Programación Concurrente
Programación Concurrente Multihilos - Java
Parte IV

Sincronización

En un escenario concurrente los hilos pueden cooperar o competir por los recursos. Nos interesan programas concurrentes en los que la ejecución de los hilos es asíncrona, es decir cada proceso se ejecuta a su propia velocidad. Sin embargo en este constexto puede ocurrir que un hilo para avanzar en su procesamiento dependa de ciertas condiciones. Alli surjen los distintos tipos de sincronización.

Cuando compiten por un recurso decimos que es **sincronización por competencia**, y debe trabajarse con mecanismos para lograr la exclusión mutua: semáforos binarios, métodos y bloques sincronizados y locks. (Recordemos que la exclusión mutua asegurar que la sección crítica de los recursos compartidos no son ejecutadas al mismo tiempo por varios hilos)

Cuando cooperan para poder avanzar decimos que es **sincronización por cooperación** (algunos autores la llaman *sincronización condicional*)**,** y en esta situación se requiere complementar las facilidades de sincronización vistas para exclusión mutua para lograr la cooperación entre los hilos. ¿Por qué hablamos de cooperación? Porque un hilo para avanzar puede depender de que se verifique una condición, y será algún otro de los hilos involucrados el que permitirá, a partir de sus acciones, que esa condición en algún momento se verifique para el hilo que espera.

En esta situación una forma de resolverlo sería utilizando para la sincronización una ***"espera activa"***. Pero esa solución NO ES APROPIADA, dado que es ineficiente ya que el hilo que espera por la condición esta ocupando ciclos de CPU sin poder avanzar, evitando que otros hilos que no necesitan esa condición puedan utilizarla y progresar. Entonces … hay que EVITAR LA ESPERA ACTIVA, y para ello hay que asegurar que el hilo se retardará si es necesario hasta que se verifique la condición.

En estos casos debemos trabajar utilizando protocolos de sincronización que permitan *"**bloquear**"* a los hilos que deben ser retrasados en espera de una condición. Trabajamos con mecanismos de ***espera y notificación***.

Estas construcciones soportan comunicación medianamente eficiente pero, al costo de mayor complejidad y mayor potencial para errores de programación.

Suponga la situación siguiente: 3 pasteleros van a realizar una tarta de crema y frutillas de forma conjunto. Un pastelero se encargará de realizar la masa mentras que el otro pastelero preparará la crema. La 3ra persona será la encargada de armar la tarta, para ello debera esperar que la masa este lista y cocinada, y que la crema tambien este lista. Entonces dispondrá la crema sobre la masa y para finalizar ubicará las frutillas sobre la crema.

Aqui pueden darse 2 situaciones:

- que dispongan de 2 recipientes para trabajar la masa y la crema. Las personas encargadas de realizar la masa y preparar la crema podrán trabajar en paralelo, y avanzar de forma independiente una de la otra, dado que cada una podra utilizar un recipiente. (fig 1)

- que dispongan de solo 1 recipiente para trabajar la masa y la crema. En este caso las personas encargadas de realizar la masa y preparar la crema deberán ***"competir"*** por el recurso recipiente, que se transforma en un recurso compartido. (fig 2)

- en ambos casos la persona encargada de armar la tarta utilizando la masa ya cocida, la crema y las frutillas, deberá esperar que se de la condición de que: la masa este lista y cocida y la crema preparada, o sea tendrá que esperar que las otras personas le avisen o ***"notifiquen"*** que ya han realizado parte o todo su trabajo para que puede varificar la condición y avanzar.

