
TEMA 10: THREADS

10.1.-INTRODUCCIÓN

Los *threads* (*hilos o hebras*) son una aparición relativamente reciente en Informática, pero muy útiles. La idea fundamental es bien sencilla. En la programación tradicional hay un solo flujo de control, motivado fundamentalmente porque la máquina internamente suele tener un solo procesador (una sola "mente" que realiza las instrucciones, una tras otra).

La programación *multithreading* (*multienhebrada o multi-hilo*) permite la ocurrencia simultánea de varios flujos de control. Cada uno de ellos puede programarse independientemente y realizar un trabajo, distinto, idéntico o complementario, a otros flujos paralelos.

Hay miles de ejemplos en los que puede ser útil pensar en varios flujos de ejecución (*threads*): la posibilidad de editar mientras seguimos cargando o salvando un gran fichero, la posibilidad de visualizar una página mientras se están buscando las siguientes, la visualización de varios procesos que ocurren a la vez de forma independiente, etc.

Para definir el concepto que iremos desgranando en este capítulo, diremos que un *thread* es un flujo de ejecución que ocurre independientemente de otros, que puede trabajar sobre datos distintos o compartidos, y que puede en un momento dado pararse, reiniciarse, sincronizarse o esperar a otros.

Realmente no deberíamos llamar a esto multiprocesamiento sino multisubprocesamiento, porque lo que ocurre es que dentro del mismo proceso (compartiendo el mismo entorno volátil) surgen varios hilos de ejecución. El S.O. irá alternando entre procesos y además una vez que llegue a nuestro proceso alternará a su vez sobre cada uno de los hilos, creándose así multisubproceso. Esto tiene la ventaja de un menor tiempo de conmutación en comparación con el alternamiento clásico de procesos en un S.O.

10.2.-CLASE JAVA.LANG.THREAD

Para trabajar con *threads* en Java se utiliza la clase `THREAD`, del paquete `java.lang`. A continuación se presenta una breve descripción de los atributos y métodos más frecuentes de esta clase (esta información más ampliada se encuentra disponible en el API de Java):

Atributos de clase:

- `int MAX_PRIORITY` - Prioridad máxima que puede tener un *thread*.
- `int MIN_PRIORITY` - Prioridad mínima.

- `int NORM_PRIORITY` - La prioridad que se asigna por defecto.

A cada thread Java se le da una prioridad numérica entre `MIN_PRIORITY` y `MAX_PRIORITY` (constantes definidas en la clase `Thread`). En un momento dado, cuando varios threads están listos para ejecutarse, el thread con prioridad superior será el elegido para su ejecución. Sólo cuando el thread para o se suspende por alguna razón, se empezará a ejecutar un thread con prioridad inferior.

Constructores:

- `Thread()`: Crea un thread con nombre "Thread-" + N (siendo n un número secuencial).
- `Thread(String name)`: Crea un thread con el nombre indicado como parámetro.
- `Thread(Runnable target)`: Crea un thread asociado al objeto target. El nombre es "Thread-" + N.
- `Thread(Runnable target, String name)`: Como el anterior, pero asignándole el nombre que se pasa como parámetro.
- `Thread(ThreadGroup group, String name)`: Crea un thread dentro de un grupo de threads que se pasa como primer parámetro, dándole un nombre como segundo parámetro.
- `Thread(ThreadGroup group, Runnable target)`: Como el anterior, pero asociándole un objeto Runnable.
- `Thread(ThreadGroup group, Runnable target, String name)`: Como los anteriores, pasando todos los parámetros posibles: el grupo de hilos al que pertenece, el objeto Runnable asociado y el nombre del Thread.

Métodos:

- `static int activeCount()`: Devuelve el número actual de hilos activos en ese `ThreadGroup`.
- `void checkAccess()`: Comprueba el `SecurityManager` para determinar si el hilo actual tiene permiso para modificar este hilo.

Las clases `Thread` y `ThreadGroup` tienen un método, `checkAccess()`, que llama al método `checkAccess()` del controlador de seguridad actual. El controlador de seguridad decide si permite el acceso basándose en los miembros del grupo de threads involucrado.

