

1.- ¿Qué son los semáforos?

Un semáforo es una variable especial (o tipo abstracto de datos) que constituye el método clásico para restringir o permitir el acceso a recursos compartidos (ejemplo , un recurso de almacenamiento del sistema o variables del código fuente) en un entorno de multiprocesamiento (en el que se ejecutan varios hilos de forma concurrente).

En definitiva un semáforo es un objeto que lleva la cuenta de un cierto número de permisos. Tenemos dos operaciones básicas:

- `acquire(n)` Bloquea al hilo que realiza la llamada hasta que hay tantos permisos disponibles como se solicita. En ese momento , se descuentan los permisos solicitados y el hilo se desbloquea.⁷
- `release(n)` Devuelve los permisos indicados. Como consecuencia de esta devolución , puede que algún hilo bloqueado en `acquire()` se desbloquee y prosiga su ejecución.

La verificación y modificación del valor del semáforo, así como la posibilidad de bloquearse se realiza en conjunto, como una sola e indivisible acción atómica. El sistema operativo garantiza que al iniciar una operación con un semáforo , ningún otro hilo puede tener acceso al semáforo hasta que la operación termine o se bloquee. Esta atomicidad es absolutamente necesaria para resolver los problemas de sincronización y evitar condiciones de competencia.

Si hay n recursos, el semáforo se inicializará al número n . Así, cada hilo, al ir solicitando un recurso , verificará que el valor del semáforo sea mayor de 0; si es así es que existen recursos libres, por lo que podrá tomar el recurso y decrementará el valor del semáforo.

Cuando el semáforo alcance el valor 0, significa que todos los recursos están siendo utilizados, y los hilos que quieran solicitar un recurso deberán esperar a que el semáforo sea positivo.

Los semáforos se emplean para permitir el acceso a diferentes partes de programas (llamados secciones críticas) donde se manipulan variables o recursos que deben ser accedidos de forma especial. Según el valor con que son inicializados se permiten a más o menos hilos utilizar el recurso de forma simultánea.

Un caso particular de semáforo es el binario , que puede tomar solamente los valores 0 y 1 . Se inicializan en 1 y son usados cuando un solo hilo puede acceder al recurso al mismo tiempo. Se les suele llamar **mutex**.

Los semáforos pueden ser usados para diferentes propósitos, entre ellos:

- Implementar cierres de exclusión mutua o locks
- Barreras
- Permitir a un máximo de N hilos acceder a un recurso, inicializando el semáforo en N .
- Notificación. Inicializando el semáforo en 0 puede usarse para comunicación entre hilos sobre la disponibilidad de un recurso.

Ejercicio 1 : Semaforo con 2 hilos

La aplicación estará compuesta por 3 clases :

Clase Hilo1

- Con un atributo de la clase Semaphore
- Contendrá dos métodos , el constructor `public Hilo1(Semaphore P1)` y el método `run()`.
- Método `run()` . Imprimirá cinco veces la palabra “Adios”.

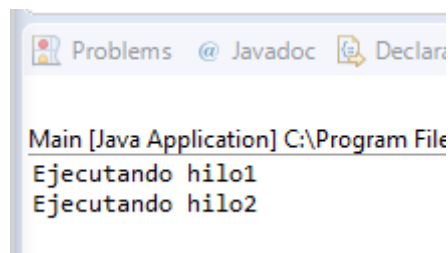
Clase Hilo2 . Igual que la clase Hilo1 con la salvedad de que imprime 5 veces la palabra “Hola”.

Clase Main. Declarará una instancia protected y static de la clase Semaphore para que pueda ser vista por las otras clases.

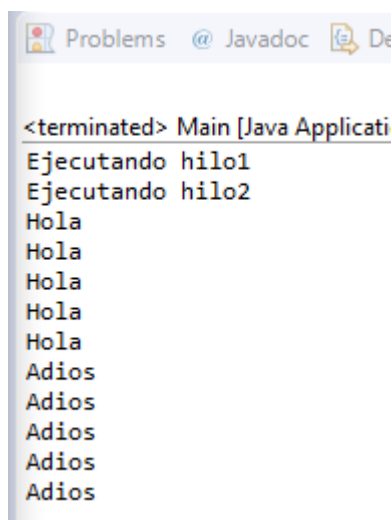
Crearé la instancia de la siguiente forma : `P1=new Semaphore(0, true)` y creará un hilo de la clase Hilo1 y lo lanzará , a continuación lanzará un hilo de la clase Hilo2 y lo lanzará.