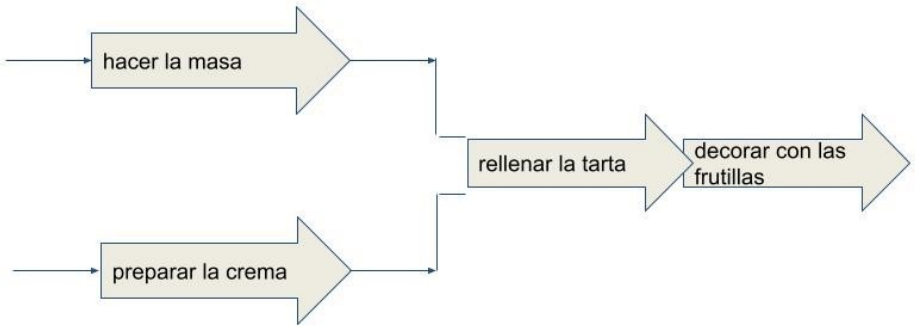


Fig 1: caso de 2 recipientes

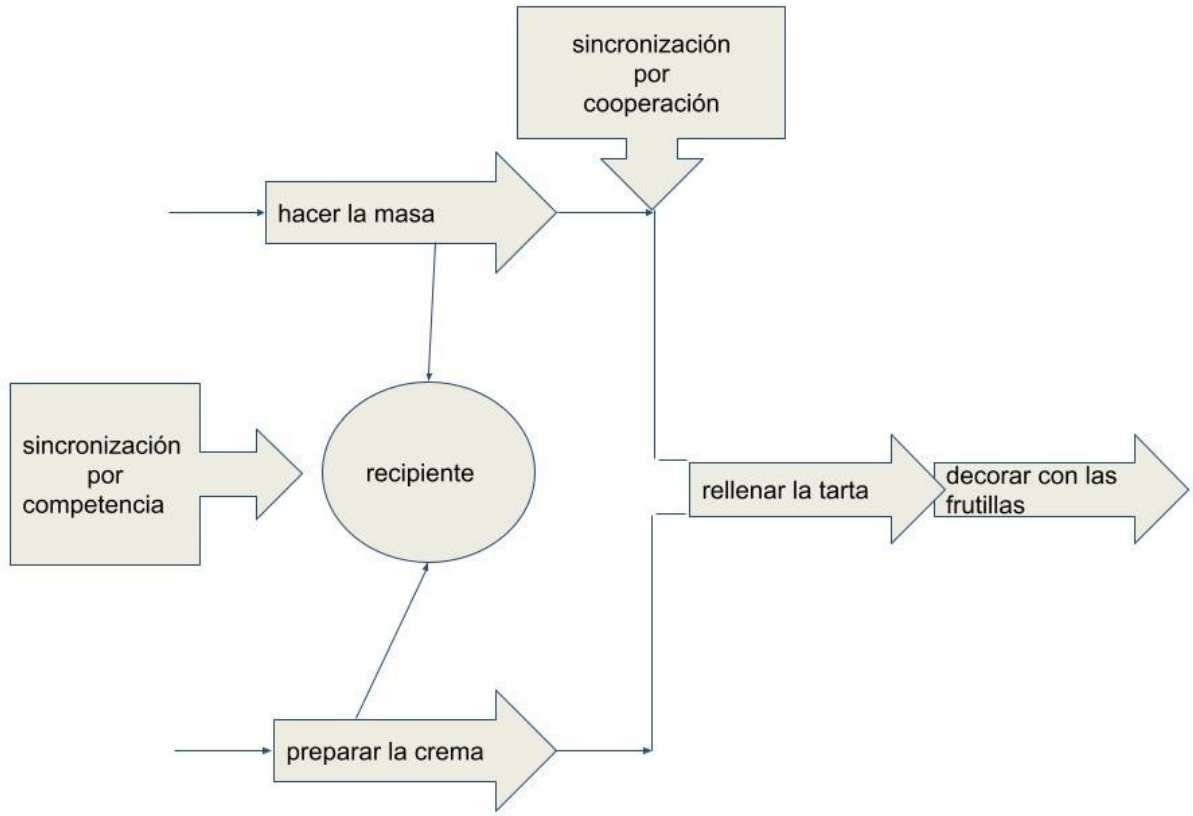


Fig 2: caso de 1 recipiente compartido

Semáforos generales.

Una forma de trabajar la sincronización por cooperación es utilizando **semáforos generales**.
Los semáforos son construcciones de control de concurrencia. Los hemos trabajado binarios, es decir con 0 o 1 permisos.
Los semáforos generales son usados para controlar el numero de actividades que pueden acceder a un recurso compartido o llevar a cabo una cierta acción al mismo tiempo. Permiten implementar la sincronización por cooperación-
Un semáforo maneja un conjunto de permisos virtuales, cuyo número inicial es dado al crear el semáforo. Las operaciones de adquisición y liberación de permisos son ***acquire(n)*** y ***release(k)***, donde ***n*** indica la cantidad de permisos que se requieren del semáforo y ***k*** indica la cantidad de permisos que se liberan, o sea que se devuelven al semáforo. Un hilo puede adquirir varios permisos y liberar varios permisos. Pero hay que tener MUCHO cuidado para no producir deadlock.

Cuando se trabaja con semáforos siempre se puede evitar la **espera activa**, dado que los hilos que no consiguen acceder al permiso o los permisos que requieren quedan bloqueados en el semáforo

Se puede utilizar un semáforo general, por ejemplo, en el problema del Barbero Dormilon para contemplar las sillas de la sala de espera de la barberia.

Entonces en el caso del estacionamiento del supermercado podrian utilizarse semáforos generales para controlar el espacio para las motos y el espacio para los auots. Y así se podría eliminar la "espera activa" en el sensorSalida.

Monitores

En el contexto de concurrencia, son un mecanismo de abstracción de datos: encapsulan la representación de recursos abstractos y proveen un conjunto de operaciones que son el único medio por el cual la representación es manipulada. En particular un monitor contiene variables que almacenan el estado de los recursos y métodos que implementan las operaciones sobre el recurso. Un monitor tiene la particularidad de asegurar la exclusión mutua en el acceso al recurso, y se utiliza para implementar un recursos compartido que cumpla con la propiedad de seguridad, es decir que un monitor es Thread-safe.

Un proceso puede acceder a las variables en un monitor solamente por medio de uno de sus métodos. La exclusión mutua se provee asegurando que la ejecución de los métodos en el mismo monitor no se solapa.

Es una estructura de datos con todos los métodos sincronizados y los atributos privados. Si bien en algunos lenguajes existe la clase Monitor que reune las características indicadas, Java no provee una clase MONITOR, pero logra "***la semántica de monitor"*** por la utilización de bloqueos intrínsecos, con el patrón ***synchronized – wait – notify***.

Es decir un monitor en Java se logra con la combinación de métodos sincronizados para lograr la exclusión mutua y el mecanismo de espera y notificación para lograr la cooperación entre los hilos.

En el modelo de objetos mixto, en el que identificamos objetos activos y objetos pasivos, los hilos son los objetos activos y los monitores los objetos pasivos.

> Todo objeto en Java tiene un lock implícito, un conjunto de espera y responde a los mensajes wait(), wait(timeout), notify() y notifyAll(). Con los metos "wait" se logra la espera y con los métodos "notify" se logra la notificación. Y con esa combinación se produce la sincronización por cooperación entrre los hilos.

A lo explicado en apuntes anteriores respecto al bloqueo implícito, se agrega el comportamiento siguiente:

Considere el objeto "monitor",

- cuando un hilo entra a un método sincronizado de monitor, adquiere el lock del objeto monitor de forma exclusiva, lo que significa que ningún otro hilo podrá ejecutar ese ni otro método sincronizado del mismo monitor (ya que el lock esta apropiado).
- una vez apropiado del lock puede ocurrir que no se den las condiciones necesarias para que el hilo continúe con la ejecución, y entonces debe esperar
- cuando el hilo debe esperar por una condición, o debe hacer uso del mensaje wait()/wait(timeout), con lo que el hilo queda bloqueado en el conjunto de espera del objeto que actúa como monitor, liberando la CPU, y ***liberando el lock*** del objeto monitor, de forma que otro hilo puede ejecutar con exclusividad el método mientras él espera su condición, (u otro metodo sincronizado sobre el mismo objeto monitor).

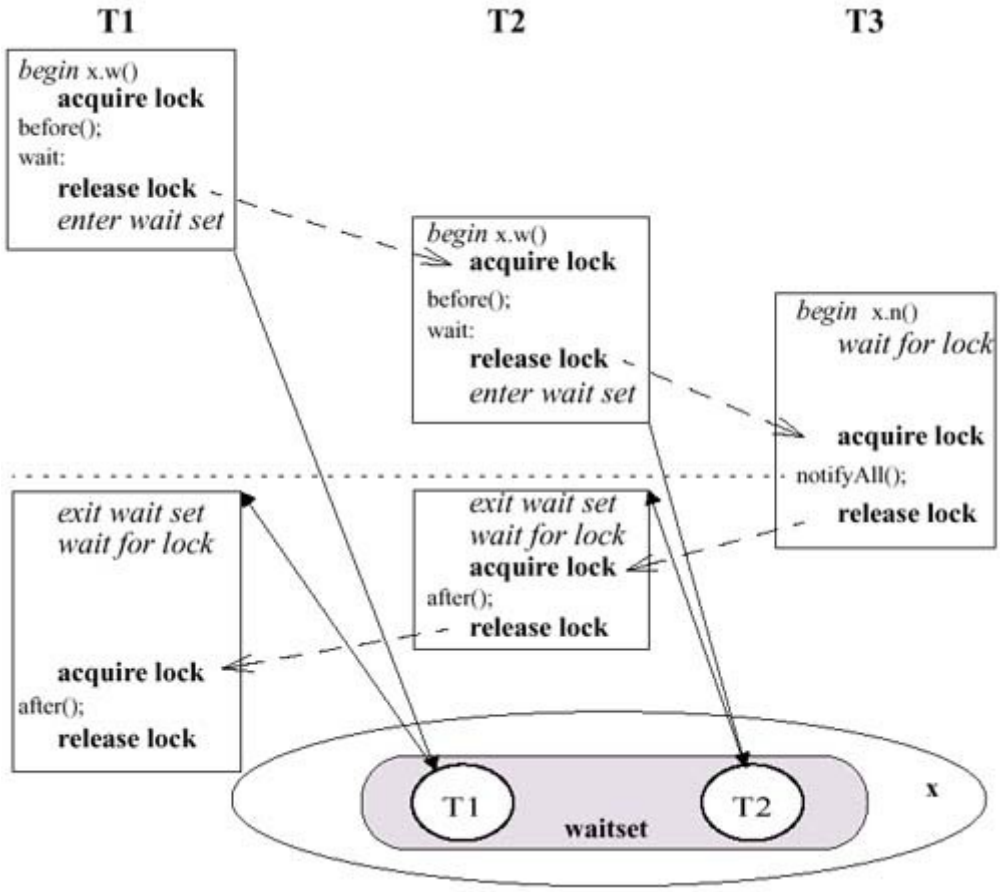
> Trabajando de esta forma se evita la espera activa, con la cual un hilo continua consumiendo CPU mientras espera que se de la condición que necesita para proceder. La espera activa NUNCA es recomendable,

- cuando un hilo ejecutando un método sincronizado en el objeto monitor realiza alguna acción que puede haber cambiado el estado del objeto (por ejemplo modificando alguna variable), envía el mensaje notify()/notifyAll() al objeto monitor, para liberar aleatoriamente a alguno de los hilos que estan esperando en su conjunto de espera (o todos si se hizo notifyAll()), para que puedan proseguir en caso que se cumplan las condiciones.

El ejemplo siguiente ilustra la mecánica de monitores (tomado de "Concurrent Programming in Java- Second Edition –Design Principles and Patterns", Doug Lea) .

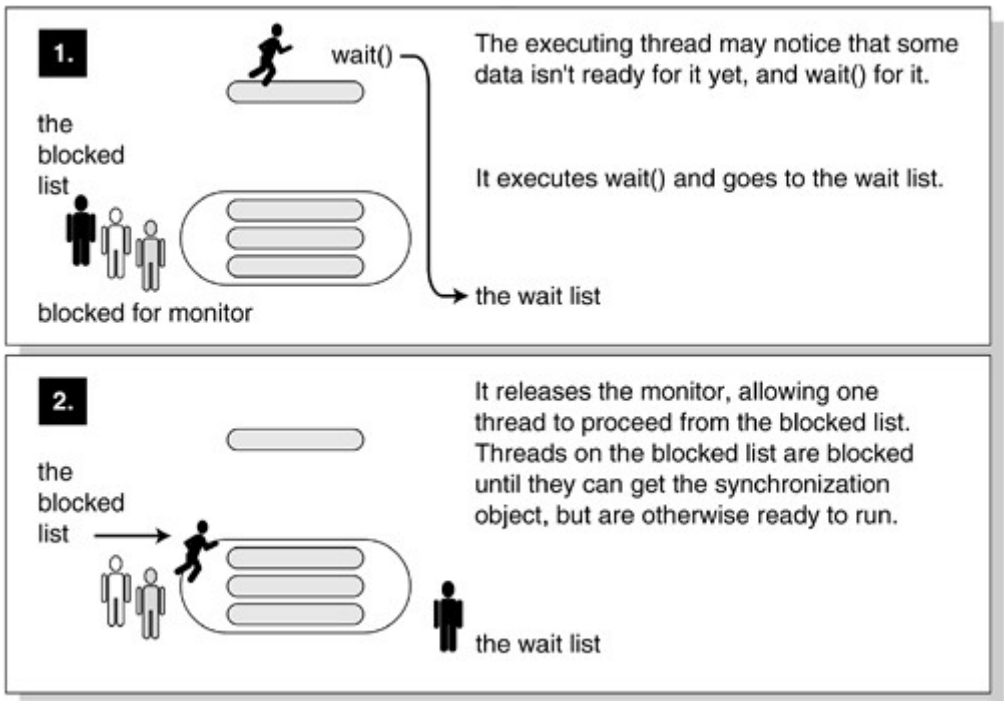
```
class MonX {
     synchronized void w() throws InterruptedException {
          this.before();
          this.wait();
          this.after();
     }

     synchronized void n() {
          this.notifyAll();
     }

     void before() {}

     void after() {}
}
```

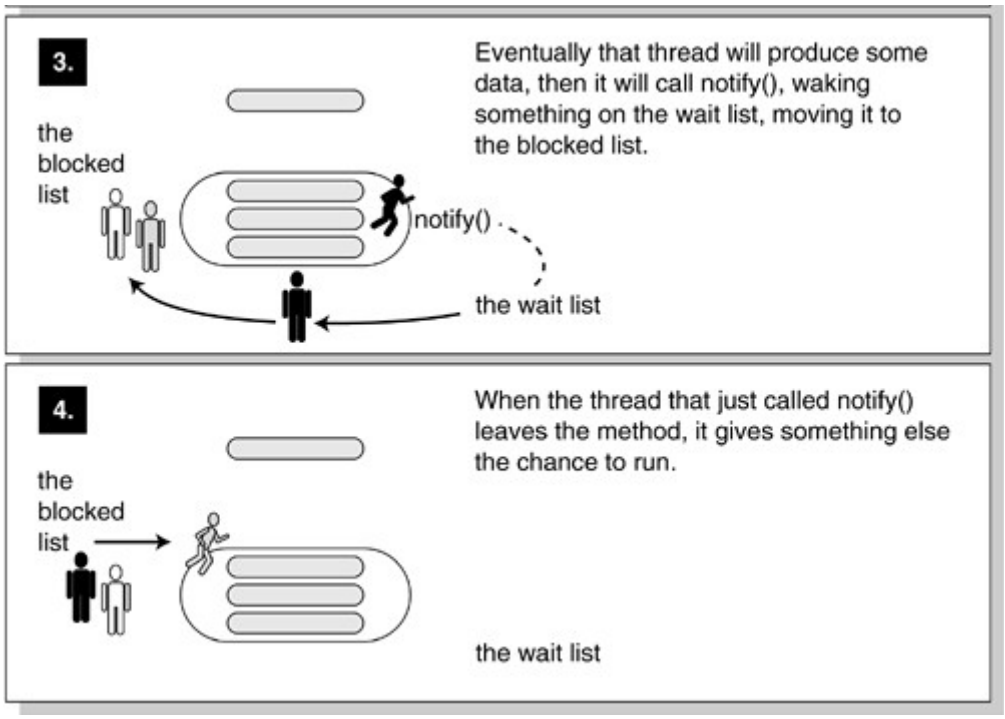
Y consideremos 3 hilos que invocan los métodos del monitor MonX sobre una instancia compartida x, (x instancia de MonX). Podria darse la siguiente ejecución, en donde T2 reasume su ejecución antes que T1, o darse a la inversa, que T1 reasuma su ejecución antes que T2.