Si el acceso no está permitido, el método `checkAccess()` lanza una excepción `SecurityException`. De otro modo el método `checkAccess()` simplemente retorna.

La siguiente lista muestra varios métodos de `ThreadGroup` que llaman a `checkAccess()` antes de realizar la acción del método. Esto se conoce como acceso regulado, esto es, accesos que deben ser aprobados por el controlador de seguridad antes de poder ser completados.

- ThreadGroup(ThreadGroup padre, String nombre)
- setDaemon(boolean isDaemon)
- setMaxPriority(int maxPriority)
- stop()
- suspend()
- resume()
- destroy()

A su vez, la lista que se muestra a continuación es la lista de métodos de la clase Thread que llaman a checkAccess() antes de proceder:

- static Thread currentThread(): Devuelve una referencia al Thread que se está ejecutando en ese momento.
- void destroy(): Produce la salida inmediata del Thread, sin oportunidad de limpieza.
- static void dumpStack(): Imprime un rastro del Thread que se está ejecutando en ese momento.
- static int enumerate(Thread[] tarray): Prepara un array de Threads que consiste en una copia de cada Thread activo en este ThreadGroup y sus subgrupos.
- String getName(): Devuelve el nombre del Thread.
- int getPriority(): Devuelve la prioridad del Thread. Por defecto, todos los hilos tienen la prioridad del hilo que iniciaron. Un hilo puede cambiar su prioridad llamando a setPriority().
- ThreadGroup getThreadGroup(): Devuelve una referencia al ThreadGroup de este Thread.
- void interrupt(): Interrumpe el hilo especificado.
- static boolean interrupted(): Comprueba si el actual Thread ha sido interrumpido.
- boolean isAlive(): Si el Thread especificado está vivo aún, devuelve true. Un Thread está vivo cuando es llamado su método start(), y permanece así hasta que muere.

Este método devuelve true si el thread ha sido arrancado y no ha parado. Así, si el método isAlive() devuelve false sabrás que se trata de un "Nuevo thread" o de un thread "Muerto". Por el contrario si devuelve true sabrás que el thread es "Ejecutable" o "No Ejecutable". No se puede diferenciar entre un "Nuevo thread" y un thread "Muerto", como tampoco se puede hacer entre un thread "Ejecutable" y otro "No Ejecutable"

- boolean isDaemon(): Si el Thread especificado es un demonio, devuelve true. Un Thread es un demonio cuando está diseñado para ejecutarse en background independientemente de cualquier interfaz de usuario. Cuando se hayan terminado todos los Threads de interfaz de usuario, y los únicos ejecutándose sean demonios, también termina la JVM.
- boolean isInterrupted(): Si este Thread ha sido interrumpido, devuelve true.
- void join(): Mezcla dos hilos que esperan a un tercero para morir.
- void join(long millis): Igual que el anterior, pero se limita la espera a los milisegundos indicados como parámetro.
- void join(long millis, int nanos): Al igual que el anterior, la espera se limita a los milisegundos y nanosegundos indicados.
- void run(): Comienza a ejecutar el objeto Runnable de destino del hilo, si hay alguno.
- void setDaemon(boolean on): Marca el Thread, tanto como de demonio o como de usuario, dependiendo del valor del parámetro.
- void setName(String name): Cambia el nombre del Thread.
- void setPriority(int newPriority): Comprueba si el Thread tiene permiso para modificar su propia prioridad. Si es así, configura la prioridad pedida.
- static void sleep(long millis): Causa que el Thread actual ceda el procesador y se deje de ejecutar durante el tiempo indicado en milisegundos.
- static void sleep(long millis, int nanos): Igual que el anterior, sólo que el tiempo se indica en milisegundos y nanosegundos.
- void start(): Llama a este hilo para que empiece a ejecutarse; la JVM llama al método run() del Thread.
- String toString(): Devuelve una representación en un String del Thread con su nombre, prioridad, y ThreadGroup.
- static void yield(): Causa que el Thread actual ceda el procesador; el planificador permitirá que se ejecuten otros Threads.

Versiones anteriores del JDK incluían algunos métodos adicionales en la clase Thread que no aparecen en la lista anterior como son: stop(), suspend() y resume().