Con esto, cuando hilo1 quiere tomar el recurso no puede porque el semáforo se ha inicializado con 0 recursos, por lo que el primero en tomar el recurso será el hilo 2, este se ejecuta porque no solicita previamente acceso al recurso que imprimirá cinco veces “Hola”.

¿Qué sucede si el hilo2 solicita también acceso al recurso con `this.P1.acquire()`; en este caso, ninguno de los hilos se ejecutará.



Cuando el hilo2 libera el recurso con la instrucción `this.P1.release(1)`; , como se ha inicializado el semáforo con true, tomará el recurso el hilo 1 que imprimirá cinco veces “Adios”



Comprueba el resultado cuando el semáforo se inicializa con “1”

SOLUCIÓN

EJERCICIO 2

Igual que el ejercicio 1 donde la clase Hilo1 imprime 5 veces “Hola” , la clase Hilo2 imprime “Adios” y la clase Hilo3 “Bye “. La clase Main es la siguiente:

```
import java.util.concurrent.Semaphore;

public class Main
{
    protected static Semaphore P1;

    public static void main(String[] args)
    {
        P1 = new Semaphore(0, true);

        Hilo1 Hil1 = new Hilo1(P1);
        Hil1.start();

        Hilo2 Hil2 = new Hilo2(P1);
        Hil2.start();

        Hilo3 Hil3 = new Hilo3(P1);
        Hil3.start();
    }
}
```

Nota: En este caso la clase Hilo1 no pedirá recurso por lo que entrará primero y liberará dos recursos.

EJERCICIO 3

Crear nuestro propio semáforo.

Aunque java proporciona la clase Semaphore, es bueno a nivel pedagógico realizar nuestro propio semáforo para comprender el funcionamiento.

Un semáforo es un objeto que lleva la cuenta de un cierto número de permisos . Los semáforos tienen dos operaciones básicas que debemos de implementar:

- acquire(n) . Bloquea al hilo llamante hasta que hay tantos permisos disponibles como se solicitan. En ese momento, se descuentan los permisos solicitados y el hilo se desbloquea.
- release(n). Devuelve los permisos indicados como parámetro. Como consecuencia de esta devolución , puede que algún hilo bloqueado por acquire() se desbloquee y continúe su ejecución.

```
public class MiSemaforo {
    private int permisos;

    public MiSemaforo(int permisos) {
        this.permisos = permisos;
    }
    public void acquire() throws InterruptedException {

    }
    public void acquire(int n) throws InterruptedException {

    }
}
```

```
        public void    release() {

        }

        public void release(int n) {
//Cuando se devuelve más de un permiso debemos hacer una llamada a todos los hilos bloqueados en cola.
        }
    }

    public class SemaforoTest {
        private static final int CNT= 100000; //Constante para realizar un bucle que incremente la variable
x
        private int x = 0; //Variable global compartida, que incrementará cada hilo

        public static void main(String[] args)throws InterruptedException {
//En el main se declara un objeto SemaforoTest y se llama al método iniciar()
        }

        private void iniciar()throws InterruptedException {
//Se crea tres objetos Hilo y se lanza
// Se debe de esperar a que termine cada hilo
            t1.join();
            t2.join();
            t3.join();

            if (x == 3 * CNT)
// Si la variable X ha llegado a 3000 indica que cada hilo ha realizado bien su tarea, por lo que se imprime
OK
                System.out.println("OK");
            else
//Si el valor de X no es 3000 el semáforo no ha realizado su función , se ha incrementado x de forma
anormal
                System.out.println("CONDICIÓN DE CARRERA!");
        }

//Creamos una clase interna para generar los hilos
        private class Hilo extends Thread {

            //Cada instancia de la clase Hilo se le pasa el atributo "semaforo"
            private final MiSemaforo semaforo;

            //Constructor
            public Hilo(MiSemaforo semaforo) {

            }

            //Método run() que debe de ejecutar cada hilo, consistirá en incrementar la variable x un total de
CNT veces, solicitando permiso y devolviendo el permiso.

            @Override
            public void run() {
                try {
                    for (int i = 0; i < CNT; i++) {
```

```
        //Incrementamos la variable x de modo atómico.  
        }  
    } catch (InterruptedException e) {  
        e.printStackTrace();  
    }  
}  
}  
}
```

NOTA: Si el semáforo funciona, los hilos terminan diciendo OK. Si el semáforo bloquea , el programa no termina y no dice nada. Si el semáforo no hace bien su función , los hilos terminan diciendo “CONDICIÓN DE CARRERA”, que quiere decir que se ha dado un caso de carrera o modificación descontrolada de una variable compartida.

SOLUCIÓN

EJERCICIO 4

Para este ejemplo, simularemos un sistema Productor_Consumidor y usaremos semáforos para controlar el flujo de datos.

El programa estará constituido por cuatro clases; Almacén, Consumidor, Productor y ProductorConsumidor.

Clase Consumidor. Se creará una instancia del almacén y en el método run() solo llamaremos al método consumir() de este.

```
public class Consumidor extends Thread {  
    private Almacen almacen;  
  
    public Consumidor(String name, Almacen almacen) {  
  
    }  
  
    public void run() {  
        while(true){  
  
        }  
    }  
}
```

Clase Productor: similar a la clase Consumidor con la salvedad de que aquí se llama al método producir()

```
public class Productor extends Thread {  
  
    private Almacen almacen;  
  
    public Productor(String name, Almacen almacen) {  
  
    }  
}
```

```
public void run() {  
    while (true) {  
  
    }  
}  
}
```

Clase Almacen: Definimos un limite máximo de productos a almacenar y declaramos los 3 semáforos, el productor, el consumidor y un mutex.

En el método producir(), daremos permisos al semáforo productor y al mutex, aumentaremos los productos y liberaremos los permisos del mutex y del semáforo consumidor (para que pueda empezar a consumir)

Métodos de la clase Semaphore:

- acquire() adquiere un permiso del semáforo, bloqueandolo hasta que haya un recurso disponible o se bloquee el hilo.
- acquire(int permits) Adquiere el número de permisos pasado por parámetro del semáforo, hasta que todos los permisos estén disponibles o se interrumpa el hilo.
- release() . Libera un permiso devolviéndolo al semáforo.
- release (int permits) . Libera el número de permisos pasado por parámetro, devolviéndolos al semáforo.
- Constructor **Semaphore(int permits, boolean fair)** . Crea un semáforo con el número dado de permisos y la configuración de equidad data.. El número de permisos puede ser negativo, en cuyo caso las liberaciones deben ocurrir antes de que se otorguen las adquisiciones.
Si fair= true , el semáforo garantizará la concesión de permisos en disputa de forma que el primero en entrar será el primero en salir, de lo contrario será false.
- **Semaphore(int permits)** Crea un semáforo con el número dado de permisos y una configuración de equidad injusta.

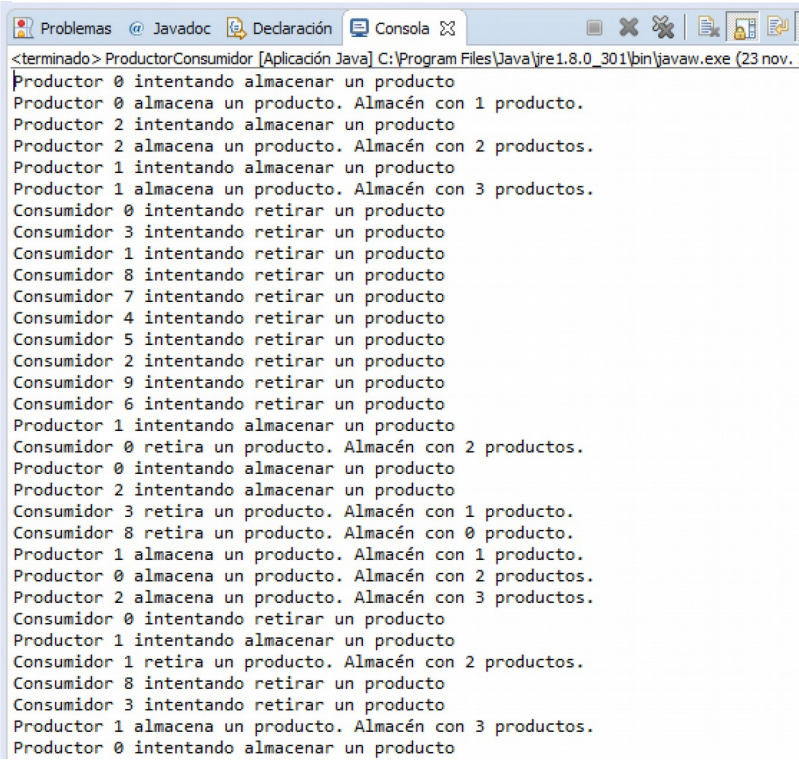
```
public class Almacen {  
  
    private final int MAX_LIMITE = 20;  
    private int producto = 0;  
    private Semaphore productor = new Semaphore(MAX_LIMITE);  
    private Semaphore consumidor = new Semaphore(0);  
    private Semaphore mutex = new Semaphore(1);  
  
    public void producir(String nombreProductor) {  
        System.out.println(nombreProductor + " intentando almacenar un producto");  
        try {  
  
            System.out.println(nombreProductor + " almacena un producto. "  
                + "Almacén con " + producto + (producto > 1 ? " productos." : " producto."));  
  
            Thread.sleep(500);  
  
        } catch (InterruptedException ex) {  
            Logger.getLogger(Almacen.class.getName()).log(Level.SEVERE, null, ex);  
        }  
    }  
}
```

```
    } finally {  
  
    }  
  
}  
  
public void consumir(String nombreConsumidor) {  
    System.out.println(nombreConsumidor + " intentando retirar un producto");  
    try {  
        System.out.println(nombreConsumidor + " retira un producto. "  
            + "Almacén con " + producto + (producto > 1 ? " productos." : " producto."));  
  
        Thread.sleep(500);  
    } catch (InterruptedException ex) {  
        Logger.getLogger(Almacen.class.getName()).log(Level.SEVERE, null, ex);  
    } finally {  
  
    }  
}
```

