FACULTAD DE INFORMATICA
DEPARTAMENTO DE PROGRAMACION
CÁTEDRA Programación Concurrente + Laboratorio Prog
2015

Programación Concurrente – 2do. C 2015
Programación Concurrente Multihilos - Java
Parte II

En un escenario concurrente los hilos pueden cooperar o competir por los recursos. Los sistemas interactúan a través de la comunicación. Los hilos para comunicarse utilizan estrategias de acceso a memoria compartida, y facilidades de sincronización basadas en memoria, tales como locks y mecanismos de espera y notificación. Estas construcciones soportan comunicación medianamente eficiente pero, al costo de mayor complejidad y mayor potencial para errores de programación.

Al trabajar con concurrencia se define una relación abstracta entre hilos y memoria principal. Cada hilo es definido para tener una memoria de trabajo en la cual almacena valores. Se garantizan unas pocas propiedades respect a las interacciones de secuencias de instrucciones que corresponden a métodos y celdas de memoria. La mayoría de las reglas se dan en términos de cuando los valores deben ser transferidos entre la memoria principal y la memoria de trabajo por hilo. Se consideran 3 cuestiones entrelazadas:

- *Atomicidad:* ¿qué intrucciones deben tener efectos indivisibles? Las lecturas y escrituras de celdas de memoria que representan campos, es decir variables de instancia y de clase, elementos de arreglos, pero no se cinsideran variables locales de los métodos.
- *Visibilidad:* baja qué condiciones los efectos de un hilo son visibles a otros. Los efectos de interés son escrituras, en cuanto son vistas a través de las lecturas de esos campos.
- *Ordenamiento:* bajo que condiciones los efectos de las operaciones peden aparecer fuera de ordena un hilo dado. Las principales cuestiones de orden relacionadas con lecturas y escrituras asociadas con secuencias de asignación.

Al utilizar la sincronización de forma apropiada se logra que estas propiedades se vean simples. Todos los cambios realizados en un método o bloque sincronizado son atómicos y visibles con respecto a otros métodos y bloques sincronizados que emplean el mismo lock. Además el procesamiento de métodos o bloques sincronizados dentro de un hilo dado están en el orden especificado en el programa.

Mecanismos de Sincronización.

Existen varias posibilidades para lograr la sincronizacion cooperativa y competitiva entre hilos. Llamamos ***sincronizador*** a cualquier objeto que controla el flujo de control de hilos basado en su estado. Todos los sincronizadores comparten ciertas propiedades estructurales: encapsulan el estado que determina si a los hilos que arrivan al sincronizador se les debe permitir pasar o hay que forzarlos a esperar, proveen métodos para manipular ese estado, y proveen métodos para esperar eficientemente para que el sincronizador entre en el estado deseado.

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.*

Vamos a proponer soluciones para el problema utilizando los distintos mecanismos de sincronización. Los objetos activos del problema son los hamster, por lo que serán los hilos. ...

Monitores

Es una estructura de datos con todos los métodos sincronizados y los atributos privados (como corresponde a un buen trabajo orientado a objetos). Si bien en algunos lenguajes existe la clase Monitor, Java logra la semántica de monitor por la utilización de bloqueos intrínsecos, con el patrón *synchronized- wait – notify.*
Todo objeto en Java tiene un lock implícito, un conjunto de espera y responde a los mensajes *wait(), wait(timeout), notify()* y *notifyAll().*
A lo explicado anteriormente (bloqueo implícito), se agrega el comportamiento siguiente:

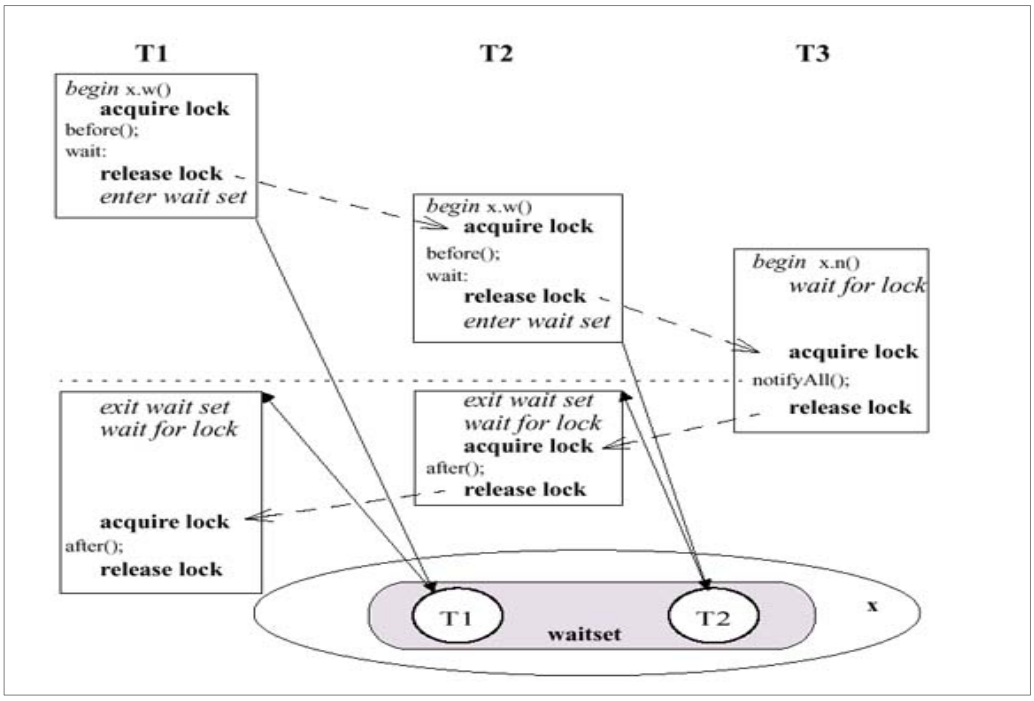
Considere el objeto "monitor",

- 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, puede hacer una *espera activa*, consumiendo CPU (no recomendable), o hacer uso del mensaje *wait()/wait(timeout)*, con lo que el hilo queda bloqueado en el conjunto de espera del objeto, 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 objeto monitor).
- 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)*

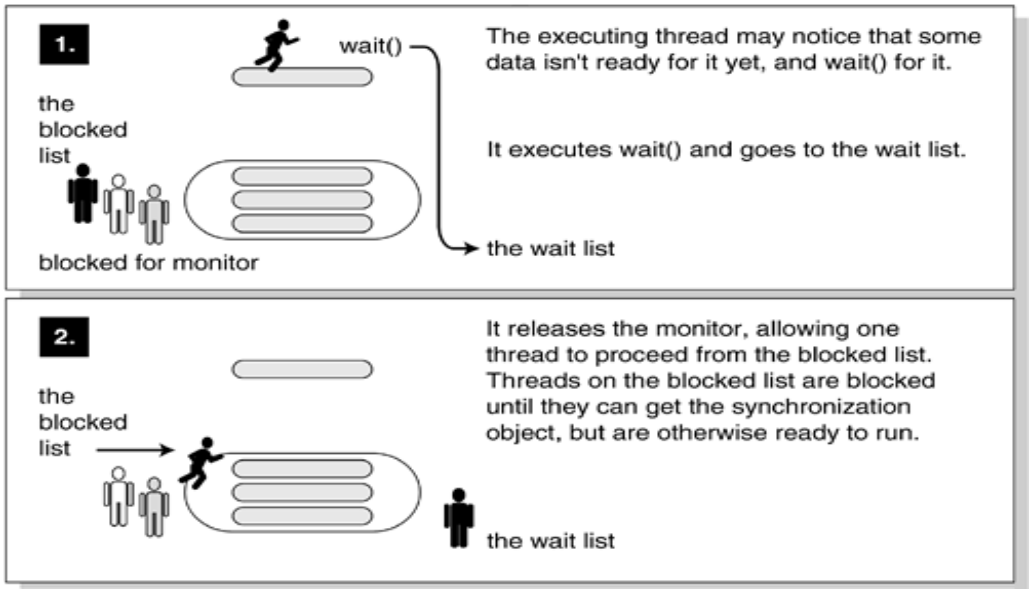
```
class MonX {
            synchronized void w() throws InterruptedException {
                        this.before();
                        this.wait();
                        this.after();
            }

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

            void before() {}
            void after() {}
}
```