En las versiones actuales del JDK estos métodos aparecen como *Deprecated*, es decir, obsoletos. A pesar de que aparecen todavía están en el API (por razones de compatibilidad), están en desuso. Sun determinó que tienden a conducir a interbloqueos, de modo que no recomiendan más su uso.

10.3.-CREACIÓN DE THREADS

Existen dos mecanismos que permiten la creación y manipulación de Threads:

- Creando objetos que deriven directamente de la clase THREAD
- Creando objetos que implementen el interfaces RUNNABLE

10.3.1.-INSTANCIANDO LA CLASE THREAD

La manera más rápida de construir un programa multihilo es crear una instancia de la clase Thread, es decir, algo así:

```
Thread miThread = new Thread();
```

Hay 7 constructores distintos para Thread que pueden llegar a recibir 3 parámetros:

- Threadgroup: Cada thread de Java es miembro de un grupo de threads. Los grupos proporcionan un mecanismo para la colección de varios threads dentro de un sólo objeto y la manipulación de esos threads de una vez, mejor que de forma individual. Por ejemplo, se puede arrancar o suspender todos los threads de un grupo con una sola llamada a un método. Los grupos de threads de Java están implementados por la clase ThreadGroup del paquete java.lang.

El sistema de ejecución pone un thread dentro de un grupo durante su construcción. Cuando se crea un thread, también se puede permitir que el sistema de ejecución ponga el nuevo thread en algún grupo por defecto razonable o se puede especificar explícitamente el grupo del nuevo thread. El thread es un miembro permanente del grupo al que se unió durante su creación - no se puede mover un thread a otro grupo después de haber sido creado.

Si se crea un nuevo Thread sin especificar su grupo en el constructor, el sistema de ejecución automáticamente sitúa el nuevo thread en el mismo grupo que del thread que lo creó (conocido como el grupo de threads actual y el thread actual, respectivamente).

Cuando se arranca por primera vez una aplicación Java, el sistema de ejecución crea un ThreadGroup llamado "main". Entonces, a menos que se especifique otra cosa, todos los nuevos threads que se creen se convierten en miembros del grupo de threads "main".

- Runnable: Se utiliza para configurar un destino para el nuevo Thread a ejecutar. Si prescinde del mismo o lo configura como null, el nuevo Thread se inicia llamando al actual método run() del Thread. Si el nuevo Thread tiene un destino no null, entonces se invoca al método run() del destino al iniciar el nuevo Thread.
- String: Se utiliza para nombrar al nuevo Thread.

En Java existen dos tipos de Threads: la mayor parte son hilos del usuario, tienen una interfaz de usuario y existen para servir a esta interfaz. Algunos hilos solamente se ejecutan en

background. Se denominan Threads demonio. Llamando a `setDaemon()` se puede especificar que un Thread es un demonio.

Heredando de la clase Thread

Además de crear instancias de la clase Thread, se puede hacer una clase que herede de la clase Thread. Para ello vamos a ver este ejemplo:

```
class Thread1 extends Thread
{
    public void run()
    {
        for( int i=0; i<100; i++ )
            System.out.println( "1" );
    }
}

class Thread2 extends Thread
{
    public void run()
    {
        for( int i=0; i<100; i++ )
            System.out.println( "2" );
    }
}
```

Vemos que hasta aquí simplemente son dos clases cuyo método `run()`, una vez ejecutado, visualizará 100 caracteres "1" o "2" en pantalla. Nada extraño.

Bien, pues vamos a hacer que se ejecuten a la vez. Para ello hemos hecho que desciendan de la clase Thread. Cualquier subclase de Thread permite después crear objetos y ejecutar su método `run()` (que debe redefinirse, ya que por omisión no hace nada) en paralelo con otros.

