# Concurrencia
# Exámenes resueltos

*josé a. mañas*

*26.4.2018*

## Contenido

# 1 Exámenes

## 1.1 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
   }
```

## 1.2 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(){. . .}
   }
```

## 1.3 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.

## 1.4 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 ( ...) { ...}
}
```

## 1.5 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) {. . .}
}
```

## 1.6 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 (...) {...}
}
```

## 1.7 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.

## 1.8 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.

## 1.9 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.

## 1.10 Junio 2015

Se tiene un sistema con dos hebras (H1 y H2) que acceden continuamente a un recurso compartido, que se debe usar con exclusión mutua. Para que el sistema funcione correctamente las hebras tienen que acceder según el siguiente patrón, que se debe repetir cíclicamente:  H1, H1, H2, H1, H2, como se ve en la figura siguiente:


Se pide desarrollar el monitor (GestorCiclosAcceso) para sincronizar las hebras según el comportamiento descrito. El monitor debe proporcionar las siguientes operaciones:

- ... *void accederH1()*: Este método lo invoca la hebra H1 para solicitar acceso al recurso compartido.
- ... *void accederH2()*: Este método lo invoca la hebra H2 para solicitar acceso al recurso compartido.
- ... *void liberar()*: Este método lo invocan las hebras para liberar el recurso compartido.

## 1.11 Junio 2016

En un programa existe una zona crítica especial, llamémosla "supersticiosa", en la que puede haber cualquier número de hebras, siempre que no sean trece.

Se pide realizar una clase sincronizada (según el esquema que da a continuación) que realice tal protocolo de sincronización. Para simplificar el código, no tenga en cuenta la excepción que pueda presentarse en la operación wait().

```
package es.upm.dit.adsw.super;

public class Supersticiosa {
...
	// quiero entrar en la zona crítica
	public synchronized void entrar() {
		...
	}

	// quiero salir de la zona crítica
	public synchronized void salir() {
		...
	}
}
```

NOTA. El enunciado se presta a cierta ambigüedad en la interpretación de qué significa que "no hayan 13 threads en la zona crítica".

Una interpretación es el modelo "discoteca" donde si llega el cliente nº 13 debe esperar a otro cliente que quiera entrar, o a que alguien de dentro quiera salir. Lo mismo si desea salir, dejando 13 clientes dentro: deben salir 2 o compensar con uno que desee entrar. En esta interpretación nos fijamos en el número de clientes con derecho a estar dentro, independientemente del momento preciso en que hagan uso de ese derecho. En esta interpretación puede haber … 10, 11, 12, 14, 15, 16, … clientes con derecho.

Otra interpretación es el modelo torno de entrada individual según el cual los clientes entran y salen de 1 en 1, no pudiendo entrar o salir emparejados. En esta interpretación, es imposible pasar de 12 clientes porque en algún momento habría 13.

## 1.12 Junio 2017

(Solo se contempla la parte de sincronización de threads por medio de un monitor)

Se quiere desarrollar un sistema compuesto de dos conjuntos de hebras. Un conjunto de hebras (editoras) producen contenidos sobre un tema y los envían a un monitor, Gestor. Otro conjunto de hebras (suscriptoras) reciben los contenidos publicados sobre

los temas a los que se suscriben. Las hebras editoras y suscriptoras utilizan los siguientes métodos del gestor para sincronizarse:

```
public class Gestor {
   ...
   public ... void enviarContenido(Tema unTema,
                                   Contenido unContenido) {...}
   public ... Contenido recibirContenido(Tema unTema) {...}
}
```

Completar la clase Gestor de forma que se comporte como un monitor, con lo necesario para sincronizar las hebras de las dos clases citadas. La implementación de los métodos del gestor debe cumplir la siguiente especificación:

```
... void enviarContenido(Contenido unContenido, Tema unTema)
```

Cuando una hebra editora invoca este método para publicar un contenido, se debe reanudar la ejecución de todas las hebras suscriptoras que estuvieran esperando este tema. El contenido debe estar disponible para las suscriptoras hasta que una hebra editora envíe el siguiente contenido.

