Concurrencia
Ejercicios resueltos

josé a. mañas

16.4.2015

Contenido

1 Introducción

Ejercicios clásicos y no tan clásicos de concurrencia resueltos en java usando monitores.

Los ejercicios están ordenados de forma un poco arbitraria. Quizás empezando por los más fáciles y se van complicando poco a poco. Pero a veces un planteamiento es engañosamente sencillo y otras veces la solución es muy fácil.

2 Ejercicios

2.1 Semáforo

Java proporciona Semaphores. Poner estos objetos como ejercicio sólo tiene interés académico.

Un semáforo es un objeto que lleva la cuenta de un cierto número de permisos. Hay dos operaciones básicas

acquire(n)

> Bloquea a la thread llamante hasta que hay tantos permisos disponibles como se solicitan. En ese momento, se descuentan los permisos solicitados y la thread se desbloquea.

release(n)

> Devuelve los permisos indicados. Como consecuencia de esta devolución, puede que alguna otra thread bloqueada en un acquire() pueda proseguir.

La clase Sepahore de java tiene más métodos; pero vamos a implementar sólo esos dos.

```
public class MySemaphore {
    private int permits;

    public MySemaphore(int permits) {
        this.permits = permits;
    }
```

```
    public void acquire()
            throws InterruptedException {
        acquire(1);
    }

    public void acquire(int n)
            throws InterruptedException {
    }
```

```
    public void release() {
        release(1);
    }

    public void release(int n) {
    }
}
```

Escenario de prueba

```
public class SempahoreTest {
    private static final int CNT = 100000;

    private int x = 0;
```

```
    public static void main(String[] args)
            throws InterruptedException {
        SempahoreTest test = new SempahoreTest();
        test.go();
    }
```

```
    private void go()
            throws InterruptedException {
        MySemaphore s = new MySemaphore();

        Thread t1 = new Agent(s);
        Thread t2 = new Agent(s);
        Thread t3 = new Agent(s);

        t1.start();
        t2.start();
        t3.start();

        t1.join();
        t2.join();
        t3.join();

        if (x == 3 * CNT)
            System.out.println("OK");
        else
            System.out.println("Race error!");
    }
```

```
    private class Agent extends Thread {
        private final MySemaphore semaphore;

        public Agent(MySemaphore semaphore) {
            this.semaphore = semaphore;
        }

        @Override
        public void run() {
            try {
                for (int i = 0; i < CNT; i++) {
                    semaphore.acquire();
                    x = x + 1;
                    semaphore.release();
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Si el semáforo funciona, las threads terminan diciendo OK.

Si el semáforo bloquea, el programa no termina y no dice nada.

Si el semáforo no hace bien su función, las threads terminan diciendo “race condition!” que quiere decir que se ha dado un caso de carrera o modificación descontrolada de una variable compartida: x.

2.2 Pestillo con cuenta atrás (count down latch)

Ver java.util.concurrent.CountDownLatch.

Se trata de un objeto que se inicializa con un contador N que podemos ir decrementando uno a uno. Podemos poner tareas a esperar a que el contador llegue a cero, liberándose todas las tareas que esperan.

```
public class MyCountDownLatch {
    private int n;

    public MyCountDownLatch(int n) {
        this.n = n;
    }
```

```
    public void countDown() {
    }
```

```
    public void await()
            throws InterruptedException {
    }
}
```

Ejemplo de uso (tomado de la documentación de java para CountDownLatch).

```
public class Driver {
    private static final int N = 5;

    public static void main(String[] args)
      throws InterruptedException {
        MyCountDownLatch startSignal =
          new MyCountDownLatch(1);
        MyCountDownLatch doneSignal =
          new MyCountDownLatch(N);

        for (int i = 0; i < N; ++i) {
            Worker worker =
              new Worker(i, startSignal, doneSignal);
            Thread thread = new Thread(worker);
            thread.start();
        }

        System.out.println("before");
        startSignal.countDown();
        for (int i = 0; i < N; i++)
            System.out.print('.');
        doneSignal.await();
        System.out.println();
        System.out.println("after");
    }
}
```

```
public class Worker implements Runnable {
    private final int id;
    private final MyCountDownLatch startSignal;
    private final MyCountDownLatch doneSignal;

    Worker(int id,
           MyCountDownLatch startSignal,
           MyCountDownLatch doneSignal) {
        this.id = id;
        this.startSignal = startSignal;
        this.doneSignal = doneSignal;
    }

    public void run() {
        try {
            startSignal.await();
            doWork();
            doneSignal.countDown();
        } catch (InterruptedException ex) {
        }
    }

    void doWork() {
        for (int i = 0; i < id; i++)
            System.out.print(id);
    }
}
```

2.3 Contador compartido

Se presenta como una cuenta corriente donde una operación libera el thread provocando situaciones conflictivas (carreras o races).

```
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) {
        }
    }
}
```

Escenario de prueba:

```
    public static void main(String[] args)
        throws InterruptedException {
        CuentaCorriente cc = new CuentaCorriente();

        int N = 10;
        Robot[] robots = new Robot[N];
        for (int i = 0; i < robots.length; i++)
            robots[i] = new Robot(cc);
        for (Robot robot : robots)
            robot.start();
        for (Robot robot : robots)
            robot.join();
        System.out.println("CC.getN(): " + cc.getSaldo());
    }
```

```
    class Robot extends Thread {
        private final CuentaCorriente cc;

        public Robot(CuentaCorriente cc) {
            this.cc = cc;
        }

        @Override
        public void run() {
            nap();
            cc.saca(1);
            nap();
            cc.mete(1);
        }
    }
```

2.4 Productor – Consumidor

Es un problema clásico en el que 2 threads P y C deben sincronizarse. P produce datos a su ritmo. C los consume al suyo. C debe esperar a que P le facilite un dato antes de avanzar. Y 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.

```
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;
    }
}
```

Escenario de uso

```
public static final int P_DELAY = 3000;
public static final int C_DELAY = 1000;
```

```
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
}
```

```
private void nap(int ms) {
    try {
        Thread.sleep(ms);
    } catch (InterruptedException ignored) {
    }
}
```

```
class Productor
        extends Thread {
    private final Buffer<Integer> buffer;
    private 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);
        }
    }
}
```

```
class Consumidor extends Thread {
    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);
        }
    }
}
```

2.5 Readers & Writers

Escenario clásico donde varios agentes 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: sólo 1 lector y ningún escritor.

Usaremos este objeto dato para organizar la concurrencia

```
public class Data {

    public void openReading() {
    }

    public void closeReading() {
    }

    public void openWriting(ReaderWriter writer) {
    }

    public void closeWriting() {
    }
}
```

Escenario de pruebas.