Lo hacemos en esta clase principal:

```
public class ThreadTest
{
    public static void main( String[] pars )
    {
        Thread1 t1 = new Thread1();
        Thread2 t2 = new Thread2();
        t1.start(); // Ejecuta t1.run()
        t2.start(); // Ejecuta t2.run()
        System.out.println( "Ahora estan los 2 en marcha" );
        try {
            t1.join(); // Espera hasta que acabe t1.run()
            t2.join(); // Espera hasta que acabe t2.run()
        } catch( InterruptedException e ) { }
        System.out.println();
        System.out.println( "Final de los 2 threads" );
    }
}
```



```

    ...
}

```

Cuando se invoque el método `start()` del thread, automáticamente llamará al método `run()` del objeto `Runnable` que se le asoció en el constructor, en este caso, a la instancia de `MiClase`.

Vamos a ver ahora un ejemplo de utilización de `Runnable` con varios objetos:

```

class MiEjecutable implements Runnable
{
    static long horaInicio = System.currentTimeMillis();
    int cont = 0;
    int num;
    MiEjecutable( int numObjeto )
    {
        num = numObjeto;
    }
    public void run()
    {
        while( System.currentTimeMillis() - horaInicio < 3000)
        {
            System.out.print( num ); // visualiza su número durante 3s.
            cont++; // cuenta el nº de visualizaciones
        }
    }
    public void visualizar()
    {
        System.out.println("Objeto num." + num + " - " + cont + " veces." );
    }
}

```

Vemos que esta clase simplemente se construye pasándole un número, y luego al ejecutarse el método `run()` se visualizará en pantalla ese número durante 3 segundos.

Lo que vamos a hacer es crear varias instancias con distintos números y ejecutarlas a la vez para ver cómo se van intercalando:

```

public class ThreadTest3
{
    public static void main( String[] pars )
    {
        Thread [] t = new Thread [4];
        MiEjecutable [] o = new MiEjecutable [4];
        for (int i = 0; i < 4; i++)
        {
            o[i] = new MiEjecutable( i );
            t[i] = new Thread( o[i], "Objeto-" + i );
            System.out.println( "nuevo Thread() " + (t[i] == null ?
                "fallo" : "correcto") + " - " + t[i].getName() );
        }
        for (int i = 0; i < 4; i++)
            t[i].start();
        try {
            for (int i = 0; i < 4; i++)
                t[i].join();
        }
    }
}

```



```

public void run()
{
    while (System.currentTimeMillis() - horaInicial < 3000)
    {
        // Espera un segundo:
        try {
            Thread.sleep( 1000 );
        } catch (InterruptedException e) { }
        System.out.println( "Estoy en el hilo " +
            Thread.currentThread().getName() + " - segundo " +
            System.currentTimeMillis()- horaInicial)/1000.0 );
    }
}

```

Básicamente, el método `run()` visualiza cada segundo de ejecución durante tres segundos. Obsérvese como funciona el método `sleep()`, que deja suspendido el thread durante los milisegundos que se indican. Ahora vamos a crear un objeto y ejecutar cuatro veces su método `run()` concurrentemente:

```

public class ThreadTest4
{
    public static void main( String[] pars )
    {
        Hilo a = new Hilo(); // un único objeto...
        for (int i=0; i < 4; i++)
        {
            Thread t = new Thread(a); // ...pero cuatro threads
            System.out.println( "nuevo Thread() " + (t == null ? "fallo" :
                correcto) + " - " + t.getName() );
            t.start();
        }
    }
}

```

Una posible salida de este programa es:

```

Creado nuevo objeto Hilo

nuevo Thread() correcto - Thread-1

nuevo Thread() correcto - Thread-2

nuevo Thread() correcto - Thread-3

nuevo Thread() correcto - Thread-4

Estoy en el hilo Thread-1 - segundo 1.04

Estoy en el hilo Thread-2 - segundo 1.04

Estoy en el hilo Thread-3 - segundo 1.04

Estoy en el hilo Thread-4 - segundo 1.04

```

Estoy en el hilo Thread-1 - segundo 2.03

Estoy en el hilo Thread-2 - segundo 2.03

Estoy en el hilo Thread-3 - segundo 2.09

Estoy en el hilo Thread-4 - segundo 2.09

Estoy en el hilo Thread-1 - segundo 3.08

Estoy en el hilo Thread-2 - segundo 3.08

Estoy en el hilo Thread-3 - segundo 3.08

Estoy en el hilo Thread-4 - segundo 3.08

Hay que pensar en esto como en que el mismo programa (el método run()) sobre un mismo objeto) se está ejecutando a la vez con distintos flujos de control. Por supuesto, en este caso coincidirán todos los atributos (sean de clase o de instancia), las que sí serán distintas serán las variables locales del método run().

