Mini Lección 1 - Multiprogramación y sockets en Java

Sitio:	CampusVirtual UVa -
Curso:	SISTEMAS DISTRIBUIDOS - Grado en Ingeniería Informática
Libro:	Mini Lección 1 - Multiprogramación y sockets en Java
Imprimido por:	LLAMAS BELLO, CESAR
Día:	jueves, 17 de febrero de 2023

Tabla de contenidos

Mini Lección 1 - Multiprogramación y sockets en Java

Tabla de contenidos
1. Presentación de la Mini-lección.
2. Multienhebrado en Java
    Mecanismos de creación de hilos de ejecución.
    Pequeña cuestión: para qué solemos usar *hilos*
    2.1. Reloj en Java con dos hilos
    2.2. Estados de un proceso.
3. Sincronización en Java
    3.1. Synchronized
        Métodos synchronized
    3.2. Colas de Strings
    3.3. Coordinación del trabajo entre hilos.
    3.4. Código de ColaStrings.
    3.5. Más código.
4. Conclusión de la Minilección

¿Qué hemos hecho?

¿Qué hacer?:

1. Presentación de la Mini-lección.

La programación de aplicaciones cliente-servidor en Java, con sockets, obedece a un planteamiento muy sencillo que luego se complica tecnológicamente hasta donde no está escrito, por ello es preciso presentar los temas de los que se compone, con cierto orden:

- **Multienhebrado**: Para permitir diversos hilos en ejecución para servidores que se adhieren a la categoría de servidores multihilo;
- **Sincronización**: Para resolver los problemas de sincronización de código que conlleva la compartición de recursos; y

- **Comunicación por sockets:** Para resolver el problema de comunicación en red.

A continuación, lee y completa los ejercicios de multienhebrado. Cuando termines, repasa los conceptos de sincronización y comunicación por sockets que te serán útiles para la próxima Lección de laboratorio.

---

2. Multienhebrado en Java

La unidad de ejecución de Java es el hilo (*thread*), que coincide con nuestra noción de proceso ligero, es decir:

- todos los hilos acceden a los mismos recursos, y
- cada hilo tiene su propia pila, PC, etc.

Los hilos de la MV Java pueden ejecutarse en diversos procesadores o ser simulados en el núcleo de la máquina virtual. En el primer caso se habla de hilos soportados de modo nativo (*native threads*), y en el segundo de hilos Java (*green threads*). Esto se decide en el momento de instalar la máquina virtual en la plataforma.
Por naturaleza, la máquina virtual, ejecuta, de modo transparente al usuario, diversos hilos que dan soporte a la plataforma. Esto es necesario para el buen funcionamiento del gestor de referencias (*reference handler*), el finalizador, el distribuidor de señales (*signal dispatcher*) o los entornos gráficos (AWT,...).

Mecanismos de creación de hilos de ejecución.

Existen dos formas usuales de crear hilos en Java:

- extendiendo la clase "`Thread`", o
- implementando la interfaz "`Runnable`".

Ambas formas se comportan de manera muy similar, pero hay que recordar las diferencias entre clases e interfaces en Java. **Es importante que repases estos conceptos si crees que no puedes diferenciarlo.**

A partir de aquí, los mecanismos básicos de sincronización giran en torno al constructor del lenguaje "`synchronize`", y a los métodos de gestión de "`Threads`": "`wait`", "`notify`" y "`notifyAll`".

Vamos a ver cómo se lleva esto a la práctica con un pequeño ejemplo de reloj en Java con dos hilos. Pero antes una pequeña pregunta para comprobar que comprendes bien para qué se usan los hilos.

Pequeña cuestión: para qué solemos usar *hilos*

Los hilos se van a ejecutar sobre métodos de objetos visibles en el momento de la invocación. En otras palabras, todos los hilos de un proceso pesado comparten el mismo área de objetos. Posiblemente alguna referencia a un objeto existente la tenga un sólo hilo, pero está en el mismo pool de trabajo que funciona como una memoria compartida.

Para el estilo normal de programar hay dos motivaciones principales en el uso de hilos:

1. Cuando hay recursos que se bloquean en métodos que tardan en retornar. Esto ocurre al esperar por mensajes, al leer de un archivo, al esperar un texto por teclado... Con los hilos, podemos dejar un hilo esperando, y seguir procesando por otro lado.

2. Cuando hay tareas que se ***pueden*** hacer a la vez. Lo cual no quiere decir que se hagan a la vez, pero es un estilo de programación que modela muchos procesos informáticos de una forma mucho más realista que el proceso secuencias con un sólo hilo.

Los problemas que trae la programación concurrente son serios, y hay que tener muy claro si realmente sale a cuenta, pues no hay muchos programadores realmente solventes en esta cuestión.

2.1. Reloj en Java con dos hilos

El siguiente es un ejemplo de reloj en Java con dos hilos.

```
/*
*   TimePrinter.java Imprime la hora cada segundo.
*/
package ejemplo.timeprinter;

public class TimePrinter implements Runnable {
    public void run() {
        while (true) {
            System.out.println(new java.util.Date());
            try { Thread.sleep(1000); // milis
            } catch (InterruptedException x) { }
        }
    }

    public static void main(String[ ] args) {
        final Runnable tarea = new TimePrinter();
        new Thread(tarea, "Hilo de TimePrinter").****();
        // resto del programa
    }
}
```

Como puede intuirse, el metodo "`main()`" prepara el lanzamiento de hilos, lo lanza, y prosigue con su tarea. El lanzamiento efectivo se produce en virtud del método de arranque del "`Thread`", que a la postre inicia el método "`run()`" de la clase de tipo "`Runnable`", que es ésta misma. Este método no hace más que ejecutar indefinidamente un bucle, en el que periódicamente se imprime la hora, y se duerme durante 1000 milisegundos debido al método "`sleep`" que se invoca sobre el "`Thread`".

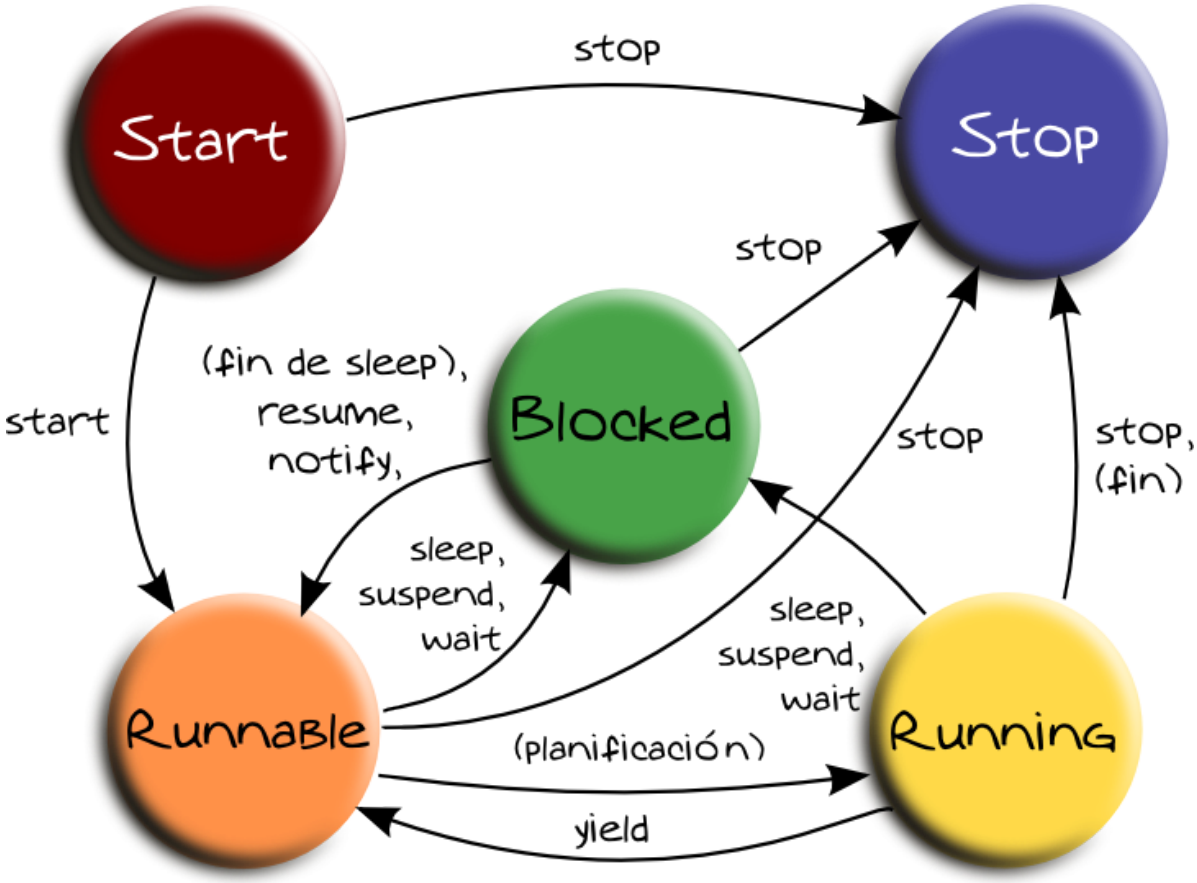
Copie literalmente este fragmento de programa y compílelo en el entorno. Pero antes, busca qué método hay que emplear en lugar de `****` para iniciar la ejecución del thread.

```
wait()
paint()
run()
start()
```

La correcta es precisamente esta última, y aunque la anterior también es cierta, sólo lo es en este caso, y las consecuencias para el multienhebrado son muy distintas.

2.2. Estados de un proceso.

La siguiente figura resume el conjunto de estados que se admiten en un proceso en Java. El círculo "Start" representa el estado de los objetos "Thread" nuevos, y el círculo "Stop" representa el estado de los objetos "Thread" terminados.



En esta figura puede verse el ya habitual triángulo "Runnable-Running-Blocked" de los hilos, con los posibles cambios de estado en razón de los métodos de "Thread".

Cada hilo tiene una prioridad comprendida entre "`MIN PRIORITY`" y "`MAX_PRIORITY`". Esta prioridad se hereda del hilo padre en un principio, y se relaciona con los métodos "`getPriority()`" y "`setPriority()`". El planificador de la MV funciona con desalojo (hilos apropiables o "*preemtive*"), y su política de ejecución es de turno rotatorio (*round-robin*), aunque también se pueden enclavar hilos no desalojables.

Hemos terminado la parte práctica de la minilección. En la próxima lección retomaremos el estado de los hilos, así que es importante que hayas hecho funcionar el código del reloj en Java con dos hilos.

A continuación repasa los conceptos de sincronización y sockets que utilizarás en la próxima lección de laboratorio.

---

3. Sincronización en Java

La necesidad de sincronizar el trabajo es consecuencia inmediata de la existencia de varios hilos de control. Como ya te habrán indicado en la asignatura de Sistemas operativos, la sincronización implica situaciones donde un proceso "espera" por alguna circunstancia que obedece en principio a dos razones:

- La necesidad de coordinar el trabajo para conseguir un objetivo común.
  Por ejemplo, cuando un proceso espera que otro tenga listo un resultado, o cumpla alguna condición necesaria para poder seguir con el trabajo.
- La necesidad de no entorpecerse mutuamente en el acceso a recursos compartidos.
  Que es muy habitual cuando varios procesos necesitan por casualidad de acceder a la actualización o la consulta de algún recursos común.

La primera y la segunda son, en el fondo, dos caras de los mismo. Para que un sistema funcione, hay recursos comunes, y hay tareas que deben coordinar su acceso, pero *la diferencia semántica es importante*: en el primer caso, los procesos esperan "por construcción", para poder coordinarse, mientras que en el segundo caso se trata de imponer mecanismos de seguridad (coactivos) para prohibir situaciones indeseables.

A continuación veremos ambos casos, empezando por el segundo, la "sincronización" coactiva de los hilos en el acceso a recursos comunes, y después la segunda, mecanismos para coordinación del acceso a recursos.

3.1. Synchronized

La sincronización para el acceso a regiones críticas se fundamenta en la palabra clave "synchronized". Todos los objetos en un proceso Java tienen asociado un bloqueo (cerrojo o *lock*), que podrá ser adquirido y liberado mediante el uso de métodos y sentencias "synchronized", lo cual permite un grado muy fino de sincronización, pues se desciende al punto de exclusión mutua sobre un objeto.

El término "*código sincronizado*" describe a cualquier código que esté dentro de un método "synchronized" o bajo el alcance de una sentencia "synchronized". A continuación se introduce la sincronización por métodos, y más tarde se ve la sincronización por sentencia.

Métodos synchronized

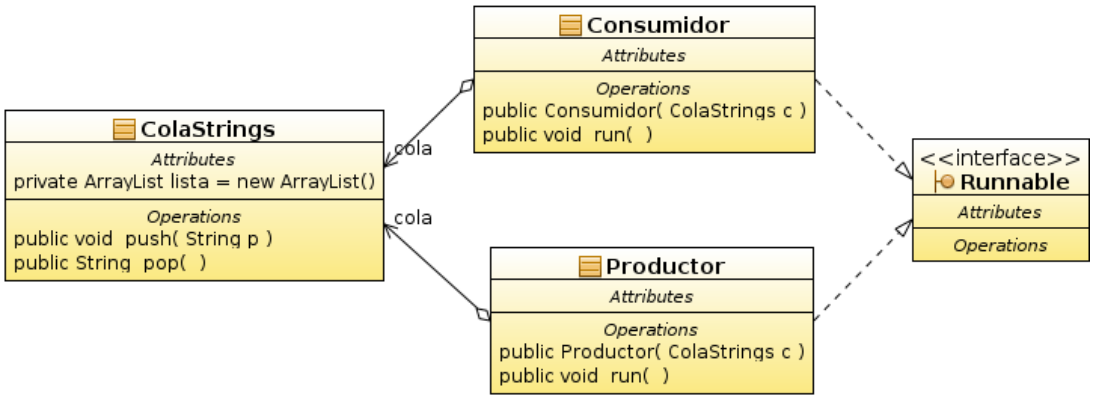
La forma más sencilla de proteger regiones críticas de código es diseñando al efecto métodos que se cualifican como "synchronized". Esto fuerza a que en el caso de que haya dos hilos con intención de ejecutar dicho método sobre el mismo objeto, se ejecuten secuencialmente, de modo que el primero que acceda al método será el que adquiera el bloqueo, que se cederá o liberará cuando finalice dicho método.
El bloqueo se libera tan pronto como termina el método sincronizado, tanto normalmente (sentencia "return") y terminación del método, como excepcionalmente al lanzar una excepción. Este sistema es muy **seguro** pues resulta imposible dejar de liberar un bloqueo por error de programación.
Los métodos estáticos también pueden sincronizarse; en este caso, el bloqueo se relaciona con el objeto "Class" asociado a la clase. *¡Ojo!*, pues este bloqueo no afecta al resto de métodos sincronizados no estáticos, pues es en este caso nos estamos refiriendo a instancias de la clase, con lo que al ser objetos distintos, también son bloqueos distintos.

En esta lección de laboratorio vemos un pequeño ejemplo con colas sincronizadas. Puedes revisar el ejemplo para comprobar que no tienes problemas en entender su funcionamiento y compilar el código. Esto te ayudaría a ir más rápido en la lección de laboratorio. Si no lo deseas, en la siguiente página pasa directamente a la teoría sobre sincronización de hilos.

3.2. Colas de Strings

En este sencillo ejemplo se ve cómo sincronizar dos métodos excluyentes sobre un tipo de dato muy sencillo, ColaStrings. Tiene dos métodos, "push", para introducir cadenas, y "pop", para extraerlas, que son mutuamente excluyentes.