```
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) * 1000);
        } catch (InterruptedException ignored) {
        }
    }
}
```

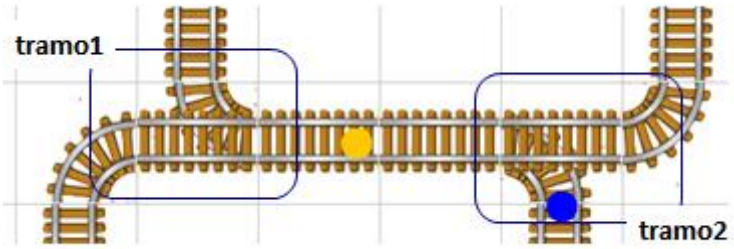
```
public class Botones {
    private static final int N_ROWS = 5;
    private static final int N_COLS = 4;

    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);
    }
}
```

2.6 Trenes

Estos ejemplos se pueden animar usando la simulación de trenes en

http://www.dit.upm.es/~pepe/doc/adsw/trenes/

En abstracto, tenemos una zona de vía compartida por varios trenes y queremos controlar la entrada de ternes por uno y otro lado



```
public class MonitorTunel
        extends Monitor {
    private final Tramo tramo1;
    private final Tramo tramo2;

    public MonitorTunel(Tramo tramo1, Tramo tramo2) {
        this.tramo1 = tramo1;
        this.tramo2 = tramo2;
    }

    // la thread se detiene hasta que pueda entrar
    public void entro(Tren tren, Tramo tramo, Enlace entrada) {
    }

    // la thread sale
    public void salgo(Tren tren, Tramo tramo, Enlace salida) {
    }

}
```

Se proponen 4 ejercicios de control del tramo compartido:

Ejercicio 1

El monitor debe controlar que en el tramo de vía compartido nunca hay más de 1 tren en cada momento. Es decir, o está vacío o está ocupado con 1 tren.

Cuando un tren quiere entrar y hay otro tren dentro, queda esperando hasta que la vía esté libre.

Cuando un tren sale, se lo notifica a los trenes que estén esperando.

Ejercicio 2

Al igual que el ejercicio 1, no puede haber más de un tren dentro. Pero además, cuando un tren sale, tiene preferencia para entrar un tren que esté esperando en dirección opuesta.

Se trata de hacer que el recurso se comparta de forma equitativa (fair).

Ejercicio 3

Se permite que haya varios trenes circulando en la misma dirección por el tramo compartido.

Se trata de optimizar el uso de un recurso compartido.

Ejercicio 4

Al igual que el ejercicio 3, puede haber varios trenes circulando en el mismo sentido. Pero además, cuando un tren sale, tiene preferencia para entrar un tren que esté esperando en dirección opuesta. Es decir, que un nuevo tren sólo entra si no hay nadie esperando en sentido contrario y si no hay nadie circulando en sentido contrario.

Se trata de hacer que el recurso se comparta de forma equitativa (fair).

2.7 Barrera

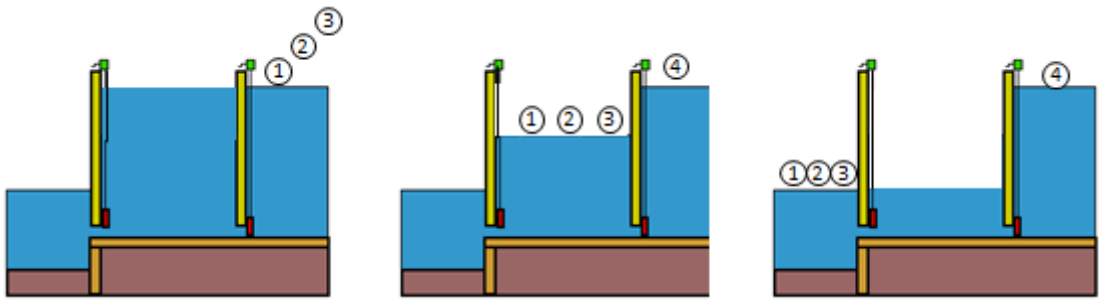
Una barrera de N posiciones retiene las primeras N-1 threads que llegan. Cuando llega la enésima, permite que salgan todas. La barrera queda derruida y nuevas threads pasan sin esperar.

```
public class Barrier {
    private final int n;

    public Barrier(int n) {
        this.n = n;
    }

    public void await() {
    }
}
```

2.8 Barrera Cíclica

Ver CyclicBarrier.

Una barrera cíclica de N posiciones retiene las primeras N-1 threads que llegan. Cuando llega la enésima, permite que salgan todas. Y empieza a contar para hacer otro grupo de N threads.



```java
public class MyCyclicBarrier {
    private final int n;

    public MyCyclicBarrier(int n) {
        this.n = n;
    }

    public synchronized void await() {
        TO DO
    }
```

Escenario de prueba

```java
    public static void main(String[] args) {
        MyCyclicBarrier barrier = new MyCyclicBarrier(3);

        for (int i= 1; i < 100; i++) {
            Barco barco = new Barco(i, barrier);
            barco.start();
            nap(2);
        }
    }
```

```java
    private static void nap(int s) {
        try {
            Thread.sleep((long) (Math.random()*s*1000));
        } catch (InterruptedException ignored) {
        }
    }
```

```java
    private static class Barco extends Thread {
        private final int id;
        private final MyCyclicBarrier barrier;

        public Barco(int id, MyCyclicBarrier barrier) {
            this.id = id;
            this.barrier = barrier;
            setName(String.valueOf(id));
        }

        @Override
        public void run() {
            nap(3);
            System.out.println(id + ": llega: "
                               + new Date());
            barrier.await();
            System.out.println(id + ": sale:  "
                               + new Date());
        }
    }
```

2.9 Intercambiador (Exchanger<V>)

Dos agentes A y B se sincronizan en 1 punto. A le pasa un dato a B; B le pasa un dato a A.

```
public class Exchanger<V> {

    public synchronized V exchange(V x) {
      TO DO
    }
}
```

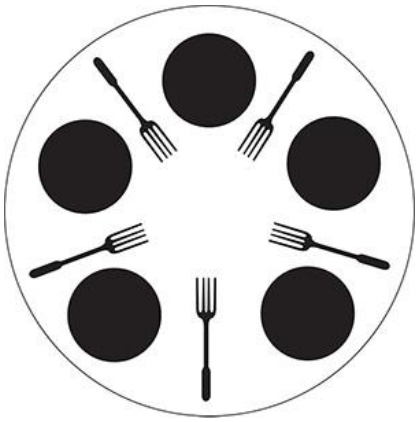
Escenario de prueba

```
public static void main(String[] args) {
    int N = 5;
    Exchanger<Integer> exchanger =
        new Exchanger<Integer>();

    for (int id = 0; id < N; id++) {
        Agente agente = new Agente(id + 1, exchanger);
        agente.start();
    }
}
```

```
class Agente extends Thread {
    private final int id;
    private int n;
    private final Exchanger<Integer> exchanger;

    public Agente(int id, Exchanger<Integer> exchanger) {
        this.id = id;
        this.n = 1000 * id;
        this.exchanger = exchanger;
    }

    @Override
    public void run() {
        while (true) {
            nap();
            int m = exchanger.exchange(this.n);
            System.out.printf(
               "agente %d: recibe %d%n", id, m);
            n++;
        }
    }
}
```

2.10 La cena de los filósofos (Dining philosophers)

Es un problema clásico que se suele plantear en cursos de sistemas operativos, pero que se aplica a situaciones donde unos pocos recursos se comparten entre unas pocas tareas y los recursos se asignan poco a poco llevando a situaciones de bloqueo (deadlock).



Se presenta como N filósofos (típicamente, N = 5) que comparten N tenedores. Cada filósofo normalmente está pensando y cuando le entra hambre coge su tenedor izquierdo, luego el derecho, como un poco, devuelve los tenedores y sigue pensando.

El problema es que si todos los filósofos cogen su tenedor izquierdo simultáneamente, todos se quedan bloqueados esperando por su tenedor derecho que está ocupado.

Cada tenedor (recurso compartido) se modela fácilmente como una clase:

```
public class Fork {
    private boolean inuse;

    public void take() {
        inuse = true;
    }

    public void leave() {
        inuse = false;
    }

    public boolean isInuse() {
        return inuse;
    }
}
```

Cada filósofo se puede modelar como una thread

```
public class Philosopher extends Thread {
    private enum Mode {
        THINKING, HUNGRY, EATING
    }

    private final Table table;
    private final int id;
    private Fork left;
    private Fork right;
    private Mode mode;
```

```
    public Philosopher(int id, Table table,
                       Fork left, Fork right) {
        this.table = table;
        this.id = id;
        this.left = left;
        this.right = right;
        this.mode = Mode.THINKING;
    }
```

```
    @Override
    public void run() {
        while (true) {
            this.mode = Mode.THINKING;
            nap(1);
            this.mode = Mode.HUNGRY;
            nap(1);
            table.take(left, right);
            this.mode = Mode.EATING;
            nap(4);
            table.leave(left, right);
        }
    }
```

```
    public boolean isThinking() {
        return mode == Mode.THINKING;
    }

    public boolean isHungry() {
        return mode == Mode.HUNGRY;
    }

    public boolean isEating() {
        return mode == Mode.EATING;
    }
```

```
    private void nap(int s) {
        try {
            sleep((long) (s * 1000));
        } catch (InterruptedException ignore) {
        }
    }
}
```

Nótese que los filósofos disponen de un objeto compartido Table, que es lo que se propone como ejercicio de programación de un monitor. Este monitor debe implementar las operaciones atómicas de coger ambos tenedores o dejarlos, sin posibilidad de coger o dejar sólo uno. Es decir, cada thread sólo progresa cuando tiene ambos tenedores asidos.

```
public class Table {

    public void take(Fork left, Fork right) {

    }

    public void leave(Fork left, Fork right) {
    }
}
```

Y necesitamos montar el escenario donde los N filósofos comparten mesa y tenedores:

```
public class DiningPhilosophers {
    private static final int N = 5;
    private static Philosopher[] philosophers =
        new Philosopher[N];

    public static void main(String[] args) {
        Table table = new Table();

        Fork[] forks = new Fork[N];
        for (int i = 0; i < N; i++)
            forks[i] = new Fork();

        for (int id = 0; id < N; id++) {
            Fork left = forks[id];
            Fork right = forks[(id + 1) % N];
            philosophers[id] =
                new Philosopher(id, table, left, right);
        }
        for (Philosopher philosopher : philosophers)
            philosopher.start();
    }
}
```

3 Soluciones

3.1 Semáforo

```
public synchronized void acquire()
        throws InterruptedException {
    acquire(1);
}
```

```
public synchronized void acquire(int n)
        throws InterruptedException {
    while (permits < n)
        wait();
    permits -= n;
}
```

```
public synchronized void release() {
    release(1);
}
```

```
public synchronized void release(int n) {
    permits += n;
    notifyAll();
}
```

3.2 Pestillo con cuenta atrás (count down latch)

Variables de estado: no son necesarias.

```
private int n;

public MyCountDownLatch(int n) {
    this.n = n;
}
```

```
public synchronized void countDown() {
    n--;
    notifyAll();
}
```

```
public synchronized void await()
        throws InterruptedException {
    while (n > 0)
        wait();
}
```

3.3 Contador compartido

3.3.1 Solución usando un monitor

```
public class CuentaCorriente {
    private int saldo = 0;

    public synchronized void mete(int n) {
        saldo += n;
    }

    public synchronized void saca(int n) {
        int x = saldo;
        nap();
        x -= n;
        saldo = x;
    }

    public synchronized int getSaldo() {
        return saldo;
    }

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

3.3.2 Solución usando un objeto atómico de java

```
public class CuentaCorriente {
    private AtomicInteger saldo = new AtomicInteger(0);

    public void mete(int n) {
        saldo.addAndGet(n);
    }

    public void saca(int n) {
        saldo.addAndGet(-n);
    }

    public int getSaldo() {
        return saldo.get();
    }
}
```

3.4 Productor – Consumidor

3.4.1 Solución programada

No necesitamos más variables de estado.

```
public Buffer(int size) {
    data = (E[]) new Object[size];
    nDatos = 0;
}
```

```
public synchronized void put(E x) {
    while (nDatos >= data.length)
        waiting();
    data[nDatos++] = x;
    notifyAll();
}
```

```
public synchronized E take() {
    while (nDatos <= 0)
        waiting();
    E x = data[0];
    nDatos--;
    System.arraycopy(data, 1, data, 0, nDatos);
    data[nDatos] = null;
    notifyAll();
    return x;
}
```

```
private void waiting() {
    try {
        wait();
    } catch (InterruptedException ignored) {
    }
}
```

3.4.2 Solución usando la biblioteca de java

La arquitectura productor-consumidor es tan frecuente que está prevista una clase con esa funcionalidad en la biblioteca de java: BlockingQueue. Java proporciona varias implementaciones. Vemos una:

```
public static final int P_DELAY = 3000;
public static final int C_DELAY = 3000;
```

```
public static void main(String[] args) {
    BlockingQueue<Integer> buffer =
        new ArrayBlockingQueue<Integer>(3);
    Productor p = new Productor(buffer);
    Consumidor c = new Consumidor(buffer);
    p.start();
    c.start();
}
```

```
private void nap(int ms) {
    try {
        Thread.sleep(ms);
    } catch (InterruptedException e) {
    }
}
```

```
class Productor
        extends Thread {
    private final BlockingQueue<Integer> buffer;
    private int n = 0;

    public Productor(BlockingQueue<Integer> buffer) {
        this.buffer = buffer;
    }

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

```
class Consumidor extends Thread {
    private final BlockingQueue<Integer> buffer;
    private int esperado = 0;

    public Consumidor(BlockingQueue<Integer> buffer) {
        this.buffer = buffer;
    }

    @Override
    public void run() {
        try {
            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);
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

3.5 Readers & Writers

3.5.1 Solución 1

Variables de estado:

int nWriters = 0

Lleva cuenta de cuántos agentes están leyendo ahora mismo.

int nReaders = 0

Lleva cuenta de cuántos escritores están escribiendo ahora mismo. Esta variable sólo puede valer 0 o 1, y podría sustituirse por un boolean.

int nWritersWaiting = 0

Lleva cuenta de cuántos lectores están esperando para leer.

```
private int nWriters = 0;
private int nReaders = 0;
private int nWritersWaiting = 0;
```

```
public synchronized void openReading() {
   while (nWriters > 0 || nWritersWaiting > 0)
            waiting();
       nReaders++;
}
```

```
public synchronized void closeReading() {
    nReaders--;
    notifyAll();
}
```

```
public synchronized void openWriting(ReaderWriter writer) {
        nWritersWaiting++;
    while (nReaders > 0 || nWriters > 0)
            waiting();
        nWritersWaiting--;
    System.out.println("en cola: " + nWritersWaiting);
    nWriters++;
}
```

```
    public synchronized void closeWriting() {
        nWriters--;
        notifyAll();
    }
```

```
    private void waiting() {
        try {
            wait();
        } catch (InterruptedException ignored) {
        }
    }
```

3.5.2 Solución 2: ordena los escritores

Un efecto curioso de la primera solución es que los escritores se agolpan a la entrada y son atendidos en cualquier orden. Esta segunda solución respeta el orden de llegada (FIFO – First In, First out)

Variables de estado:

int nWriters = 0

Lleva cuenta de cuántos agentes están leyendo ahora mismo.

int nReaders = 0

Lleva cuenta de cuántos escritores están escribiendo ahora mismo. Esta variable sólo puede valer 0 o 1, y podría sustituirse por un boolean.