Una consideración: una vez que empieza a ejecutarse el método run(), sea cual sea la forma elegida de hacerlo, puede llamar a cualquier otro método de cualquier otro objeto y eso seguirá formando un thread propio de ejecución.

10.4.-CICLO DE VIDA DE UN THREAD

Un Thread desde su ejecución hasta su muerte pasa por diversos estados de vida. Estos estados y sus transiciones se muestran en la figura 10.1.

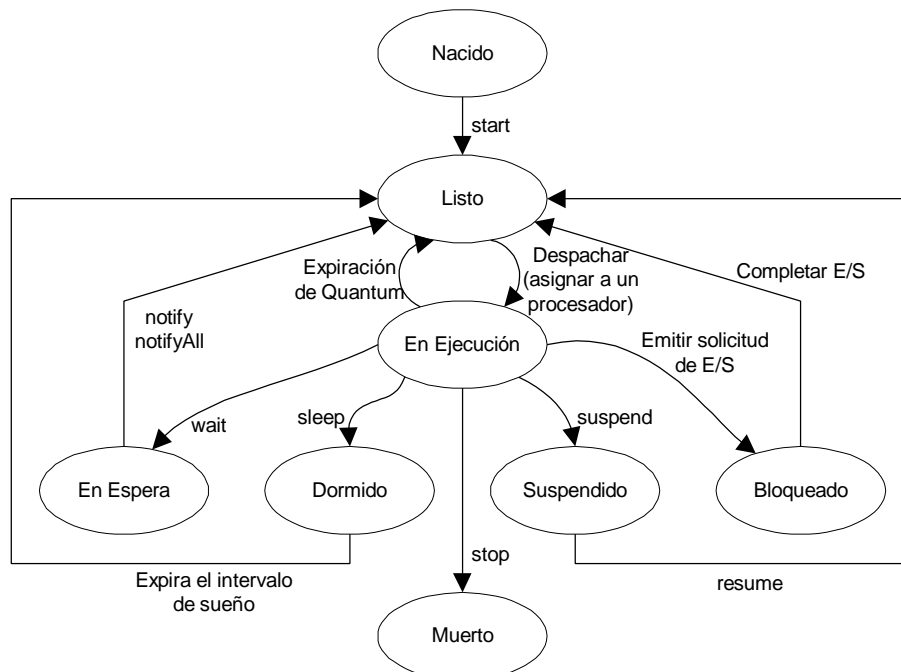


Figura 10.1: Estados por los que pasa un Thread

Un thread tras su creación pasa a estado LISTO de ahí puede pasar a EN EJECUCIÓN o no, esto dependerá del Dispatcher. Una vez EN EJECUCIÓN procesará su código hasta que se le acabe el quantum asignado por el S.O. o bien hasta que termine (si le da tiempo), o también se puede dar el caso de que alguien lo elimine mediante stop(). Los procesos pasaran a ejecución dependiendo de sus prioridades y haciendo un *round robin*.

Una vez EN EJECUCIÓN los threads pueden, o pasar a LISTO de nuevo (si se acaba el quantum) o pasar a otros estados, a saber: EN ESPERA, DORMIDO, SUSPENDIDO y BLOQUEADO. Esto dependerá de la ejecución de ciertos métodos sobre el Thread (o la ocurrencia de ciertos sucesos).