Esperas guardadas (dependen de una condicion)

En general las invocaciones de wait() se hacen dentro de un bucle while. Esto es porque cuando una acción es reasumida, la tarea en espera NO SABE si la condición por la que estaba esperando ahora se cumple, y por ello debe volver a verificar, para mantener la propiedad de seguridad. Es una buena práctica que este estilo se utilice aun cuando haya una única instancia que pueda esperar por la condición. Hay que tener en cuenta que un hilo que queda en el conjunto de espera de un monitor por efecto de una operación wait, LIBERA EL LOCK, luego permite que otros hilos comiencen a ejecutar cualquier método sincronizado del objeto. Ademas no hay que olvidar que los métodos no sincronizados pueden ejecutarse sin restricciones.


1.
wait()
The executing thread may notice that some data isn't ready for it yet, and wait() for it.
the blocked list
It executes wait() and goes to the wait list.
the wait list
blocked for monitor
2.
It releases the monitor, allowing one thread to proceed from the blocked list. Threads on the blocked list are blocked until they can get the synchronization object, but are otherwise ready to run.
the blocked list
the wait list


3.
Eventually that thread will produce some data, then it will call notify(), waking something on the wait list, moving it to the blocked list.
the blocked list
notify()
the wait list
4.
When the thread that just called notify() leaves the method, it gives something else the chance to run.
the blocked list
the wait list

Las figuras anteriroes muestran gráficamente los pasos que se producen por el uso de wait y notify. El hilo que esta bloqueado en el conjunto de espera del monitor, esperando por una condicion, cuando sale del estado bloqueado por efecto de un notify o notifyAll, debe volver a adquirir tiempo de CPU y el lock del monitor, por lo que puede quedar bloqueado en la lista de espera de ese lock (parte 4). Es decir el hilo que se libera se ejecutará en algun momento futuro cuando logre readquirir el acceso exclusivo al monitor.
Además el hilo que hace el notify continúa manteniendo el control exclusivo del monitor, por lo que puede continuar con su ejecución.

> **ATENCION!!** Si un hilo ejecutando la acción "accionMon-1" en un monitor Mon-1 llama a una acción "accionMon-2" en otro monitor Mon-2 y en esa accion espera (o sea hace un wait), solo libera la exclusión en el segundo monitor (Mon-2), mientras que retiene el control exclusivo, o sea el lock sobre el primero (Mon-1.9)

Vamos a trabajar sobre el siguiente problema:

> ***En una tienda de mascotas están teniendo problemas para tener a todos sus hámster felices. Los hámster comparten una jaula en la que hay un plato con comida y una rueda para hacer ejercicio. Todos los hámster quieren inicialmente comer del plato y, después correr en la rueda. Pero se encuentran con el inconveniente de que solo tres de ellos pueden comer del plato al mismo tiempo y solo uno puede correr en la rueda.***