El código se encuentra adyacente a este documento, y contiene dos clases `Runnable` para demostrar con dos hilos el funcionamiento de la clase `ColaStrings`. Cárguelo en su entorno de desarrollo, ejecútelo y vea cómo funciona. El programa principal, queda a disposición del usuario para que teclee cadenas de texto, que entrarán en la cola, y serán mostradas en pantalla. El diagrama de clases de diseño es como sigue:



Esta clase es utilizada desde las clases "Productor" y "Consumidor", que implementan el típico problema ya conocido de sincronización de procesos y recursos. Para refrescar la memoria, repase por encima este documento. Este problema viene al pelo para comentar un conjunto de métodos sobre "Thread" muy interesantes que nos permiten sincronizar el trabajo entre los hilos.

3.3. Coordinación del trabajo entre hilos.

Los métodos que se emplean para comunicar determinadas condiciones de espera entre hilos son:

- `wait()`: deja al hilo esperando hasta que es notificado (típicamente alguna condición que rompe la espera).
- `notify()`: indica a un hilo que despierte; el sistema escoge qué hilo despertar.
- `notifyAll()`: indica a todos los hilos que despierten.

Estos métodos son heredados por todos los objetos, con lo cual pueden invocarse desde cualquier instancia. Dado que puede haber varios hilos esperando sobre el mismo objeto, posiblemente por condiciones diferentes, es normal utilizar siempre "`notifyAll()`" en lugar de "`notify()`", que sólo usamos cuando todos los hilos esperan por la misma condición; sólo un hilo puede beneficiarse del despertar; y esto ocurre en todas las posibles subclases.
El patrón estándar de utilización de "`wait()`" es:

```
synchronized void hacerCuandoCondicion() {
   while (!condicion) wait();
      ... // hacer lo que debe hacerse cuando la condición es cierta.
}
```

donde debemos poner cuidado especial en: la sincronización, y que esté todo en un bucle.

3.4. Código de ColaStrings.

La alternativa que hemos adoptado en nuestro diseño es reunir la sincronización del acceso a la región crítica en los métodos de "`olaStrings`". La otra alternativa, el incluír el código de sincronización dentro de los tipos de proceso "`Productor`" y "`Consumidor`" suele ser muy arriesgada. Entre los inconvenientes de esta última están:

- Los cambios en el modelo de sincronización afectan a dos tipos de proceso "`Productor`" y "`Consumidor`", mientras que si lo concentramos en "`ColaStrings`" se reduce todo al código de una clase.
- Un programador malicioso podría utilizar un `Productor` o un `Consumidor` diseñados para bloquear el sistema o aprovecharse de él.
- Cada objeto de tipo "`ColaStrings`" se convierte en el proveedor del bloqueo para la sincronización.

He aquí el código de "`ColaStrings`" comentado previamente:

```
1  package protocolo.Servidor;
2  import java.util.*;
3
4  public class ColaStrings {
5      private ArrayList<String> lista = new ArrayList<String>();
6
7      public synchronized void push(final String p) {
8          lista.add(p);
9          this.notify(); // hace saber que ha llegado un String
10     }
11
12     public synchronized String pop() {
13         while (lista.size() == 0)
14             try { this.wait(); // espera la llegada de un String
15             } catch (final InterruptedException e) { }
16         return lista.remove(0);
17     }
18
19     public boolean estaVacia() {
20         return lista.isEmpty();
21     }
22 }
```

Un análisis superficial de esta clase nos muestra que dispone de dos métodos "*synchronized*" (sincronizados). El método más complejo parece ser "`pop()`", que se etiqueta así para evitar el que haya dos procesos o más intentando ejecutando el mismo trozo de código. Si no hubiera elementos en la cola, el hilo activo se duerme a sí mismo mediante "`wait()`", evitando que se dispare "`IndexOutOfBoundsException`". Un efecto colateral sorprendente del método `wait()` es que libera el bloqueo sobre el objeto de tipo `ColaStrings`, con lo que así queda libre para que cualquier otro thread invoque `push()` o `pop()`.