List<ReaderWriter> writersWaiting

Lista de los agentes que están esperando a escribir. Se añaden al final según llegan y se sacan al principio según se les atiende. El efecto es una cola FIFO.

```
private int nWriters = 0;
private int nReaders = 0;
private List<ReaderWriter> writersWaiting =
    new ArrayList<ReaderWriter>();
```

```
public synchronized void openReading() {
    while (nWriters > 0 || writersWaiting.size() > 0)
        waiting();
    nReaders++;
}
```

```
public synchronized void closeReading() {
    nReaders--;
    notifyAll();
}
```

```
public synchronized void openWriting(ReaderWriter writer) {
    writersWaiting.add(writer);
    while (nReaders > 0
        || nWriters > 0
        || !writersWaiting.get(0).equals(writer))
        waiting();
    writersWaiting.remove(0);
    nWriters++;
}
```

```
public synchronized void closeWriting() {
    nWriters--;
    notifyAll();
}
```

```
private void waiting() {
    try {
        wait();
    } catch (InterruptedException e) {
    }
}
```

3.6 Trenes

3.6.1 Ejercicio 1

Sólo se admite 1 tren dentro en cada momento. Los demás, esperan.

Variables de estado:

boolean ocupado = false

TRUE si hay un tren dentro

```
public class MonitorTunel_1
        extends Monitor {
    private final Tramo tramo1;
    private final Tramo tramo2;

    private boolean ocupado = false;

    public MonitorTunel_1(Tramo tramo1, Tramo tramo2) {
        this.tramo1 = tramo1;
        this.tramo2 = tramo2;
    }
```

```
    public synchronized void entro(
        Tren tren, Tramo tramo, Enlace entrada) {
        while (ocupado)
            waiting();
        ocupado = true;
    }
```

```
    public synchronized void salgo(
        Tren tren, Tramo tramo, Enlace salida) {
        ocupado = false;
        notifyAll();
    }
```

```
    private void waiting() {
        try {
            wait();
        } catch (InterruptedException ignored) {
        }
    }
}
```

3.6.2 Ejercicio 2

Sólo puede haber un tren dentro; pero se hace un reparto equitativo de quién puede entrar alternando la entrada por uno u otro tramo.

Variables de estado:

int esperando1 = 0;

Lleva la cuenta de cuántos trenes hay esperando entrar por el tramo 1.

int esperando2 = 0;

Lleva la cuenta de cuántos trenes hay esperando entrar por el tramo 2.

boolean ocupado = false;

TRUE si hay un tren dentro.

int ultimaEntrada = 1;

Memoriza si la última entrada fue por el tramo 1 o por el tramo 2.

```
public class MonitorTunel_2
        extends Monitor {
    private final Tramo tramo1;
    private final Tramo tramo2;

    private int esperando1 = 0;
    private int esperando2 = 0;
    private boolean ocupado = false;
    private int ultimaEntrada = 1;

    public MonitorTunel_2(Tramo tramo1, Tramo tramo2) {
        this.tramo1 = tramo1;
        this.tramo2 = tramo2;
    }
```

```
    public synchronized void entro(
        Tren tren, Tramo tramo, Enlace entrada) {
        if (tramo.equals(tramo1)) {
            esperando1++;
            while (ocupado
                || ultimaEntrada == 1 && esperando2 > 0)
                waiting();
            esperando1--;
            ocupado = true;
            ultimaEntrada = 1;
        }
        if (tramo.equals(tramo2)) {
            esperando2++;
            while (ocupado
                || ultimaEntrada == 2 && esperando1 > 0)
                waiting();
            esperando2--;
            ocupado = true;
            ultimaEntrada = 2;
        }
    }
```

```
    public synchronized void salgo(
        Tren tren, Tramo tramo, Enlace salida) {
        ocupado = false;
        notifyAll();
    }
```

```
    private void waiting() {
        try {
            wait();
        } catch (InterruptedException ignored) {
        }
    }
}
```

3.6.3 Ejercicio 3

Se permiten varios trenes dentro, si circulan en la misma dirección.

Variables de estado:

int ocupado12 = 0

Lleva cuenta de cuántos trenes hay circulando que entraron por el tramo 1 en dirección al tramo 2.

int ocupado21 = 0

Lleva cuenta de cuántos trenes hay circulando que entraron por el tramo 2 en dirección al tramo 1.

```
public class MonitorTunel_3
        extends Monitor {
    private final Tramo tramo1;
    private final Tramo tramo2;

    private int ocupado12 = 0;
    private int ocupado21 = 0;

    public MonitorTunel_3(Tramo tramo1, Tramo tramo2) {
        this.tramo1 = tramo1;
        this.tramo2 = tramo2;
    }
```

```
    public synchronized void entro(
        Tren tren, Tramo tramo, Enlace entrada) {
        if (tramo.equals(tramo1)) {
            while (ocupado21 > 0)
                waiting();
            ocupado12++;
        }
        if (tramo.equals(tramo2)) {
            while (ocupado12 > 0)
                waiting();
            ocupado21++;
        }
    }
```

```
    public synchronized void salgo(
        Tren tren, Tramo tramo, Enlace salida) {
        if (tramo.equals(tramo1))
            ocupado21--;
        if (tramo.equals(tramo2))
            ocupado12--;
        notifyAll();
    }
```

```
    private void waiting() {
        try {
            wait();
        } catch (InterruptedException ignored) {
        }
    }
}
```

3.6.4 Ejercicio 4

Se permiten varios trenes en la misma dirección; pero si hay trenes esperando en dirección opuesta, se les da prioridad para evitar que se monopolice la vía en una cierta dirección.

Variables de estado:

int ocupado12 = 0

Lleva cuenta de cuántos trenes hay circulando que entraron por el tramo 1 en dirección al tramo 2.

int ocupado21 = 0

Lleva cuenta de cuántos trenes hay circulando que entraron por el tramo 2 en dirección al tramo 1.

int esperando1 = 0

Lleva cuenta de cuántos trenes hay esperando a entrar por el tramo 1.

int esperando2 = 0

Lleva cuenta de cuántos trenes hay esperando a entrar por el tramo 2.

int ultimaEntrada = 1

Memoriza por dónde entró el último tren, para alterna.

```
public class MonitorTunel_4
        extends Monitor {
    private final Tramo tramo1;
    private final Tramo tramo2;

    private int esperando1 = 0;
    private int esperando2 = 0;
    private int ocupado12 = 0;
    private int ocupado21 = 0;
    private int ultimaEntrada = 1;

    public MonitorTunel_4(Tramo tramo1, Tramo tramo2) {
        this.tramo1 = tramo1;
        this.tramo2 = tramo2;
    }
```

```
    public synchronized void entro(
        Tren tren, Tramo tramo, Enlace entrada) {
        if (tramo.equals(tramo1)) {
            esperando1++;
            while (!puedoPasar(ocupado21, 1, esperando2))
                waiting();
            esperando1--;
            ocupado12++;
            ultimaEntrada = 1;
        }
        if (tramo.equals(tramo2)) {
            esperando2++;
            while (!puedoPasar(ocupado12, 2, esperando1))
                waiting();
            esperando2--;
            ocupado21++;
            ultimaEntrada = 2;
        }
    }
```

```
    private boolean puedoPasar(
        int ocupado, int tramo, int esperando) {
        if (ocupado > 0)
            return false;
        if (ultimaEntrada == tramo && esperando > 0)
            return false;
        return true;
    }
```

```
    public synchronized void salgo(
        Tren tren, Tramo tramo, Enlace salida) {
        if (tramo.equals(tramo1))
            ocupado21--;
        if (tramo.equals(tramo2))
            ocupado12--;
        notifyAll();
    }
```

```
    private void waiting() {
        try {
            wait();
        } catch (InterruptedException ignored) {
        }
    }
}
```

3.7 Barrera

Variables de estado:

int waiting = 0

Lleva cuenta de cuántas tareas han llegado a la barrera

boolean broken = false

Pasa a TRUE cuando se rompe la barrera y ya no hay que controlar más threads.

```
public class Barrier {
    private final int n;
    private int waiting = 0;
    private boolean broken = false;

    public Barrier(int n) {
        this.n = n;
    }
```

```
    public synchronized void await() {
        waiting++;
        while (!broken && waiting < n)
            waiting();
        broken = true;
        waiting = 0;
        notifyAll();
    }
```

```
    private void waiting() {
        try {
            wait();
        } catch (InterruptedException ignored) {
        }
    }
```