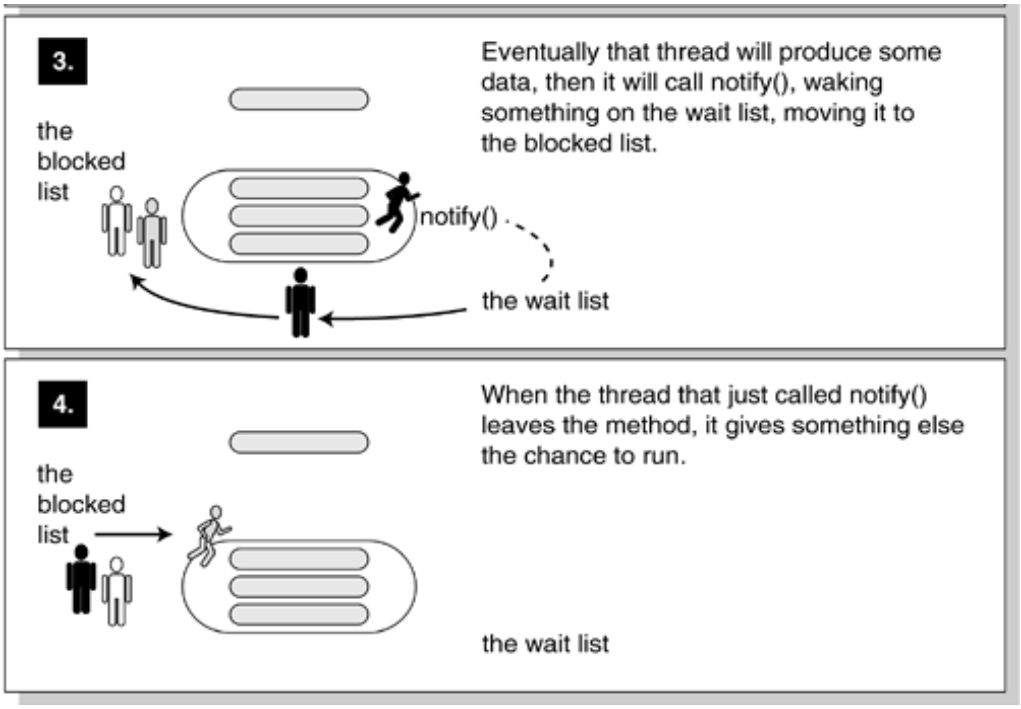
Y consideremos 3 hilos que invocan los métodos del monitor MonX sobre una instancia compartida x, (x instancia de MonX). Podria darse la siguiente ejecución, en donde T2 reasume su ejecución antes que T1, o darse a la inversa, que T1 reasuma su ejecución antes que T2.



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.





(Imágenes tomadas de material de la Universidad de Cantabria).

Problema: Solución 1

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);
   }
  }
}
```

Problema: Solución 2

Otra posibilidad es considerar un método comer y sincronizar por bloques. La idea es la misma, utilizar la variable "comiendo" como guarda para producir la espera de los hilos.

```
public void run(){

   while (true) {
       comida.comer(miNombre);
       ejercicio.rodar(miNombre);
       try {
          Thread.sleep((long) Math.random()*3500);
       } catch (InterruptedException ex)
       {...}

   }
}
```

```
public void comer(String nombre){
   synchronized (this){
       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()*3500);
   } catch (InterruptedException ex)  {...}

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

¿Como se daría la ejecución si el método "comer" fuera el siguiente? ¿Se lograría el efecto esperado? ¿por qué si o por qué no?

```
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();
}
```

Semáforos

Los semáforos son construcciones de control de concurrencia. 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. Un semáforo maneja un conjunto de permisos virtuales, cuyo número inicial es dado al crear el semáforo. Las operaciones básicas para trabajar con semáforos son:

- wait / acquire (Java)
- signal / release (Java)

Las actividades pueden adquirir (*acquire*) permiso (siempre que haya alguno disponible) y liberarlo cuando terminan. Si ningun permiso esta libre entonces la operacion *acquire* se bloquea hasta que un permiso este libre, sea interrumpida o expire su tiempo. El método *release* libera un permiso, o sea lo retorna al semáforo.
Una version mas simple es un semáforo binario, con un contador inicial de 1 que puede ser usado como un mutex.

La clase Semaphore de Java también ofrece la operación tryAcquire que permite que el hilo interesado en adquirir un permiso pueda verificar si hay una disponible, y asi evitar bloquearse. Esto es de utilidad cuando el hilo puede seguir con otra tarea mientras espera que se libere algun permiso. Es una diferencia fundamental con "synchronized", dado que en el caso de métodos y bloques sincronizados el hilo no tiene forme de evitar el bloqueo.