Aparentemente, sincronizar el método `push()` parece innecesario, pues no vemos en él una región crítica que nos pueda crear problemas, sin embargo ocurre que para despertar los hilos dormidos en el método `pop()` mediante `notify()` es necesario hacerlo desde el interior de un método o región sincronizada con el mismo objeto de bloqueo! En conclusión, tanto `push()` como `pop()` deben ser *synchronized*.

Por último, podría sorprendernos el que la llamada a `wait()` se encuentre dentro de un bucle (`while`). Realmente no debería ser necesario, pero si quieres saber porqué está ahí, lee el apartado dedicado al método `Object.wait()` en las páginas del manual del API de Java. La razón se encuentra en la posibilidad remota de producir un "*spurious wake-up*", que podría ser fatal para la lógica del programa. A mi modo de ver, esta posibilidad es más bien remota, y es una herencia de que en método `pop()` una vía alternativa en lugar de `notify()` es utilizar `notifyAll()` -otrora más viable- con lo que despertaríamos todos los hilos dormidos, y habría que volver a dormirlos selectivamente.

3.5. Más código.

Este es el código del tipo de proceso `Consumidor`. Obsérvese que la clase implementa `Runnable`.

```
1  package protocolo.Servidor;
2
3  public class Consumidor implements Runnable {
4      private final ColaStrings cola ;
5
6      public Consumidor(ColaStrings c) {
7          this.cola = c;
8      }
9
10     public void run() {
11         String linea;
12         while (true) {
13             linea = cola.pop();
14             System.out.println(">> " + linea); // imprime
15         }
16     }
17 }
```

También implementa `Runnable` el `Productor`

```
1  package protocolo.Servidor;
2
3  import java.io.*;
4
5  public class Productor implements Runnable {
6      private final ColaStrings cola ;
7
8      public Productor(ColaStrings c) {
9          this.cola = c;
10     }
11
12     public void run() {
13         final Reader r1 = new InputStreamReader(System.in);
14         final BufferedReader teclado = new BufferedReader(r1);
```

```
15
16          String linea;
17          try {
18              while ((linea = teclado.readLine()) != null) { // lee teclado
19                  cola.push(linea);
20              }
21          } catch (final IOException x) { x.printStackTrace(System.err); }
22      }
23  }
```

Y finalmente, la clase lanzadora del pequeño programa de prueba

```
1  package protocolo.Servidor;
2
3  public class Main {
4      public static void main(String[] args) {
5          ColaStrings cola = new ColaStrings() ;
6          Runnable genera = new Productor(cola) ;
7          Runnable come = new Consumidor(cola) ;
8
9          new Thread(genera).start();
10         new Thread(come).start();
11     }
12 }
```

---

4. Conclusión de la Minilección

¡¡Has finalizado la Minilección!!

¿Qué hemos hecho?

- Has tomado contacto con el multienhebrado, entiendes la utilidad de los hilos y sus estados.
- Has podido compilar y entender un ejemplo de reloj en Java con dos hilos.
- Tienes a tu disposición un ejemplo completo de sincronización con colas de strings.

¿Qué hacer?:

- Refuerza los conocimientos y prueba a repetir esta minilección.
- Comprueba que eres capaz de compilar y entender el ejemplo de reloj.
- Prueba y trata de comprender el ejemplo de sincronización con colas de strings.
- Añade la información que necesites para que podamos pasar un parámetro al constructor del consumidor, de modo que al imprimir el mensaje que es resultado del consumo del `String` de la cola, pueda aparecer información adicional que identifique al consumidor para...
    - Añadir varios consumidores más!!! Haz que los consumidores esperen una cantidad aleatoria de tiempo de varios segundos, de modo que según vas tecleando en el productor cadenas de texto a toda velocidad, los consumidores las saquen de la cola poco a poco.
    - Pon 3 o cuatro consumidores, por ejemplo.