3.8 Barrera Cíclica

3.8.1 Solución con contadores:

Variables de estado:

int waitingIn = 0;

Lleva la cuenta de cuántas tareas han llegado a la barrera y esperan a entrar.

int waitingOut = 0;

Lleva la cuenta de cuántas tareas están pendientes de salir en el grupo actual.

```
    private int waitingIn = 0;
    private int waitingOut = 0;
```

```
    public synchronized void await() {
        while (waitingOut > 0)
            waiting();
        waitingIn++;

        while (waitingIn < n && waitingOut == 0)
            waiting();

        if (waitingIn >= n) {
            waitingOut = n;
            waitingIn -= n;
        }
        waitingOut--;
        notifyAll();
    }
```

```
    private void waiting() {
        try {
            wait();
        } catch (InterruptedException ignored) {
        }
    }
```

3.8.2 Solución con conjuntos

Es un tipo de solución que puede servir para trazar el estado de la barrera a efectos de depuración de código.

Variables de estado:

Set<Thread> incomingGroup = new HashSet<Thread>();

Lleva la cuenta de cuántas tareas han llegado a la barrera y esperan a entrar.

Set<Thread> outgoingGroup = new HashSet<Thread>();

Lleva la cuenta de cuántas tareas están pendientes de salir en el grupo actual.

```
private Set<Thread> incomingGroup = new HashSet<Thread>();
private Set<Thread> outgoingGroup = new HashSet<Thread>();
```

```
    public synchronized void await() {
        Thread me = Thread.currentThread();
        while (outgoingGroup.size() > 0)
            waiting();
        incomingGroup.add(me);

        while (incomingGroup.size() < n
            && outgoingGroup.size() == 0)
            waiting();

        if (incomingGroup.size() == n) {
            outgoingGroup.addAll(incomingGroup);
            incomingGroup.clear();
        }
        outgoingGroup.remove(me);
        notifyAll();
    }
```

```
    private void waiting() {
        try {
            wait();
        } catch (InterruptedException ignored) {
        }
    }
```

3.9 Intercambiador (Exchanger<V>)

Variables de estado:

int in = 0

Lleva la cuenta de cuántas threads han entrado a sincronizar. Al principio hay 0; cuando llega a 2, pasamos a liberar las tareas y regresamos a 0.

int out = 2

Lleva la cuenta de cuántas tareas han salido. Cuando in llega a 2, out se pone a 0 y según van saliendo, se incrementa hasta llegar a 2. Se inicializa a 2 indicando que no queda nada por sacar.

V[] dato = new V[2]

Guarda los datos de las thread primera y segunda.

```
    private int in = 0;
    private int out = 2;
    private V[] dato = (V[]) new Object[2];
```

```
public synchronized V exchange(V x) {
    while (out < 2)      // datos pendientes de salir
        waiting();

    dato[in++] = x;
    if (in == 2)
```

```
        out = 0;          // suficientes entradas
    while (in < 2)
        waiting();        // necesitamos mas entradas

    V d = dato[out];
    dato[out] = null;
    out++;
    if (out == 2)
        in = 0;
    notifyAll();
    return d;
}
```

```
public void waiting() {
    try {
        wait();
    } catch (InterruptedException ignored) {
    }
}
```

3.10 La cena de los filósofos (Dining philosophers)

Variables de estado

No necesitamos más que saber si los tenedores están en uso o no.

```
public synchronized void take(Fork left, Fork right) {
    while (left.isInuse() || right.isInuse())
        waiting();
    left.take();
    right.take();
}
```

```
public synchronized void leave(Fork left, Fork right) {
    left.leave();
    right.leave();
    notifyAll();
}
```

```
private void waiting() {
    try {
        wait();
    } catch (InterruptedException ignored) {
    }
}
```

4 Tareas en background

No es exactamente un ejercicio. Es un repaso a las diferentes formas que tiene un programa (o tarea principal) de lanzar otra tarea en background, dedicarse a los suyo y recoger el resultado de la otra tarea cuando lo necesite, esperando a que termine si fuera necesario.

4.1 Esto no funciona

```
public class User0 {
    public static void main(String[] args)
            throws InterruptedException {
        BgTask0 task = new BgTask0();
        task.start();
        Thread.sleep(2000);
        String result = task.getResult();
        System.out.println("result: " + result);
    }
}
```

```
public class BgTask0
        extends Thread {
    private String result;

    @Override
    public void run() {
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
        }
        result = "done";
    }

    public String getResult() {
        return result;
    }
}
```

4.2 Usando join

Es probablemente el método más simple.

Permite que varias thread actúen como clientes. El resultado sólo se calcula una vez y todos los clientes esperan ordenadamente para recogerlo.

Inconveniente: si la thread en background lanza una excepción, nadie se entera.

```java
public class User01 {
    public static void main(String[] args)
            throws InterruptedException {
        BgTask01 task = new BgTask01();
        task.start();
        Thread.sleep(10000);
        task.join();
        String result = task.getResult();
        System.out.println("result: " + result);
    }
}
```

```java
public class BgTask01
        extends Thread {
    private String result;

    @Override
    public void run() {
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
        }
        System.out.println("end:    " + new Date());
        this.result = "done";
    }

    public String getResult() {
        return result;
    }
}
```

4.3 Programando la tarea auxiliar como monitor

No es muy complejo.

Permite que varias thread actúen como clientes. El resultado sólo se calcula una vez y todos los clientes esperan ordenadamente para recogerlo.

Inconveniente: si la thread en background lanza una excepción, nadie se entera.

```java
public class User02 {
    public static void main(String[] args)
            throws InterruptedException {
        BgTask02 task = new BgTask02();
        task.start();
        Thread.sleep(10000);
        String result = task.getResult();
        System.out.println("result: " + result);
    }
}
```

```
public class BgTask02
        extends Thread {
    private String result;

    @Override
    public void run() {
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
        }
        System.out.println("end:    " + new Date());
        setResult("done");
    }

    private synchronized void setResult(String result) {
        this.result = result;
        notifyAll();
    }

    public synchronized String getResult()
            throws InterruptedException {
        while (result == null)
            wait();
        return result;
    }
}
```

4.4 Con un contador pestillo

No es muy complejo; pero recurre a un tercer objeto para coordinar actividades.

Permite que varias thread actúen como clientes. El resultado sólo se calcula una vez y todos los clientes esperan ordenadamente para recogerlo.

Inconveniente: si la thread en background lanza una excepción, nadie se entera.

```
public class User03 {
    public static void main(String[] args)
            throws InterruptedException {
        CountDownLatch latch = new CountDownLatch(1);
        BgTask03 task = new BgTask03(latch);
        task.start();
        Thread.sleep(10000);

        latch.await();
        String result = task.getResult();
        System.out.println("result: " + result);
    }
}
```

```
public class BgTask03
        extends Thread {
    private final CountDownLatch latch;
    private String result;

    public BgTask03(CountDownLatch latch) {
        this.latch = latch;
    }

    @Override
    public void run() {
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
        }
        result = "done";
        latch.countDown();
    }

    public String getResult() {
        return result;
    }
}
```

4.5 Modelo productor-consumidor

No es muy complejo; pero recurre a un tercer objeto para coordinar actividades.

No permite que varias thread actúen como clientes. El resultado sólo se calcula una vez y el primer cliente se lo lleva.