- Un Thread pasará a DORMIDO cuando se invoque el método sleep(), permanecerá así (sin consumir recursos) hasta que se le acabe el tiempo de "siesta", momento en el que volverá a LISTO. En este estado, el thread no consume recursos, es decir, no es candidato a serle asignado el procesador hasta que no despierte.
- El Thread pasará a BLOQUEADO cuando tenga que sufrir una espera debida a E/S, saldrá de este estado en cuanto termine la E/S (tampoco consume recursos mientras espera).
- El estado SUSPENDIDO es para aquellos Threads que han sufrido la invocación del método suspend(), en este estado permanecerán hasta que alguien los llame mediante resume(). Por supuesto tampoco consumen recursos en este estado.
- Y por último el Thread pasará a EN ESPERA cuando alguien invoque un wait(), entonces el Thread pasará a esperar en una cola. Esto implica que la próxima vez que se ejecute un notify(), el Thread que se despertará será el primero que entró. También podemos ejecutar notifyAll(), de forma que sacamos a todos de la cola.

Resumiendo, un thread entra en el estado "No Ejecutable" cuando ocurre uno de estos cuatro eventos:

- Alguien llama a su método sleep().
- Alguien llama a su método suspend().
- El thread utiliza su método wait() para esperar una condición variable.
- El thread está bloqueado durante la I/O.

Esta lista indica la ruta de escape para cada entrada en el estado "No Ejecutable":

- Si se ha puesto a dormir un thread, deben pasar el número de milisegundos especificados.
- Si se ha suspendido un thread, alguien debe llamar a su método resume().
- Si un thread está esperando una condición variable, siempre que el objeto propietario de la variable renuncie mediante notify() o notifyAll().
- Si un thread está bloqueado durante la I/O, cuando se complete la I/O.

Un thread puede morir de dos formas: por causas naturales o siendo asesinado (parado). Una muerte natural se produce cuando su método `run()` sale normalmente. Se puede matar un thread en cualquier momento llamando a su método `stop()`.

El método `stop()` provoca una terminación súbita del método `run()` del thread. Si el método `run()` estuviera realizando cálculos sensibles, `stop()` podría dejar el programa en un estado inconsistente. Normalmente, no se debería llamar al método `stop()`.

10.5.-MÉTODOS SINCRONIZADOS

Modificador *synchronized*

El modificador *synchronized* puede aplicarse sobre un método:

- de instancia: indicando entonces que sólo puede haber un thread ejecutando un método sincronizado sobre el objeto correspondiente en cada momento.
- de clase: sólo un thread creado a partir de un objeto de la clase puede estar ejecutando un método sincronizado en cada momento.

Por ejemplo:

```
public class MiClaseThread implements Runnable
{
    ...
    synchronized void m1( ) //Método de instancia sincronizado
    {...}

    synchronized static void m2( ) //Método de clase sincronizado
    {...}

    public void run()
    {
        m1();
        m2();
    }
}

....

public static void main(String[] args)
{
    MiClaseThread z1=new MiClaseThread();
    MiClaseThread z2=new MiClaseThread();
    Thread t1=new Thread(z1);
    Thread t2=new Thread(z1);
    Thread t3=new Thread(z2);
    t1.start();
    t2.start();
    t3.start();
}
```

En el método de instancia: Si t1 entra en el método m1() y, estando dentro, se pasa el control del procesador a t2, este thread no podrá entrar porque es un thread construido a partir de la misma instancia Runnable. Sin embargo el t3 sí podrá ejecutar ese método porque está creado a partir de otra instancia de la clase Runnable.

En el método de clase: Si t1 entra en el método m2() y, estando dentro, se pasa el control del procesador a t2, este thread no podrá entrar porque es un thread construido a partir de la misma clase (MiClaseThread). t3 tampoco podrá ejecutar el método m2() por la misma razón, están instanciados a partir de la misma clase (MiClaseThread)

¿Cuándo necesitaremos usar este mecanismo? Consideremos el siguiente ejemplo:

Hay un algoritmo de ordenación que se llama *ShakerSort* para ordenar un vector de números, que funciona como una ordenación por burbuja que hace una pasada ascendente y otra descendente, y así sucesivamente.