Clase ProductorConsumidor

En el main() principal iniciaremos con un bucle los productores y los consumidores

```
public static void main(String[] args) {  
    final int PRODUCTOR = 3;  
    final int CONSUMIDOR = 10;  
  
    Almacen almacen = new Almacen();  
  
    for (int i = 0; i < PRODUCTOR; i++) {  
        new Productor("Productor " + i, almacen).start();  
    }  
  
    for (int i = 0; i < CONSUMIDOR; i++) {  
        new Consumidor("Consumidor " + i, almacen).start();  
    }  
}
```



```
<terminado> ProductorConsumidor [Aplicación Java] C:\Program Files\Java\jre1.8.0_301\bin\javaw.exe (23 nov. .
Productor 0 intentando almacenar un producto
Productor 0 almacena un producto. Almacén con 1 producto.
Productor 2 intentando almacenar un producto
Productor 2 almacena un producto. Almacén con 2 productos.
Productor 1 intentando almacenar un producto
Productor 1 almacena un producto. Almacén con 3 productos.
Consumidor 0 intentando retirar un producto
Consumidor 3 intentando retirar un producto
Consumidor 1 intentando retirar un producto
Consumidor 8 intentando retirar un producto
Consumidor 7 intentando retirar un producto
Consumidor 4 intentando retirar un producto
Consumidor 5 intentando retirar un producto
Consumidor 2 intentando retirar un producto
Consumidor 9 intentando retirar un producto
Consumidor 6 intentando retirar un producto
Productor 1 intentando almacenar un producto
Consumidor 0 retira un producto. Almacén con 2 productos.
Productor 0 intentando almacenar un producto
Productor 2 intentando almacenar un producto
Consumidor 3 retira un producto. Almacén con 1 producto.
Consumidor 8 retira un producto. Almacén con 0 productos.
Productor 1 almacena un producto. Almacén con 1 producto.
Productor 0 almacena un producto. Almacén con 2 productos.
Productor 2 almacena un producto. Almacén con 3 productos.
Consumidor 0 intentando retirar un producto
Productor 1 intentando almacenar un producto
Consumidor 1 retira un producto. Almacén con 2 productos.
Consumidor 8 intentando retirar un producto
Consumidor 3 intentando retirar un producto
Productor 1 almacena un producto. Almacén con 3 productos.
Productor 0 intentando almacenar un producto
```

EJERCICIO 5 (VARIANTE DEL PRODUCTOR CONSUMIDOR MEDIANTE MÉTODOS SINCRONIZADOS: MONITOR)

En este problema volvemos a tener 2 hilos , P (productor) y C (consumidor). P produce datos a su ritmo. C los consume al suyo. C debe esperar a que P le facilite un dato antes de avanzar . U si C avanza despacio, P puede tener que frenar la producción de datos.

Entre P y C se establece un buffer o almacén intermedio que puede retener N datos ya producidos por P hasta que C los consuma.

El problema se generaliza para varios productores y varios consumidores que se coordinan a través de un único almacén o buffer.

Modifica el código de la clase Buffer para que se ejecute de forma sincronizada

```
public class Buffer<E> {
    private final E[] data;
    private int nDatos;

    public Buffer(int size) {

        data = (E[]) new Object[size];
        nDatos= 0;
    }

    public void put(E x) {
        data[nDatos++] = x;
    }
}
```



```
public E take() {

    E x = data[0];
    nDatos--;
    System.arraycopy(data, 1, data, 0, nDatos);
    data[nDatos] = null;
    return x;
}

}

*****

class Consumidor extends Thread {
    public static final int C_DELAY = 1000;
    private final Buffer<Integer> buffer;
    private int esperado = 0;

    public Consumidor(Buffer<Integer> buffer) {

        this.buffer = buffer;
    }

    @Override
    public void run() {

        while(true) {
            esperado++;
            nap(C_DELAY);
            int n = buffer.take();
            System.out.println("C: " + n);
            if(n != esperado)
                System.out.println("C: ERROR: esperado "+ esperado);
        }
    }

    private void nap(int ms) {

        try {
            Thread.sleep(ms);
        }
        catch (InterruptedException ignored) {
        }
    }

}

*****

class Productor extends Thread {
    public static final int P_DELAY = 1000;
    private final Buffer<Integer> buffer;
```

```
private static int n = 0;

public Productor(Buffer<Integer> buffer) {

    this.buffer = buffer;
}

@Override
public void run() {
    while(true) {
        n++;
        System.out.println("P: " + n);
        buffer.put(n);
        nap(P_DELAY);
    }
}

private void nap(int ms) {

    try {
        Thread.sleep(ms);
    }
    catch (InterruptedException ignored) {
    }
}
}
*****

public class Main {
    public static void main(String[] args) {

        Buffer<Integer> buffer = new Buffer<Integer>(3);

        Productor p = new Productor(buffer);
        Consumidor c = new Consumidor(buffer);

        p.start();
        c.start();
        // podria haber muchos productores
        // y muchos consumidores
    }
}
```

SOLUCIÓN

EJERCICIO 6

Ejemplo de semáforo binario.

Normalmente un mutex se utiliza generalmente para guardar cosas. El mutex puede proteger una parte del sistema, de modo que cuando vale 1 , el hilo puede acceder a ese subsistema y ningún otro hilo podrá tener acceso. En definitiva , un mutex es un semáforo de exclusión mutua.

Este ejercicio consiste en un contador compartido que será incrementado por cada hilo.

Tendremos 3 clases

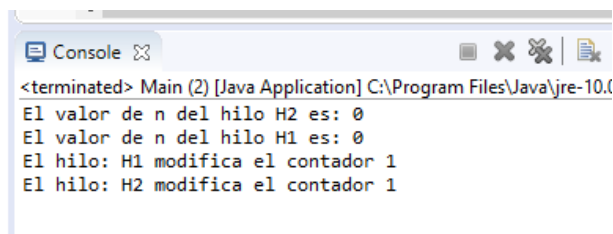
Clase Contador Compartido

- Contiene un atributo de tipo entero que será quien almacene los incrementos de cada hilo.
- Método getN(String id) se le pasa el identificador del hilo y devuelve el valor del atributo.(el contador correspondiente a ese hilo.
- Método setN (String id, int n) Se le pasa el identificador de del hilo y el valor del contador , asigna el valor de n pasado como parámetro al objeto ContadorComaprtido.

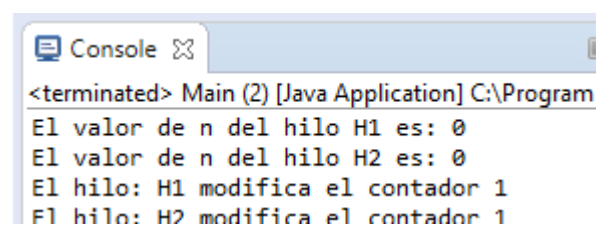
Clase IncrementadorLento

- Contiene tres atributos
 - String id: identificador del hilo que se ejecuta.
 - Objeto ContadorCompartido
 - Objeto Semaphore. Este objeto se inicializa con valor igual a 1.
- Constructor IncrementadorLento(String id, ContadorCompartido cc)
- método run() : Este método solicita recurso al semáforo , solicita el valor del contador del hilo, espera 1000 ms e incrementa el valor n del ContadorCompartido y por último debe de liberar el semáforo.
- Clase Main: contiene el método main que crea un objeto ContadorCompartido y 2 hilos de la clase IncrementadorLento . Posteriormente lanza estos dos hilos.

El resultado sin semáforo :



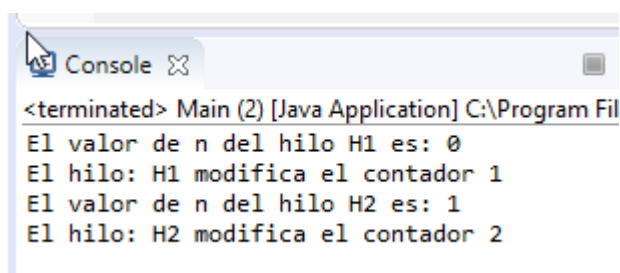
```
<terminated> Main (2) [Java Application] C:\Program Files\Java\jre-10.0.2\bin\java.exe
El valor de n del hilo H2 es: 0
El valor de n del hilo H1 es: 0
El hilo: H1 modifica el contador 1
El hilo: H2 modifica el contador 1
```



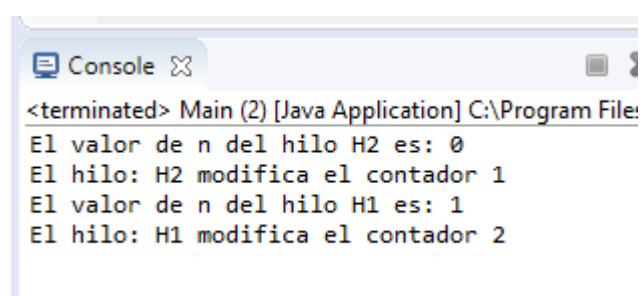
```
<terminated> Main (2) [Java Application] C:\Program Files\Java\jre-10.0.2\bin\java.exe
El valor de n del hilo H1 es: 0
El valor de n del hilo H2 es: 0
El hilo: H1 modifica el contador 1
El hilo: H2 modifica el contador 1
```

Si ejecutamos la aplicación varias veces, vemos que una veces entra primero el hilo 1 y otras el hilos 2

Con el mutex



```
<terminated> Main (2) [Java Application] C:\Program Files\Java\jre-10.0.2\bin\java.exe
El valor de n del hilo H1 es: 0
El hilo: H1 modifica el contador 1
El valor de n del hilo H2 es: 1
El hilo: H2 modifica el contador 2
```



```
<terminated> Main (2) [Java Application] C:\Program Files\Java\jre-10.0.2\bin\java.exe
El valor de n del hilo H2 es: 0
El hilo: H2 modifica el contador 1
El valor de n del hilo H1 es: 1
El hilo: H1 modifica el contador 2
```

SOLUCIÓN

EJERCICIO 7 (CONTADOR COMPARTIDO)

Tenemos una cuenta corriente donde una operación libera el hilo provocando situaciones conflictivas (condición de carrera)

Se crea un array de 10 clientes y para cliente mete y saca 1 de la cuenta corriente, por lo que el resultado final será 0.

NOTA: Ver en la api de java Class AtomicInteger

Actualiza una variable int de forma atómica.

Un AtomicInteger se usa en aplicaciones como contadores incrementados atómicamente y no se puede usar como reemplazo de un Integer. Sin embargo, esta clase amplía Number para permitir el acceso uniforme de herramientas y utilidades que tratan con clases basadas en números.

Realizar la modificación de la clase CuentaCorriente para obtener la solución mediante monitor y mediante una variable atómica.

```
public class CuentaCorriente {  
  
    private int    saldo = 0;  
  
    public void    mete(int n) {  
        saldo += n;  
    }  
  
    public void    saca(int n) {  
  
        int x = saldo;  
        nap      ();  
        x -= n;  
        saldo = x;  
    }  
  
    public int    getSaldo() {  
        return saldo;  
    }  
  
    private static void nap() {  
  
        try {  
            Thread.sleep((long) (100* Math.    random()));  
        } catch (InterruptedException ignored) {  
        }  
    }  
}
```

```
public class Cliente extends Thread {

    private final CuentaCorriente cc;

    public Cliente(CuentaCorriente cc) {

        this.cc = cc;
    }

    @Override
    public void run() {

        nap();
        cc.saca(1);
        nap();
        cc.mete(1);
    }

    private static void nap() {

        try {
            Thread.sleep((long) (100* Math.        random()));
        } catch (InterruptedException ignored) {
        }
    }
}
```

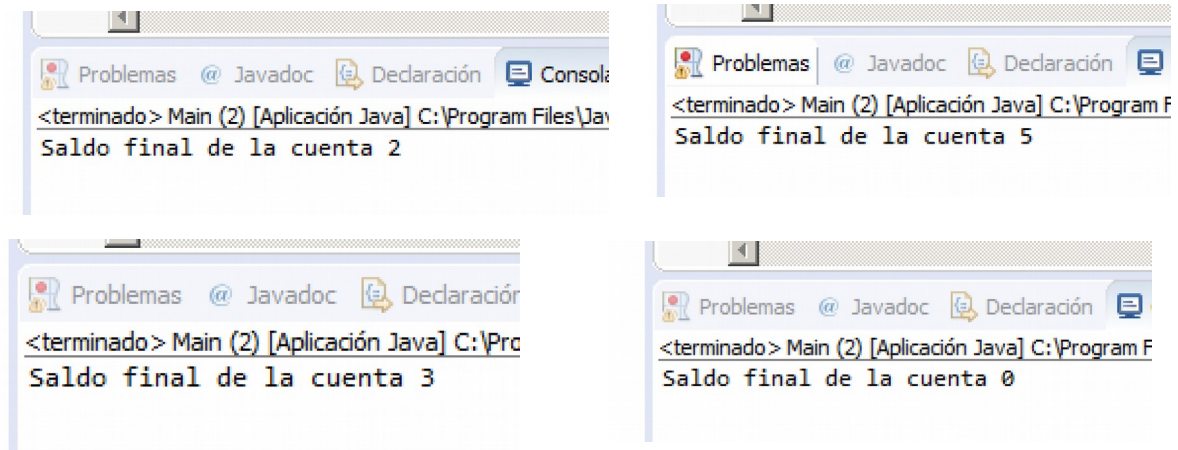
```
public class Main {

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

        CuentaCorriente        cc =new CuentaCorriente        ();

        int N = 10;
        Cliente[] clientes = new Cliente[N];

        for (int i = 0; i < clientes.length; i++)
            clientes[i] = new Cliente(cc);
        for (Cliente cliente : clientes)
            cliente.start();
        for (Cliente cliente : clientes)
            cliente.join();
        System.out.println("CC.getN(): "+ cc.getSaldo());
    }
}
```



SOLUCIÓN USANDO MONITOR

SOLUCIÓN USANDO UN OBJETO ATÓMICO DE JAVA

EJERCICIO 8: LECTORES Y ESCRITORES

Este sería el caso donde varios hilos compiten por un recurso común, pero de forma asimétrica. Concretamente, podemos pensar en un dato compartido que algunos quieren leer y otros escribir. Cada operación de lectura o escritura debe ser atómica; pero además queremos permitir varias lecturas simultáneas, pero cuando alguien escribe el acceso debe ser exclusivo.

Usar el siguiente objeto “Dato” para organizar la concurrencia.

```
public class Data {  
  
    public void openReading() {  
    }  
    public void closeReading() {  
    }  
    public void openWriting(ReaderWriter writer) {  
    }  
    public void closeWriting() {  
    }  
}
```

Colores :

- Gris Inactivo
- Azul lee
- Rojo escribe

Botones el primer número indica la fila y el segundo la columna.

| | | | |
|-----|-----|-----|-----|
| 0,0 | 0,1 | 0,2 | 0,3 |
| 1,0 | 1,1 | 1,2 | 1,3 |
| 2,0 | 2,1 | 2,2 | 2,3 |
| 3,0 | 3,1 | 3,2 | 3,3 |
| 4,0 | 4,1 | 4,2 | 4,3 |

| | | | |
|-----|-----|-----|-----|
| 0,0 | 0,1 | 0,2 | 0,3 |
| 1,0 | 1,1 | 1,2 | 1,3 |
| 2,0 | 2,1 | 2,2 | 2,3 |
| 3,0 | 3,1 | 3,2 | 3,3 |
| 4,0 | 4,1 | 4,2 | 4,3 |