Inconveniente: si la thread en background lanza una excepción, nadie se entera.

Flexibilidad: permite que la thread en background siga produciendo resultados, y cada cliente usará uno de ellos.

```
public class User04 {
    public static void main(String[] args)
            throws InterruptedException {
        BlockingQueue<String> queue =
          new ArrayBlockingQueue<String>(1);
        BgTask04 task = new BgTask04(queue);
        task.start();
        Thread.sleep(10000);
        String result = queue.take();
        System.out.println("result: " + result);
    }
}
```

```
public class BgTask04
        extends Thread {
    private final BlockingQueue<String> queue;

    public BgTask04(BlockingQueue<String> queue) {

        this.queue = queue;
    }

    @Override
    public void run() {
        try {
            Thread.sleep(5000);
            queue.put("done");
        } catch (InterruptedException e) {
        }
    }

}
```

4.6 Usando ejecutores de tareas y promesas de futuro

En cierto sentido es el esquema más “profesional” en el sentido de que en lugar de ir lanzando threads, le encargamos la ejecución a un servicio profesional de ejecución de threads.

Permite que varios clientes compartan el resultado. El resultado sólo se calcula una vez y todos esperan ordenadamente antes de recogerlo.

Y si la thread en background lanza una excepción, todos los clientes la reciben.

```
public class User05 {
    public static void main(String[] args)
            throws InterruptedException,
                   ExecutionException {
        ExecutorService pool =
          Executors.newFixedThreadPool(1);

        BgTask05 task = new BgTask05();
        Future<String> future = pool.submit(task);
        Thread.sleep(10000);
        String result = future.get();
        System.out.println("result: " + result);
    }
}
```

```
public class BgTask05
        implements Callable<String> {
    @Override
    public String call()
            throws Exception {
        System.out.println("start: " + new Date());
        Thread.sleep(5000);
        System.out.println("end:   " + new Date());
        return "done";
    }
}
```

5 Exámenes

5.1 Mayo 2012

Escriba una clase con un contador interno, que se incrementa cada vez que se invoca el método siguiente(). La clase debe poderse utilizar en un programa concurrente. Además, la clase proporciona otros dos métodos, esperarPar() y esperarImpar(), que hacen que la hebra (thread) que los invoca se quede bloqueada hasta que el valor del contador sea par o impar, respectivamente.

Se supone que el intervalo entre dos invocaciones consecutivas de siguiente() es suficiente para que todas las hebras que estuvieran bloqueadas puedan continuar. El esquema de la clase es el siguiente:

```
public class Secuenciador {
    private int numero = 0;

    public int siguiente() {...}
        // devuelve 1 la primera vez que se invoca,
        // 2 la segunda, etc.

    public void esperarPar() {..}
       // suspende la ejecución de la hebra
       // hasta que el valor sea par

    public void esperarImpar() {...}
       // suspende la ejecución de la hebra
       // hasta que el valor sea impar
   }
```

5.2 Junio 2012

Sea un puente con capacidad para un vehículo y dos accesos: norte y sur. En caso de que haya vehículos intentando entrar por los dos accesos, debe entrar un vehículo por el extremo en el que haya más esperando (si el número de vehículos esperando en cada extremo es el mismo, no es necesario imponer un orden). En el caso de que intente entrar una ambulancia, tendrá prioridad sobre el resto de vehículos. No es necesario considerar el caso en que dos ambulancias intenten acceder simultáneamente al puente.

Se pide desarrollar una clase monitor GestorPuente que gestione el acceso al puente, según la especificación previa. Los métodos de esta clase no retornan valores. El esqueleto de la clase es el siguiente:

```
public class GestorPuente { ...
       . . . void entrarNorte () { . . .}
       . . . void entrarSur () { . . .}
       . . . void entrarAmbulancia () { . . .}
       . . . void salirPuente(){. . .}
   }
```

5.3 Julio 2012

Se quiere desarrollar un sistema para controlar la temperatura y el número de personas que se encuentran en una sala de un museo. En condiciones normales, se permiten 50 personas en la sala. Si la temperatura sube por encima de un umbral (tUmbral = 30), se limita el número de personas a 35. Si cuando se detecta este suceso el número de personas en la sala es mayor que 35, no es necesario desalojarlas.

Si una persona jubilada intenta entrar, tendrá prioridad frente al resto de personas que estén esperando.

Cada persona se representa mediante una hebra. Además, hay una hebra que mide periódicamente la temperatura de la sala y notifica su valor al sistema. Se pide desarrollar un monitor (GestorSala) que sincronice a las hebras que representan personas y a la hebra que mide la temperatura, de acuerdo con las especificaciones anteriores. El monitor debe proporcionar los siguientes métodos:

```
... void entrarSala()
   // se invoca cuando una persona
   // quiere entrar en la sala.

... void entrarSalaJubilado()
   // se invoca cuando una persona jubilada
   // quiere entrar en la sala.

... void salirSala()
   // se invoca cuando una persona, jubilada o no,
   // quiere salir de la sala.

... void notificarTemperatura(int temperatura)
   // lo invoca la hebra que mide la temperatura
   // de la sala para indicar el último valor medido.
```

No es necesario garantizar que el orden de acceso a la sala coincide con el orden de llegada a la puerta de entrada.

5.4 Abril 2013

Sean tres hebras (threads), T1, T2 y T3, que utilizan tres recursos, R1, R2 y R3. La hebra T1 sólo necesita el recurso R1. La hebra T2 necesita los recursos R2 y R3. Por último, la hebra T3 requiere los tres recursos, R1, R2 y R3.

Escriba un monitor que controle el acceso de las hebras a los recursos. Cada hebra solicita los recursos que necesita invocando un método del monitor. Cuando una hebra termina de usar los recursos que necesita, lo indica para que otras hebras puedan usarlos. El monitor ha de asegurar que ningún recurso es utilizado por más de una hebra a la vez.

El esqueleto del monitor con los nombres de los métodos es:

```
    class Monitor {
    ...
    ... requiereR1 (...) { ... }
    ... requiereR2_R3 (...) { ... }
    ... requiereR1_R2_R3 ( ...) { ...}
    ... liberaR1(...) { ... }
    ... liberaR2_R3 (...) { ... }
    ... liberaR1_R2_R3 ( ...) { ...}
    }
```

5.5 Junio 2013

Desarrolle un monitor en Java que gestione el despegue de aviones y avionetas en un aeropuerto como se especifica a continuación:

Los aviones al despegar generan turbulencias, por lo que entre dos despegues consecutivos tiene que transcurrir un intervalo de tiempo mínimo:

- 3 minutos después del despegue de un avión.
- 2 minutos después del despegue de una avioneta.

Además, se debe impedir que despeguen consecutivamente dos avionetas si hay aviones esperando. No hay restricciones de este tipo respecto a los aviones.

El monitor responde al siguiente esquema:

```
class GestorDespegue {
    ...
    ... despegarAvion() {...}
        // lo invoca un avión cuando quiere despegar

    ... despegarAvioneta() {...}
        // lo invoca una avioneta cuando quiere despegar

    ... autorizarDespegue() {...}
        // lo invoca el temporizador (ver má adelante)
        // para indicar que ha transcurrido el intervalo
        // mínimo desde el despegue anterior

}
```

Para gestionar este intervalo de tiempo, se dispone de una clase Temporizador, cuya interfaz se muestra a continuación. El método iniciarTemporizador arranca un temporizador que deja pasar un cierto tiempo. Cuando el tiempo expira, se invoca el método autorizarDespegue del objeto de la clase GestorDespegue que se pasa en el constructor. No es necesario desarrollar esta clase.

```
public class Temporizador {
    public Temporizador(GestorDespegue gestor) {. . .}
    public void iniciarTemporizador(int minutos) {. . .}
}
```

5.6 Julio 2013

Escriba un monitor en java que controle el acceso a un parking de coches. El parking tiene un número de plazas N, y dispone de dos accesos, Este y Oeste.