Problema: solución 3

Las tareas que serán ejecutadas por los hilos comparten el objeto instancia de RuedaSem y el objeto instancia de PlatoSem. Cada uno de ellos utilizará semaforos para lograr el control de acceso. Notese que para los runnable es básicamente el mismo código que para la segunda solución con monitor.

```
public class HamsterSemaforo implements Runnable {

    private PlatoSem comida;
    private RuedaSem ejercicio;
    private String miNombre;
```

```
    public HamsterSemaforo(PlatoSem laComida,
                           RuedaSem elEjercicio,
                           String nombre){
        comida = laComida;
        ejercicio = elEjercicio;
        miNombre = nombre;
    }

    public void run(){

        while (true) {

            comida.comer(miNombre);
            ejercicio.rodar(miNombre);
             try {
            Thread.sleep((long) Math.random()*4500);
            } catch (InterruptedException ex){...}
        }
    }
}
```

En la clase PlatoSem se utiliza un semáforo general, con *maximo* permisos (en este caso *maximo* es 3) para controlar el acceso de los hamster al plato. No es exclusión mútua, ya que podrá haber mas de un hilo ejecutando el método comer, pero no podrá haber mas de *maximo* hilos ejecutando el trozo de código entre el *acquire* y el *release*

```
public class PlatoSem {

    private Semaphore comiendo;

    public PlatoSem(int maximo){
        comiendo = new Semaphore(maximo);
    }


    public void comer(String nombre){

        try {
            comiendo.acquire();

            System.out.println( nombre + " empieza a comer");
            Thread.sleep((long) Math.random()*3500);
        } catch (InterruptedException ex){}

        System.out.println( nombre + " ------ termino de comer");
        comiendo.release();

    }
}
```

En la clase RuedaSem el semáforo se uiliza para lograr la exclusión mutua en el acceso al uso de la rueda. El hilo que ejecuta el método rodar, al hacer el *semRueda.acquire* puede tener éxito y continuar utilizando la rueda, o puede quedar bloqueado en espera del permiso (cuando otro hilo lo haya adquirido y aún no lo haya liberado). Como es un semáforo inicializado en 1, actuará como semáforo binario, y en esta situación es el mismo hilo que adquiere el permiso el que debe liberarlo.

```
public class RuedaSem {
    private Semaphore semRueda;

    public RuedaSem(){
        semRueda = new Semaphore(1);
    }

    public void rodar(String nombre){
        try {
            semRueda.acquire();
            System.out.println (nombre + " empieza a rodar" );

            Thread.sleep((long) Math.random()*4500);
        } catch (InterruptedException ex)
        {...}



    }
}
```

Llaves de exclusión mutua - Locks

Es un tipo de datos que debe implementar una interfaz con 3 métodos: *adquirir* la exclusión mutua sobre el objeto, *liberar* la exclusión sobre el objeto, e *intentar* adquirir la exclusión luego del tiempo t. En Java se utiliza la interfaz *LOCK* y una de sus implementaciones *ReentrantLock*
El "lock" o cerrojo reentrante de Java es fácil de usar pero tiene muchas limitaciones. Por eso el paquete java.util.concurrent.locks incluye una serie de utilidades relacionadas con lock. La interfaz más básica de éstas es Lock.

Los objetos cuyas clases implementan la interfaz Lock funcionan de manera muy similar a los locks implícitos que se utilizan en código sincronizado, de manera que sólo un hilo puede poseer el Lock al mismo tiempo. La ventaja de trabajar con locks explícitos, es que posibilitan rechazar un intento de adquirir el cerrojo. El método *tryLock()* rechaza la entrega del lock si éste no está disponible inmediatamente, o bien tras un tiempo máximo de espera, si se especifica así. El método *lockInterruptibly* rechaza dar el lock si otro hilo envía una interrupción antes de que el lock haya sido adquirido.