```
... Contenido recibirContenido (Tema unTema)
```

Cuando una hebra suscriptora invoca este método, recibe el último contenido publicado si su tema coincide con el solicitado. Si el último contenido es de otro tema, la hebra se bloquea hasta que una hebra editora envíe un contenido del tema solicitado.

Clases auxiliares (no hay que implementarlas):

```
public class Contenido {
    private String contenido;

    public Contenido (String contenido) {
        this.contenido = contenido;
    }
  . .
}
```

```
public enum Tema {NACIONAL, INTERNACIONAL, CULTURA, DEPORTE;

    public static Tema random() {...}

}
```

```
public class Nap {
     ...
    /**
     * Duerme un periodo aleatorio entre los limites indicados.
     *
     * @param min milisegundos minimos.
     * @param max milisegundos maximos.
     */
    public static void random (int min, int max) {...}
```

### 1.13 Julio 2017

Varios ingenieros utilizan una base de datos para almacenar informes. Para ello disponen de una aplicación que permite almacenar un informe o recuperar un informe anterior. El acceso a los datos se organiza mediante una clase de Java que responde al esquema siguiente esquema:

```
public class Datos {
        …
        public … void guardar (String clave) {…}
        public … void consultar (String clave) {…}
        public … void terminar (String clave) {…}
}
```

Las aplicaciones se ejecutan concurrentemente, y llaman a guardar o a consultar antes de realizar una secuencia de operaciones de almacenar o recuperar informes, respectivamente. Cuando han terminado de realizar estas operaciones llaman a terminar. Puede haber varias aplicaciones consultando informes a la vez, pero no consultando y guardando, ni varias guardando a la vez.

Los objetos de la clase Ingeniero son hebras que ejecutan repetidamente el siguiente código:

```
datos.guardar(clave);
        for (int i = 1; i <= ndoc; i++) {
           // generar informe y almacenarlo en la base de datos
        }
datos.terminar(clave);
```

o bien

```
        datos.consultar(clave);
        for (int i = 1; i <= ndoc; i++) {
           // realizar consulta en la base de datos
        }
        datos.terminar(clave);
```

# 2 Soluciones a los exámenes

## 2.1 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) {
        }
    }
```

## 2.2 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();
}
```

## 2.3 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;
}
```

## 2.4 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();
}
```

## 2.5 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();
}
```

## 2.6 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) {
        }
    }
}
```

## 2.7 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();
}
```

## 2.8 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();
}
```

## 2.9 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) {
    }
```

## 2.10 Junio 2015

Variables de estado:

**private boolean busy = false;**

indica si el recurso está ocupado

**private int state = 0;**

indica en qué estado estamos; con el siguiente convenio

| state | situación | |
|---|---|---|
| 0 | espera H1 |  |
| 1 | espera H1 | |
| 2 | espera H2 | |
| 3 | espera H1 | |
| 4 | espera H2 | |



```
public synchronized void accederH1()
        throws InterruptedException {
    while (busy || !espera(1))
        waiting();
    busy = true;
    state = (state + 1) % 5;
}
```

```
public synchronized void accederH2()
        throws InterruptedException {
    while (busy || !espera(2))
        waiting();
    busy = true;
    state = (state + 1) % 5;
}
```

```
public synchronized void liberar() {
    busy = false;
    notifyAll();
}
```

```
private boolean espera(int next) {
    switch (state) {
        case 0:
        case 1:
        case 3:
            return next == 1;
        case 2:
        case 4:
            return next == 2;
    }
    return false;
}
```

## 2.11 Junio 2016

### 2.11.1 Solución 1

Variables de estado:

**private int n= 0;**

indica el número de peticiones de entrada - número de peticiones de salida

```
public class Supersticioso {
    public static final int MAGIC = 13;

    private int n= 0;

    public  void entrar() {
        n++;
        while (n == MAGIC)
            waiting();
        notifyAll();
    }

    public synchronized void salir() {
        n--;
        while (n == MAGIC)
            waiting();
        notifyAll();
    }

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

### 2.11.2 Solución 2

Sigue el modelo interpretativo de la solución 1; pero explicita el estado de cada tarea

ENTRANDO → DENTRO → SALIENDO

Variables de estado:

**private Set<Thread> entrando = new HashSet<>();**

las que quieren entrar

**private Set<Thread> dentro = new HashSet<>();**

las que están dentro

**private Set<Thread> saliendo = new HashSet<>();**

las que quieren salir

```
public class Supersticioso {
    public static final int MAGIC = 13;

    private Set<Thread> entrando = new HashSet<>();
    private Set<Thread> dentro = new HashSet<>();
    private Set<Thread> saliendo = new HashSet<>();

    public synchronized void entrar() {
        Thread me = Thread.currentThread();
        entrando.add(me);
        while (true) {
            balance();
            if (dentro.contains(me))
                return;
            waiting();
        }
    }

    public synchronized void salir() {
        Thread me = Thread.currentThread();
        saliendo.add(me);
        while (true) {
            balance();
            if (!dentro.contains(me))
                return;
            waiting();
        }
    }

    private void balance() {
        int n0 = dentro.size();
        int n1 = entrando.size();
        int n2 = saliendo.size();
        if (n0 + n1 - n2 != MAGIC) {
            dentro.addAll(entrando);
            dentro.removeAll(saliendo);
            entrando.clear();
            saliendo.clear();
            notifyAll();
        }
    }

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

### 2.11.3 Solución 3

Sigue la segunda interpretación del enunciado: modelo torno.

Variables de estado

**private int nDentro= 0;**

número de threads dentro de la zona crítica

```
public class Supersticioso {
    private int nDentro= 0;

    @Override
    public synchronized void entrar() {
        while (nDentro + 1 == 13)
            waiting();
        nDentro++;
    }

    @Override
    public synchronized void salir() {
        nDentro--;
        notifyAll();
    }


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

## 2.12 Junio 2017

Variables de estado:

**private Tema ultimoTema**

indica el último tema publicado

**private Contenido ultimoContenido**

indica el último contenido publicado

```
public class Gestor {
    private Tema ultimoTema;
    private Contenido ultimoContenido;

    public synchronized void enviarContenido(
            Tema unTema, Contenido unContenido) {
        this.ultimoTema = unTema;
        this.ultimoContenido = unContenido;
        notifyAll();
    }
```

```
    public synchronized Contenido recibirContenido(Tema unTema) {
```

```
        while (unTema != ultimoTema)
            waiting();
        return ultimoContenido;
    }
```

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

## 2.13 Julio 2017

Se trata de un problema del tipo readers-writers, donde puede haber varios readers (consultando) pero solamente un writer (guardando).

El estado del monitor puede resumirse en 2 variables:

**int nReaders**

número de readers en un momento dado

**boolean saving**

si hay alguien guardando en este momento

```
public class Datos {
    private int nReaders = 0;
    private boolean saving = false;
```

```
    public synchronized void guardar(String clave) {
        while (saving)
            waiting();
        nReaders++;
    }
```

```
    public synchronized void consultar(String clave) {
        while (saving || nReaders > 0)
            waiting();
        saving = true;
    }
```

```
    public synchronized void terminar(String clave) {
        if (nReaders > 0)
            nReaders--;
        saving = false;
        notifyAll();
    }
```

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

La solución presentada puede presentar situaciones de inanición (hambruna, *starvation*) cuando una serie de ingenieros se dedican a consultar datos de forma ininterrumpida y las peticiones para guardar datos no consiguen permiso para entrar. La solución típica consiste en dar prioridad a las escrituras (guardar) de forma que las autorizaciones para consultar no son concedidas si hay peticiones para guardar esperando.