# TEMA 15: THREADS: PROGRAMAS MULTITAREA

## 1. INTRODUCCIÓN

Los procesadores y los Sistemas Operativos modernos permiten la ***multitarea***, es decir, la realización simultánea de dos o más actividades (al menos aparentemente). En la realidad, un ordenador con una sola CPU no puede realizar dos actividades a la vez. Sin embargo los Sistemas Operativos actuales son capaces de ejecutar varios programas "simultáneamente" aunque sólo se disponga de una CPU: reparten el tiempo entre dos (o más) actividades, o bien utilizan los tiempos muertos de una actividad (por ejemplo, operaciones de lectura de datos desde el teclado) para trabajar en la otra. En ordenadores con dos o más procesadores la multitarea es real, ya que cada procesador puede ejecutar un ***hilo*** o ***thread*** diferente.

Un ***proceso*** es un programa ejecutándose de forma independiente y con un espacio propio de memoria. Un Sistema Operativo multitarea es capaz de ejecutar más de un ***proceso*** simultáneamente. Un ***thread*** o ***hilo*** es un ***flujo secuencial simple*** dentro de un ***proceso***. Un único ***proceso*** puede tener varios ***hilos*** ejecutándose. Por ejemplo el programa ***Netscape*** sería un proceso, mientras que cada una de las ventanas que se pueden tener abiertas simultáneamente trayendo páginas HTML estaría formada por al menos un ***hilo***.

Un sistema multitarea da realmente la impresión de estar haciendo varias cosas a la vez y eso es una gran ventaja para el usuario. Sin el uso de ***threads*** hay tareas que son prácticamente imposibles de ejecutar, particularmente las que tienen tiempos de espera importantes entre etapas.

Los ***threads*** o ***hilos*** de ejecución permiten organizar los recursos del ordenador de forma que pueda haber varios programas actuando en paralelo. Un ***hilo*** de ejecución puede realizar cualquier tarea que pueda realizar un programa normal y corriente. Bastará con indicar lo que tiene que hacer en el método ***run()***, que es el que define la actividad principal de las ***threads***.

Los ***threads*** pueden ser ***daemon*** o ***no daemon***. Son ***daemon*** aquellos hilos que realizan en ***background*** (en un segundo plano) servicios generales, esto es, tareas que no forman parte de la esencia del programa y que se están ejecutando mientras no finalice la aplicación. Un ***thread daemon*** podría ser por ejemplo aquél que está comprobando permanentemente si el usuario pulsa un botón. Un programa de ***Java*** finaliza cuando sólo quedan corriendo ***threads*** de tipo ***daemon***. Por defecto, y si no se indica lo contrario, los ***threads*** son del tipo ***no daemon***.

## 2. CREACIÓN DE THREADS.

En ***Java*** hay dos formas de crear nuevos ***threads***. La primera de ellas consiste en crear una nueva clase que herede de la clase ***java.lang.Thread*** y sobrecargar el método ***run()*** de dicha clase. El segundo método consiste en declarar una clase que implemente la interface ***java.lang.Runnable***, la cual declarará el método ***run()***; posteriormente se crea un objeto de tipo ***Thread*** pasándole como argumento al constructor el objeto creado de la nueva clase (la que implementa la interface ***Runnable***). Como ya se ha apuntado, tanto la clase ***Thread*** como la interface ***Runnable*** pertenecen al package ***java.lang***, por lo que no es necesario importarlas.

A continuación se presentan dos ejemplos de creación de ***threads*** con cada uno de los dos métodos citados.

### 2.1. Creación de threads derivando de la clase Thread.

Considérese el siguiente ejemplo de declaración de una nueva clase:

```
public class SimpleThread extends Thread {

      // constructor

      public SimpleThread (String str) {

            super(str);

      }

      // redefinición del método run()

      public void run() {

            for(int i=0;i<10;i++)

                  System.out.println("Este es el thread : " + getName());

      }

}
```

En este caso, se ha creado la clase ***SimpleThread***, que hereda de ***Thread***. En su constructor se utiliza un ***String*** (opcional) para poner nombre al nuevo ***thread*** creado, y mediante ***super()*** se llama al constructor de la super-clase ***Thread***. Asimismo, se redefine el método ***run()***, que define la principal actividad del ***thread***, para que escriba 10 veces el nombre del ***thread*** creado.

Para poner en marcha este nuevo ***thread*** se debe crear un objeto de la clase ***SimpleThread***, y llamar al método ***start()***,heredado de la super-clase ***Thread***, que se encarga de llamar a ***run()***. Por ejemplo:

```
SimpleThread miThread = new SimpleThread("Hilo de prueba");

miThread.start();
```

### 2.2. