Impresora.
 *
 * @author Natalia Partera
 * @version 1.0
 */

import java.util.concurrent.*;
import java.util.concurrent.locks.*;

public class Impresora {
  //Atributos
  private final ReentrantLock cerrojo = new ReentrantLock();
  private final Condition noVacia = cerrojo.newCondition();
  private final Condition noLlena = cerrojo.newCondition();

  private final ArrayBlockingQueue<String> colaImpresion;
  private int tareasEnCola = 0;

  //Constructores
  public Impresora() {
    colaImpresion = new ArrayBlockingQueue<String>(10);
  }

  //Métodos
  public void mandarImprimir(String documento) throws InterruptedException {
    cerrojo.lock();
    try {
      while(colaImpresion.size() == 10)
        noLlena.await();
      colaImpresion.put(documento);
      System.out.println("Se pone en cola un nuevo documento: " + documento);
      ++tareasEnCola;
      noVacia.signal();
    } finally {
      cerrojo.unlock();
    }
  }

  public void imprimir() throws InterruptedException {
    cerrojo.lock();
    try {
      while(tareasEnCola == 0)
```

```
        noVacia.await();
      String doc = colaImpresion.take();
      System.out.println("Comienza la impresión de ’’" + doc + "’’.");
      for(int i = 0; i < 100000; ++i) {}
      System.out.println("Documento ’’" + doc + "’’ impreso.");
      --tareasEnCola;
      noLlena.signal();
    } finally {
      cerrojo.unlock();
    }
  }
}
```

El programa principal permanece sin cambios:

```
/**
 * Programa que controla una cola de impresión usando cerrojos, condiciones y colas.
 *
 * @author Natalia Partera
 * @version 1.0
 */

import java.util.concurrent.locks.*;

class Tarea extends Thread {
  final Impresora impresora;
  String nombre;

  public Tarea(Impresora i, String nom) {
    impresora = i;
    nombre = nom;
  }

  public void run() {
    try {
      impresora.mandarImprimir(nombre);
    } catch (InterruptedException ex) {}
  }
}

class Imprimiendo extends Thread {
  final Impresora impresora;

  public Imprimiendo(Impresora i) {
    impresora = i;
  }

  public void run() {
    while(true) {
      try {
        impresora.imprimir();
```

```
      } catch (InterruptedException ex) {}
    }
  }
}

public class UsaImpresora {
  public static void main (String[] args) {
    Impresora imp = new Impresora();
    Imprimiendo daemonion = new Imprimiendo(imp);
    daemonion.start();

    int numTareas = 20;
    String[] tareas = new String[numTareas];

    int i = 0;
    for(; i < numTareas/2; ++i) {
      tareas[i] = "Apuntes de PCTR: guión " + (i+1);
    }
    for(; i < numTareas; ++i) {
      tareas[i] = "Apuntes de POO: guión " + (i+1-10);
    }

    for(int j= 0; j < numTareas; ++j) {
      Tarea t = new Tarea(imp, tareas[j]);
      t.start();
    }

  }
}
```