| | | | |
|-----|-----|-----|-----|
| 0,0 | 0,1 | 0,2 | 0,3 |
| 1,0 | 1,1 | 1,2 | 1,3 |
| 2,0 | 2,1 | 2,2 | 2,3 |
| 3,0 | 3,1 | 3,2 | 3,3 |
| 4,0 | 4,1 | 4,2 | 4,3 |

```
package Lectura_Escritura;
```

```
import java.awt.Color;
```

```
import java.util.Random;
```

```
import javax.swing.JButton;
```

```
public class ReaderWriter extends Thread{
```

```
    private static final Color IDLE = Color.LIGHT_GRAY;
```

```
    private static final Color READING = Color.BLUE;
```

```
    private static final Color WRITING = Color.RED ;
```

```
    private final Random random = new Random();
```

```
    private final Data data;
```

```
    private final JButton button;
```

```
    public ReaderWriter(Data data, JButton button) {
```

```
        this.data = data;
```

```
        this.button = button;
```

```
    }
```

```
    @Override
```

```
    public void run() {
```

```
        while (true) {
```

```
            button.setBackground(IDLE);
```

```
            int s = 2;
```

```
            nap(s);
```

```
            if(Math.random() < 0.1 ) {
```

```
                data.openWriting(this);
```

```
                button.setBackground(WRITING);
```

```
                nap(5);
```

```
                data.closeWriting();
```

```
            } else {
```

```
                data.openReading();
```

```
                button.setBackground(READING);
```

```
                nap(3);
```

```
        data.closeReading();
    }
}

public void nap(int s){
    try {
        sleep(random.nextInt(s) * 3000);
    } catch (InterruptedException ignored) {
    }
}
}
```

```
package Lectura_Escritura;

import java.awt.Container;
import java.awt.GridLayout;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.WindowConstants;

public class Botones {

    private static final int N_ROWS = 5;
    private static final int N_COLS = 4;

    public static void main(String[] args) {

        Data data = new Data();

        Thread[][] threads = new Thread[N_ROWS][N_COLS];

        JFrame frame = new JFrame("Readers & writers");
        frame.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
        Container container = frame.getContentPane();
        container.setLayout(new GridLayout(N_ROWS, N_COLS));

        for (int row = 0; row < N_ROWS; row++) {
            for (int col = 0; col < N_COLS; col++) {
                JButton button = new JButton(String.format("%d, %d", row, col));
                button.setOpaque(true);
                container.add(button);
                ReaderWriter rw = new ReaderWriter(data, button);
                threads[row][col] = rw;
                rw.start();
            }
        }
        frame.pack();
        frame.setVisible(true);
    }
}
```



```
}
}
```

SOLUCIÓN CON VARIABLES DE ESTADO Y MONITOR

| | | | |
|------|------|------|------|
| 0, 0 | 0, 1 | 0, 2 | 0, 3 |
| 1, 0 | 1, 1 | 1, 2 | 1, 3 |
| 2, 0 | 2, 1 | 2, 2 | 2, 3 |
| 3, 0 | 3, 1 | 3, 2 | 3, 3 |
| 4, 0 | 4, 1 | 4, 2 | 4, 3 |

| | | | |
|------|------|------|------|
| 0, 0 | 0, 1 | 0, 2 | 0, 3 |
| 1, 0 | 1, 1 | 1, 2 | 1, 3 |
| 2, 0 | 2, 1 | 2, 2 | 2, 3 |
| 3, 0 | 3, 1 | 3, 2 | 3, 3 |
| 4, 0 | 4, 1 | 4, 2 | 4, 3 |

| | | | |
|------|------|------|------|
| 0, 0 | 0, 1 | 0, 2 | 0, 3 |
| 1, 0 | 1, 1 | 1, 2 | 1, 3 |
| 2, 0 | 2, 1 | 2, 2 | 2, 3 |
| 3, 0 | 3, 1 | 3, 2 | 3, 3 |
| 4, 0 | 4, 1 | 4, 2 | 4, 3 |

EN MODO ESCRITURA SOLO ACCEDE UNO

SOLUCIÓN 2 ORDENA LOS ESCRITORES

Un efecto curioso de la solución anterior es que los escritores se agolpan a la entrada y son atendidos en cualquier orden. Por lo que se puede penalizar a alguno de los escritores. En esta solución se respeta el orden de llegada mediante una FIFO.