Los métodos Wait y Notify

A veces nos interesa que un hilo se quede bloqueado a la espera de que ocurra algún evento, como la llegada de un dato para tratar o que el usuario termine de escribir algo en una interface de usuario. Todos los objetos java tienen el método **wait()** que deja bloqueado al hilo que lo llama y el método **notify()**, que desbloquea a los hilos bloqueados por **wait()**. Vamos a ver cómo usarlo en un modelo productor/consumidor.

Bloquear un hilo

Antes de nada, que quede claro que las llamdas a **wait()** lanzan excepciones que hay que capturar. Todas las llamadas que pongamos aquí debería estar en un bloque **try-catch**, así

```
try
{
  // llamada a wait()
}
catch (Exception e)
{
  ....
}
```

pero para no liar mucho el código , no vamos a poner todo esto cada vez. Cuando escribamos una aplicación Java completa habra que ponerlo

Vamos ahora a lo que vamos...

Para que un hilo se bloquee basta con que llame al método **wait()** de cualquier objeto. Sin embargo, es necesario que dicho hilo haya marcado ese objeto como **ocupado** por medio de un **synchronized**. Si no se hace así, saltará una excepción de que "**el hilo no es propietario del monitor**" o algo así.

Imaginemos que nuestro hilo quiere retirar datos de una **lista** y si no hay datos, quiere esperar a que los haya. El hilo puede hacer algo como esto

```
synchronized(lista);
{
  if (lista.size()==0)
    lista.wait();

     dato = lista.get(0);
  lista.remove(0);
}
```

En primer lugar hemos hecho el **synchronized(lista)** para "apropiarnos" del objeto lista. Luego, si no hay datos, hacemos el **lista.wait()**. Una vez que nos metemos en el **wait()**, el objeto lista queda marcado como "desocupado", de forma que otros hilos pueden usarlo. Cuando despertemos y salgamos del **wait()**, volverá a marcarse como "ocupado."

Nuestro hilo se desbloquerá y saldrá del **wait()** cuando alguien llame a **lista.notify()**. Si el hilo que mete datos en la **lista** llama luego a **lista.notify()**, cuando salgamos del **wait()** tendremos datos disponibles en la lista, así que únicamente tenemos que leerlos (y borrarlos para no volver a

tratarlos la siguiente vez). Existe otra posibilidad de que el hilo se salga del **wait()** sin que haya datos disponibles, pero la veremos más adelante.

Notificar a los hilos que están en espera

Hemos dicho que el hilo que mete datos en la lista tiene que llamar a **lista.notify()**. Para esto también es necesario apropiarnos del objeto lista con un **synchronized**. El código del hilo que mete datos en la lista quedará así

```
synchronized(lista)
{
  lista.add(dato);
  lista.notify();
}
```

Listo, una vez que hagamos esto, el hilo que estaba bloqueado en el **wait()** despertará, saldrá del **wait()** y seguirá su código leyendo el primer dato de la lista.

wait() y notify() como cola de espera

wait() y **notify()** *funcionan como una lista de espera. Si varios hilos van llamando a **wait()** quedan bloqueados y en una lista de espera, de forma que el primero que llamó a **wait()** es el primero de la lista y el último es el útlimo.*

*Cada llamada a **notify()** despierta al primer hilo* en *la lista de espera, pero no al resto*, que siguen dormidos. Necesitamos por tanto hacer *tantos **notify()** como hilos hayan hecho **wait()** para ir despertándolos a todos de uno en uno*.

Si hacemos *varios **notify()** antes de que haya hilos en espera, quedan marcados* todos esos **notify()**, de forma que *los siguientes hilos que hagan **wait()** no se quedaran bloqueados*.

En resumen, **wait()** y **notify()** funcionan como un contador. Cada **wait()** mira el contador y si es cero o menos se queda bloqueado. Cuando se desbloquea decrementa el contador. Cada notify() incrementa el contador y si se hace 0 o positivo, despierta al primer hilo de la cola.