La clase ReentrantLock implementa la interfaz Lock permitiendo que un hilo que requiera un lock que ya tiene pueda volver a entrar a su sección exclusiva, por ser reentrante. Para adquirir y liberar el lock se utilizan los métodos *lock(), tryLock(), tryLock(t)* y *unlock()* indicados en la interfaz. Entonces …

```
unaVariable.lock();
try {
      // seccion crítica que requiere exclusion mutua
} finally {
     unaVariable.unlock();
}
```

Se sugiere liberar el cerrojo (unlock()) en un bloque finally para asegurar que se hará aún en presencia de excepciones

Los objetos Lock también dan soporte a un mecanismo de wait/notify a través de *objetos Condition.*

Variables de condición

La programación concurrente a menudo envuelve esperar para que ocurra algo. Puede ser necesario esperar que una cola no este vacía para remover un elemento, o esperar que haya espacio disponible para agregar algo a un buffer. Este tipo de situaciones es lo que atienden las variables de condición.

- Una variable de condición se asocia a un lock, y un hilo debe mantener el lock para poder esperar sobre

la condición.

- Obtenido el lock se chequea la condición, si es true el hilo continua con lo que debe hacer y libera el lock, si es false el hilo llama a *await()* sobre la condición, que atómicamente libera el lock y bloquea **sobre la condición**.
- Cuando otro hilo llama a *signal()* o *signalAll()* indicando que hubo cambios sobre la condición correspondiente el hilo que quedo en espera en un *await()* se desbloquea y vuelve a tomar el lock cuando sea su turno de CPU.
- Un mismo lock puede tener asociadas mas de 1 variable de condición

Veamos como resolver el problema de los hamster usando locks explícitos.

Problema: solución 4
Se muestran las clases RuedaLock y PlatoLock que son las que sufren mayores cambios respecto a la solución con monitores.
La clase RuedaLock utiliza un reentrantLock llave para lograr la exclusión mutua a la rueda. No es necesario en este caso trabajar con variables de ocndición, ya que la única condición es la exclusión mutua.

```
public class RuedaLock {

    private Lock llave = new ReentrantLock(true);

    public void rodar(String nombre){

        try {
            llave.lock();
            System.out.println (nombre + " empieza a rodar" );
            Thread.sleep((long) Math.random()*2500);
        } catch (InterruptedException ex)
        {  ...}
        finally {llave.unlock();}
    }
}
```

La clase PlatoLock utiliza un reentrant lock *accesoComida,* junto con 1 variable de condición *hayLugar,* que en conjunto controlarán el acceso al plato de comida por no mas de un *maximo* de hilos. Cuando un hamster quiere ingresar a comer, y obtiene el lock, es decir obtiene exclusion mutua sobre la sección crítica guardada por el lock, debe ademas verificar que se dan las condiciones para ingresar a comer, es decir que hayLugar en el plato. Si las condiciones no se dan, entonces el hilo debe esperar, y liberando el lock para permitir que otros hilos puedan ejecutar trozao de codigo cuidados por el mismo lock. El hilo que espera lo hace en la cola de espera de la variable de condición asociada con el lock. Cuando un hamster termina de comer libera a uno o todos los hilos que estan esperando. Los hilos liberados deben volver a verificar la condición. (Funciona de la misma forma que el *wait* y *notify* de monitores)

```
public class PlatoLock {

    private Lock accesoComida;
    private Condition hayLugar;
    private int cantidad;
    private int comiendo;

    public PlatoLock(int maximo){
        accesoComida = new ReentrantLock(true);
        hayLugar = accesoComida.newCondition();
        comiendo = 0;
        cantidad = maximo;
    }
```

```
    public  void empezarAComer(String nombre){

        try {
            accesoComida.lock();
            while (comiendo >= cantidad){
                System.out.println( nombre + " debe esperar para comer");
                hayLugar.await();
            }
            System.out.println( nombre + " empieza a comer");
            comiendo ++;
        } catch (InterruptedException ex){}
        finally {
            accesoComida.unlock();
        }
    }

    public void terminarDeComer(String nombre){

            accesoComida.lock();
            System.out.println( nombre + " ------ termino de comer");
            comiendo --;
            hayLugar.signalAll();
            accesoComida.unlock();
    }
}
```

Continuará ...próximamente

Bibliografia

- Doug Lea, *Concurrent Programming in Java – Second Edition – Design Principles and Patterns,* Addison Wesley 2000.
- Brian Goetz, *Java Concurrency in Practice*, Addison Wesley 2006
- Kathy Sierra, Bert Bates, *Head First Java*, O'Reilly.
- docs.oracle.com (Package java.util.concurrent)