Podríamos hacer dos threads para que a la vez se hicieran las dos pasadas ascendente y descendente, un thread yendo hacia arriba y otro hacia abajo:

```
public class ShakerSort implements Runnable
{
    boolean ascendente;
    static int[] v; // Vector de prueba
    ShakerSort( boolean esAscendente )
    {
        ascendente = esAscendente;
    }
    public static void inicVector()
    { // Inicializa v a 10 valores de prueba
        int [] v2 = { 10, 2, 3, 9, 4, 6, 7, 5, 8, 1 };
        v = v2;
    }
    static String vector()
    {
        String s = "[ ";
        for (int i = 0; i < v.length; i++ )
            s += v[i] + " ";
        return s + " ]";
    }
    public void run()
    {
        if (ascendente)
        {
            int actual;
            int pasada = 1;
            int ultimoCambioAnterior = v.length - 1;
            int ultimoCambio;
            do
            {
                actual = 0;
                ultimoCambio = -1;
                while (actual < ultimoCambioAnterior)
                {
```

Laboratorio de Informática II - Threads

```
        if (v[actual] > v[actual+1])
        {
            intercambia( actual, actual+1 );
            ultimoCambio = actual;
        }
        actual++;
    }
    System.out.println( "A Ascendente - pasada
    &quot; + pasada++ + &quot; &quot; + vector() );
    ultimoCambioAnterior = ultimoCambio;
} while (ultimoCambioAnterior > 0);
System.out.println( "A Fin de burbuja
    ascendente! &quot; + vector() );
}
else {
    int actual;
    int pasada = 1;
    int ultimoCambioAnterior = - 1;
    int ultimoCambio;
    do {
        actual = v.length - 2;
        ultimoCambio = actual + 1;
        while (actual > ultimoCambioAnterior) {
            if (v[actual] > v[actual+1]) {
                intercambia( actual, actual+1 );
                ultimoCambio = actual;
            }
            actual--;
        }
        System.out.println( "D Descendente &#150;
            pasada &quot; + pasada++ + &quot; &quot; +
            vector() );
        ultimoCambioAnterior = ultimoCambio;
    } while (ultimoCambioAnterior < v.length - 2);
    System.out.println( "D Fin de burbuja descendente!&quot;
        + vector() );
}
}

void intercambia( int i, int j ) {
    int temp = v[i];
    v[i] = v[j];
    v[j] = temp;
}

public static void main( String[] pars ) {
    ShakerSort o = new ShakerSort( true );
    inicVector();
    System.out.println( "Prueba / Burbuja ascendente: &quot; +
        vector() );

    o.run();
    System.out.println();
    inicVector();
    System.out.println( "Prueba de ShakerSort: &quot; + vector() );
}
```

```

        ShakerSort oDesc = new ShakerSort( false );
        Thread t1 = new Thread(o);
        Thread t2 = new Thread(oDesc);
        t1.start();
        t2.start();
        try {
            t1.join();
            t2.join();
        } catch (InterruptedException e) {}
    }
}

```

Una posible salida de este ejemplo es:

Prueba / Burbuja ascendente: [10 2 3 9 4 6 7 5 8 1]

A Ascendente - pasada 1 [2 3 9 4 6 7 5 8 1 10]

A Ascendente - pasada 2 [2 3 4 6 7 5 8 1 9 10]

A Ascendente - pasada 3 [2 3 4 6 5 7 1 8 9 10]

A Ascendente - pasada 4 [2 3 4 5 6 1 7 8 9 10]

A Ascendente - pasada 5 [2 3 4 5 1 6 7 8 9 10]

A Ascendente - pasada 6 [2 3 4 1 5 6 7 8 9 10]

A Ascendente - pasada 7 [2 3 1 4 5 6 7 8 9 10]

A Ascendente - pasada 8 [2 1 3 4 5 6 7 8 9 10]

A Ascendente - pasada 9 [1 2 3 4 5 6 7 8 9 10]

A Fin de burbuja ascendente! [1 2 3 4 5 6 7 8 9 10]

Prueba de ShakerSort: [10 2 3 9 4 6 7 5 8 1]

D Descendente - pasada 1 [2 3 4 9 10 6 1 7 5 8]

A Ascendente - pasada 1 [2 3 4 9 10 6 1 5 7 8]

D Descendente - pasada 2 [2 3 4 9 1 10 6 5 7 8]

D Descendente - pasada 3 [2 3 4 9 1 6 5 10 7 8]

A Ascendente - pasada 2 [2 3 4 9 1 6 5 10 7 8]