Un símil para entenderlo mejor. Una mesa en la que hay gente que pone caramelos y gente que los recoge. La gente son los hilos. Los que van a coger caramelos (hacen **wait()**) se ponen en una cola delante de la mesa, cogen un caramelo y se van. Si no hay caramelos, esperan que los haya y forman una cola. Otras personas ponen un caramelo en la mesa (hacen **notify()**). El número de caramelos en la mesa es el contador que mencionabamos.

Modelo Productor/Consumidor

Nuevamente y como comentamos en sincronizar hilos, es buena costumbre de orientación a objetos "ocultar" el tema de la sincronización a los hilos, de forma que no dependamos de que el programador se acuerde de implemetar su hilo correctamente (llamada a **synchronized** y llamada a **wait()** y **notify()**).

Para ello, es práctica habitual meter la lista de datos dentro de una clase y poner dos métodos **synchronized** para añadir y recoger datos, con el **wait()** y el **notify()** dentro.

El código para esta clase que hace todo esto puede ser así

```
public class MiListaSincronizada
{
  private LinkedList lista = new LinkedList();

  public

synchronized void addDato(Object dato)
  {
    lista.add(dato);
    lista.notify();
  }

  public

synchronized Object getDato()
  {
    if (lista.size()==0)
      wait();
    Object dato = lista.get(0);
    lista.remove(0);
    return dato;
  }
}
```

Listo, nuestros hilos ya no deben preocuparse de nada. El hilo que espera por los datos hace esto

```
Object dato = listaSincronizada.getDato();
```

y eso se quedará bloqueado hasta que haya algún dato disponible. Mientras, el hilo que guarda datos sólo tiene que hacer esto otro

```
listaSincronizada.addDato(dato);
```

Interrumpir un hilo

Comentamos antes que es posible que un hilo salga del **wait()** sin necesidad de que nadie haga **notify()**. Esta situación se da cuando se produce algún tipo de interrupción. En el caso de java es fácil provocar una interrupción llamando al método **interrupt()** del hilo.

Por ejemplo, si el **hiloLector** está bloqueado en un **wait()** esperando un dato, podemos interrumpirle con

```
hiloLector.interrupt();
```

El **hiloLector** saldrá del **wait()** y se encontrará con que no hay datos en la lista. Sabrá que alguien le ha interrumpido y hará lo que tenga que hacer en ese caso.

Por ejemplo, imagina que tenemos un hilo **lectorSocket** pendiente de un socket (una conexión con otro programa en otro ordenador a través de red) que lee datos que llegan del otro programa y los mete en la **listaSincronizada**.

Imagina ahora un hilo **lectorDatos** que está leyendo esos datos de la **listaSincronizada** y tratándolos.

¿Qué pasa si el socket se cierra?. Imagina que nuestro programa decide cerrar la conexión (socket) con el otro programa en red porque se han enfadado y ya no piensan hablarse nunca más. Una vez cerrada la conexión, el hilo **lectorSocket** puede interrumpir al hilo **lectorDatos**. Este, al ver que ha salido del **wait()** y que no hay datos disponibles, puede suponer que se ha cerrado la conexión y terminar.

El código del hilo **lectorDatos** puede ser así

```
while (true)
{
  if (listaSincronizada.size() == 0)
    wait();

  // Debemos comprobar que efectivamente hay datos.
  if (listaSincronizada.size() > 0)
  {
    // Hay datos, los tratamos
    Object dato=listaSincronizada.get(0);
    listaSincronizada.remove(0);
    // tratar el dato.
  }
  else
  {
    // No hay, datos se debe haber cerrado la conexion
    // así que nos salimos.
    return;
  }
}
```

y el hilo **lectorSocket**, cuando cierra la conexión, debe hacer

```
socket.close();
lectorDatos.interrupt();
```