Si el parking no está lleno, se admiten entradas por ambos accesos libremente. Si el parking está lleno, los coches deben esperar a que haya plazas, en cuyo caso el monitor debe alternar los accesos de los coches por las entradas Este y Oeste. Cuando un coche abandona el parking, se considera irrelevante el acceso que usa para salir.

El esqueleto del monitor con los nombres de los métodos es:

```
class GestorGaraje {
    ...
    ... GestorGaraje (int numPlazas) {...}
    ... entraCochePorEste (...) {...}
    ... entraCochePorOeste (...) {...}
    ... saleCoche (...) {...}
}
```

5.7 Abril 2014

Sea un cruce de calles, por el que circulan coches de oeste a este y de norte a sur. Para regular el tráfico hay dos semáforos, uno en la entrada oeste y otro en la entrada norte, y dos sensores que se activan cuando llega un coche a la entrada correspondiente. También hay sensores que indican la salida del cruce.

Se desea desarrollar un monitor en Java que simule la gestión de los semáforos de la siguiente forma:

- Los coches se modelan como hebras (*threads*) que invocan un método llegaNorte() o llegaOeste() cuando llegan al cruce.
- Si el semáforo correspondiente está en verde, el coche pasa inmediatamente. Si está en rojo, espera hasta que esté verde.
- Los coches tardan un cierto tiempo en pasar el cruce. Al salir invocan un método sale() en el monitor.

- Una hebra de control llama a un método cambiaSemáforos() cada cierto tiempo para cambiar la configuración de los semáforos.!

El monitor responde al siguiente esquema:

```
... class GestorCruce {
...
    ... llegaNorte() {...}
    // lo invoca un coche que llega por el N

    ... llegaOeste() {...}
    // lo invoca un coche que llega por el W

    ... sale() {...}
    // lo invoca un coche que sale del cruce

    ... cambiaSemáforos()
    // lo invoca la hebra de control
...
}
```

Se pide: desarrollar el código completo del monitor.

5.8 Julio 2014

Un sistema de gestión de un almacén de piezas está compuesto por un conjunto de productores y de consumidores, que se modelan mediante hebras. Las hebras productoras añaden piezas, mientras que las consumidoras las solicitan y retiran.

Se pide diseñar un monitor GestorPiezas que gestione las interacciones de estas hebras, cuya interfaz está formada por los siguientes métodos:

… void solicitarPiezas (int cantidadPiezas)

este método lo invocan las hebras consumidoras cuando quieren solicitar una cantidad de piezas determinada. Si hay piezas suficientes, se le proporcionan inmediatamente (se actualiza el número de piezas almacenadas). Si no las hay, se bloquea la hebra hasta que haya suficientes. En este caso, hay que bloquear al resto de hebras consumidoras hasta que se satisfaga la petición pendiente.

… void agregarPiezas (int cantidadPiezas)

este método lo invocan las hebras productoras para añadir piezas al almacén. La cantidad de piezas que se pueden almacenar es ilimitada.

Nota: el número de piezas debe ser positivo en todos los casos.

5.9 Abril 2015

Se pretende sincronizar la fabricación en una línea de ensamblado de mesas. Hay varios fabricantes de patas, que las depositan en una línea con un límite de capacidad MAX_NUM_PATAS. Cuando se llena, los fabricantes dejan de producir patas hasta que haya hueco libre. Hay varios fabricantes de tableros, que depositan en otra línea de

capacidad limitada MAX_NUM_TABLEROS. Por último, hay varios ensambladores de mesas: cada uno coge cuatro patas y un tablero y ensambla una mesa.

Se trata de escribir en Java un monitor que sincronice estos tres sistemas, de forma que la producción se detenga cuando se alcanza la capacidad máxima de almacenamiento (de patas o tableros independientemente) y sistema de ensamblaje no avance si le faltan piezas para hacer una nueva mesa.

NO ESCRIBA NINGÚN CÓDIGO para los subsistemas de producción y ensamblaje.

El sincronizador responde al siguiente esquema:

```
...class Sincronizador {
   ...

   // lo invoca el productor de patas por cada una
   ... ponPata() {...}

   // lo invoca el productor de tableros
   ... ponTablero() {...}

   // lo invoca el ensamblador de mesas
   ... cogePatasyTablero () {...}
...
}
```

Se pide: desarrollar el código completo del monitor Sincronizador.

6 Soluciones a los exámenes

6.1 Mayo 2012

Variables de estado:

int numero = 0

número siguiente

```
    private int numero = 0;
```

```
    public synchronized int siguiente() {
        notifyAll();
        return numero++;
    }
```

```
    public synchronized void esperarPar() {
        while (numero % 2 != 0)
            waiting();
    }
```

```
public synchronized void esperarImpar() {
    while (numero % 2 == 0)
        waiting();
}
```

```
private void waiting() {
    try {
        wait();
    } catch (InterruptedException ignored) {
    }
}
```

6.2 Junio 2012

Variables de estado:

boolean hayCocheEnPuente = false

Indica si hay un coche dentro del puente

int nCochesNorte = 0

Indica el número de coches que están esperando para entrar en el puente por el Norte

int nCochesSur = 0

Indica el número de coches que están esperando para entrar en el puente por el Sur

boolean hayAmbulancia = false

Indica si hay una ambulancia esperando

```
private boolean hayCocheEnPuente = false;
private int nCochesNorte = 0;
private int nCochesSur = 0;
private boolean hayAmbulancia = false;
```

```
public synchronized void entrarNorte()
        throws InterruptedException {
    nCochesNorte++;
    while (hayCocheEnPuente
            || !hayAmbulancia
            || nCochesNorte < nCochesSur)
        wait();
    hayCocheEnPuente = true;
    nCochesNorte--;
}
```

```
public synchronized void entrarSur()
        throws InterruptedException {
    nCochesSur++;
    while (hayCocheEnPuente
            || hayAmbulancia
            || nCochesSur < nCochesNorte)
        wait();
    hayCocheEnPuente = true;
    nCochesSur--;
}
```

```
public synchronized void entrarAmbulancia()
        throws InterruptedException {
    hayAmbulancia = true;
    while (hayCocheEnPuente)
        wait();
    hayCocheEnPuente = true;
    hayAmbulancia = false;
}
```

```
public synchronized void salirPuente() {
    hayCocheEnPuente = false;
    notifyAll();
}
```

6.3 Julio 2012

Variables de estado:

int nPersonas = 0

Número de personas en la sala

int nMaxPersonas = nMaxPersonasNormalT

Número máximo admisible en las condiciones actuales de temperatura

int nJubilados = 0

Número de jubilados pendientes de entrar

```
private final int tUmbral = 30;
private final int nMaxPersonasNormalT = 50;
private final int nMaxPersonasAltaT = 35;

private int nPersonas = 0;
private int nMaxPersonas = nMaxPersonasNormalT;
private int nJubilados = 0;
```

```
public synchronized void entrarSalaJubilado()
        throws InterruptedException {
    nJubilados++;
    while (nPersonas >= nMaxPersonas)
        wait();
    nJubilados--;
    nPersonas++;
}
```

```
public synchronized void entrarSala()
        throws InterruptedException {
    while (nPersonas >= nMaxPersonas
        || nJubilados > 0)
        wait();
    nPersonas++;
}
```

```
public synchronized void salirSala()
        throws InterruptedException {
    nPersonas--;
    notifyAll();
}
```

```
public synchronized void notificarTemperatura(
    int temperatura) {
    if (temperatura > tUmbral)
        nMaxPersonas = nMaxPersonasAltaT;
    if (temperatura < tUmbral)
        nMaxPersonas = nMaxPersonasNormalT;
}
```