Problema: Solución 1 con monitores

Los hamster serán runnables. El plato y la rueda son recursos compartidos por los hamster. Se propone que el plato sea un monitor, ya que puede ser compartido por varios hamster y la rueda tenga sus métodos sincronizados. No es necesario trabajar con la rueda como monitor (o sea con el patrón synchronized, wait, notify, y esperas guardadas) porque el uso de la rueda es exclusivo, y por eso es suficiente con que el método sea sincronizado.
La variable "comiendo" se utiliza para la espera guardada del hilo en ejecución cuando, a pesar de tener el lock del monitor no se dan las condiciones para seguir, en este caso que haya lugar en el plato. El hilo que hace el wait queda en el conjunto de espera del plato hasta que un hilo hamster que termine de comer notifique que se ha producido un cambio en el estado del plato (hay un lugar libre) y libere a alguno de los que están esperando. Los sleep se utilizan para simular la accion de comer/rodar.

```
public class Rueda {
     public synchronized void rodar(String nombre){
          System.out.println (nombre + " empieza a rodar" );
          try {
               Thread.sleep((long) Math.random()*1500);
          } catch (InterruptedException ex){ . . .}
     }
}


public class Plato {
     private int cantidad;
     private int comiendo;


     public Plato(int maximo){
          cantidad = maximo;
          comiendo = 0;
     }



     public synchronized void empezarAComer(String nombre){
          try {
               while (comiendo >= 3){
                    System.out.println(nombre + "debe
                                        esperar para comer");
                    this.wait();
               }
          } catch (InterruptedException ex){. . .}
          System.out.println( nombre + " empieza a comer");
          comiendo ++;
     }



     public synchronized void terminarDeComer(String nombre){
          System.out.println( nombre + " termino de comer");
          comiendo --;
          this.notify();
     }
}
```

```
public class HamsterMonitor implements Runnable {
     private Plato comida;
     private Rueda ejercicio;
     private String miNombre;

     public HamsterMonitor(Plato laComida, Rueda elEjercicio,
                                          String nombre){
          comida = laComida;
          ejercicio = elEjercicio;
          miNombre = nombre;
     }


     public void run(){
          while (true) {
               comida.empezarAComer(miNombre);
               try {
                    Thread.sleep((long) Math.random()*1500);
               } catch (InterruptedException ex){...}

               comida.terminarDeComer(miNombre);
               ejercicio.rodar(miNombre);
          }
     }
}
```

Pregunta 1:
¿Como se daría la ejecución si el método run() del hamster y el método "comer" fueran los siguientes? ¿Se lograría el efecto esperado? ¿por qué si o por qué no?

```
    public void run(){

        while (true) {
             comida.comer(miNombre);
             ejercicio.rodar(miNombre);

             try {
                  Thread.sleep((long) Math.random()*3500);
             } catch (InterruptedException ex) {...}

        }

    }
```

```
public synchronized void comer(String nombre){

     try {
          while (comiendo >= cantidad){
               System.out.println( nombre + " debe
                                   esperar para comer");
               this.wait();
          }
     } catch (InterruptedException ex)  {...}

     System.out.println( nombre + " empieza a comer");
     comiendo++;

     try {
          Thread.sleep((long) Math.random()*1500);
     } catch (InterruptedException ex){...}

     System.out.println( nombre + " termino de comer");
     comiendo--;
     this.notify();
}
```

Pregunta 2:
¿Cómo resolveria el problema si se le pidiera considerar como recurso compartido un objeto Jaula que sincronice el uso del plato y de la rueda?

Problema del Barbero Dormilón
¿Cómo utilizaria el mecanismo de espera y notificación (wait() y notify()) para resolver el problema del BarberoDormilón?

- en el caso básico sin sala de espera
- considerando una sala de espera con k sillas
- considerando que hay b barberos que atienden en la barberia en el mismo horario

Bibliografia

• **Concurrent Programming in JAVA: design principles and patterns** - Doug Lea - Addison Wesley – 2da edición.
• **Thinking in JAVA** - Bruce Eckel - President, MindView Inc. - 2008

• **Java Concurrency in Practice** - Brian Goetz et all - Addison Wesley 2006.
• **Documentacion Oracle**: http://docs.oracle.com/javase/tutorial/essential/concurrency/ y el paquete java.util.concurrent.
• **Orientación a Objetos con Java y UML** - Carlos Fontella - nueva libreria nl 2004