D Descendente - pasada 4 [2 3 4 9 1 6 5 7 10 8]

D Descendente - pasada 5 [2 3 4 9 1 6 5 7 8 10]

D Fin de burbuja descendente! [2 3 4 1 6 9 9 7 8 10]

```

A Ascendente - pasada 3 [ 2 3 4 1 6 5 9 7 8 10 ]
A Ascendente - pasada 4 [ 2 3 1 4 5 6 9 7 8 10 ]
A Ascendente - pasada 5 [ 2 1 3 4 5 6 9 7 8 10 ]
A Ascendente - pasada 6 [ 1 2 3 4 5 6 9 7 8 10 ]
A Fin de burbuja ascendente! [ 1 2 3 4 5 6 9 7 8 10 ]

```

¿A qué se debe el problema? A que el vector existe sólo una vez y, si tenemos mala suerte, puede ocurrir que el grupo de sentencias:

```

if( v[actual] > v[actual+1])
{
    intercambia( actual, actual+1 );
    ...

```

se ejecute a la vez sobre la misma pareja de valores en los dos threads. Al ocurrir eso puede pasar que el flujo de control de cada thread se corte en cualquier punto intermedio para ejecutarse la otra, por lo que en el peor de los casos los dos threads pueden comparar la misma pareja reconociendo que están en orden inverso y ejecutando dos el método *intercambia()*.

Esta situación es la crítica en códigos de threads. Hay que identificar qué métodos no pueden ser ejecutados a la vez en varios threads y marcarlos como `synchronized`: esto significará que si uno de los threads ha entrado en ese método ningún otro thread podrá entrar (ni a ese ni a ningún otro "sincronizado") hasta que el primero lo acabe.

Así, los cambios serían, en el método `run()`:

```

while (actual < ultimoCambioAnterior)
{
    if (hazIntercambio( actual ))
        ultimoCambio = actual;
    actual++;
}

```

Y el método de intercambio:

```

synchronized boolean hazIntercambio( int i )
{
    if (v[i] > v[i+1])
    {
        Thread.yield(); // Poner esto para causar problemas sin synchronized
        int temp = v[i];
        v[i] = v[i+1];
        v[i+1] = temp;
        return true;
    }
    else return false;
}

```

Pero esto no sería suficiente. Un método sincronizado lo marca el objeto si es de instancia. Esto es, cuando sobre el objeto 'o' se ejecuta `hazIntercambio()` otro thread no pueden entrar

en el método `hazIntercambio()` con el mismo objeto, pero sí ejecutar el método `hazIntercambio()` de otro objeto.

Si queremos que no se ejecute ningún otro `hazIntercambio()` cuando se está haciendo el primero, debemos hacerlo de clase:

```
static synchronized boolean hazIntercambio( int i ) {
```

Con esto la salida ya no tendrá problemas:

```
Prueba de ShakerSort: [ 10 2 3 9 4 6 7 5 8 1 ]
A Ascendente - pasada 1 [ 2 1 3 9 4 6 7 5 8 10 ]
D Descendente - pasada 1 [ 1 2 3 4 9 6 7 5 8 10 ]
D Descendente - pasada 2 [ 1 2 3 4 5 6 7 9 9 10 ]
A Ascendente - pasada 2 [ 1 2 3 4 5 6 7 8 9 10 ]
D Descendente - pasada 3 [ 1 2 3 4 5 6 7 8 9 10 ]
A Ascendente - pasada 3 [ 1 2 3 4 5 6 7 8 9 10 ]
D Fin de burbuja descendente! [ 1 2 3 4 5 6 7 8 9 10 ]
A Fin de burbuja ascendente! [ 1 2 3 4 5 6 7 8 9 10 ]
```

Uso de código *synchronized*

No sólo los métodos pueden ser *synchronized*, también puede escribirse en cualquier punto un código precedido de la construcción

```
synchronized (objeto)
{
    ...
}
```

De modo que el código entre llaves esperará a que no haya ningún método crítico (*synchronized*) ejecutándose sobre el objeto para empezar a ejecutarse, y hasta que acabe no dejará que ningún otro método o código crítico (indicado a su vez con *synchronized*) se ejecute.