6.4 Abril 2013

Variables de estado;

boolean ocupadoR1 = false

TRUE si el recurso R1 está ocupado

boolean ocupadoR2 = false

TRUE si el recurso R2 está ocupado

boolean ocupadoR3 = false

TRUE si el recurso R3 está ocupado

```
private boolean ocupadoR1 = false;
private boolean ocupadoR2 = false;
private boolean ocupadoR3 = false;
```

```
public synchronized void requiereR1()
        throws InterruptedException {
    while (ocupadoR1)
        wait();
    ocupadoR1 = true;
}
```

```
public synchronized void requiereR2_R3()
        throws InterruptedException {
    while (ocupadoR2 || ocupadoR3)
        wait();
    ocupadoR2 = true;
    ocupadoR3 = true;
}
```

```
public synchronized void requiereR1_R2_R3()
        throws InterruptedException {
    while (ocupadoR1 || ocupadoR2 || ocupadoR3)
        wait();
    ocupadoR1 = true;
    ocupadoR2 = true;
    ocupadoR3 = true;
}
```

```
public synchronized void liberaR1() {
    ocupadoR1 = false;
    notifyAll();
}
```

```
public synchronized void libreraR2_R3() {
    ocupadoR2 = false;
    ocupadoR3 = false;
    notifyAll();
}
```

```
public synchronized void liberaR1_R2_R3() {
    ocupadoR1 = false;
    ocupadoR2 = false;
    ocupadoR3 = false;
    notifyAll();
}
```

6.5 Junio 2013

Variables de estado:

boolean pistaOcupada

TRUE si hay un avión o una avioneta en pista

int nAvionesEsperando = 0

Número de aviones en cola para despegar.

boolean anteriorAvioneta = false

TRUE si lo último que despegó fue una avioneta.

```
private boolean pistaOcupada = true;
private int nAvionesEsperando = 0;
private boolean anteriorAvioneta = false;
```

```
public synchronized void despegarAvion()
        throws InterruptedException {
    nAvionesEsperando++;
    while (pistaOcupada)
        wait();
    nAvionesEsperando--;
    anteriorAvioneta = false;
    temporizador.iniciarTemporizador(tiempoAvion);
    pistaOcupada = true;
}
```

```
public synchronized void despegarAvioneta()
        throws InterruptedException {
    while (pistaOcupada
       || (nAvionesEsperando > 0 && anteriorAvioneta))
        wait();
    anteriorAvioneta = true;
    temporizador.iniciarTemporizador(tiempoAvioneta);
    pistaOcupada = true;
}
```

```
public synchronized void finTemporizador()
        throws InterruptedException {
    pistaOcupada = false;
    notifyAll();
}
```

6.6 Julio 2013

Variables de estado

int numPlazas

Número de plazas en el aparcamiento. Es un valor constante.

int numCoches

Lleva cuenta del número de coches aparcados.

boolean prioridadE

TRUE si tiene prioridad para entrar un coche que llegue por el ESTE.

FALSE si tiene prioridad para entrar un coche que llegue por el OESTE.

int esperandoE

Lleva cuenta del número de coches esperando para entrar por el ESTE.

int esperandoW

Lleva cuenta del número de coches esperando para entrar por el OESTE.

```
class GestorGaraje {
    private final int numPlazas;
    private int numCoches = 0;
    private boolean prioridadE = true;
    private int esperandoE = 0;
    private int esperandoW = 0;

    GestorGaraje(int numPlazas) {
        this.numPlazas = numPlazas;
    }
```

```
    synchronized void entraCochePorEste() {
        esperandoE++;
        while (!puedoEntrar(prioridadE, esperandoW))
            waiting();
        esperandoE--;
        prioridadE = false;
        numCoches++;
    }
```

```
    synchronized void entraCochePorOeste() {
        esperandoW++;
        while (!puedoEntrar(!prioridadE, esperandoE))
            waiting();
        esperandoW--;
        prioridadE = true;
        numCoches++;
    }
```

```
    private boolean puedoEntrar(
      boolean miPrioridad,
      int esperandoEnLaOtra) {
        if (numCoches >= numPlazas)
            return false;
        if (miPrioridad)
            return true;
        return esperandoEnLaOtra == 0;
    }
```

```
    synchronized void saleCoche() {
        numCoches--;
        notifyAll();
    }
```

```
    private void waiting() {
        try {
            wait();
        } catch (InterruptedException ignored) {
        }
    }
}
```

6.7 Abril 2014

Variables de estado:

boolean norteVerde = true

Estado de los semáforos: se puede representar mediante un booleano por semáforo, igual a true cuando esté verde y false cuando esté rojo (o viceversa). En realidad basta con una sola variable booleana, teniendo en cuenta que cuando uno de los semáforos está verde el otro está rojo.

boolean cochePasando = false

Número de coches en el cruce. Se pude representar mediante un entero o, si sólo se permite un coche a la vez, mediante un booleano.

```
private boolean norteVerde = true; // => oeste está rojo
private boolean cochePasando = false;
```

```
public synchronized void entraNorte()
    throws InterruptedException {
    while (!norteVerde || cochePasando)
        wait();
    cochePasando = true;
}
```

```
public synchronized void entraOeste()
    throws InterruptedException {
    while (norteVerde || cochePasando)
        wait();
    cochePasando = true;
}
```

```
public synchronized void sale() {
    cochePasando = false;
    notifyAll();
}
```

```
public synchronized void cambiaSemaforos(){
    norteVerde = !norteVerde;
    notifyAll();
}
```

6.8 Julio 2014

Variables de estado:

int cantidadAlmacen

Número de piezas en el almacén en un momento dado.

boolean peticionPendinte

TRUE si hay un cliente esperando a ser servido.

```
private int cantidadAlmacen = 0;
private boolean peticionPendiente = false;
```

```
public synchronized void solicitarPiezas(
        int cantidadPiezas)
        throws InterruptedException {
      while (peticionPendiente) wait();
      peticionPendiente = true;

      while (cantidadAlmacen < cantidadPiezas) wait();

      cantidadAlmacen -= cantidadPiezas;
      peticionPendiente = false;
      notifyAll();
}
```

```
public synchronized void agregarPiezas(
        int cantidadPiezas)
        throws InterruptedException {
      cantidadAlmacen += cantidadPiezas;
      notifyAll();
}
```

6.9 Abril 2015

Variables de estado:

int nPatas = 0

Número de patas en el almacén en un momento dado.

int nTableros = 0

Número de tableros en el almacén en un momento dado.

```
public class Sincronizador {
    public static final MAX_NUM_PATAS= 1000;
    public static final MAX_NUM_TABLEROS = 1000;

    private int nPatas= 0;
    private int nTableros= 0;
```

```
// lo invoca el productor de patas por cada una
public synchronized void ponPata() {
    while (nPatas >= MAX_NUM_PATAS)
      espera();
    nPatas++;
    notifyAll();
}
```

```
// lo invoca el productor de tableros
public synchronized void ponTablero() {
    while (nTableros >= MAX_NUM_TABLEROS)
        espera();
    nTableros++;
    notifyAll();
}
```

```
// lo invoca el ensamblador de mesas
public syncronized void cogePatasyTablero () {
    while (nPatas < 4 || nTableros < 1)
        espera();
    nPatas-= 4;
    nTableros-= 1;
    notifyAll();
}
```

```
private void espera() {
    try {
        wait();
    } catch (InterruptedException ignored) {
}
```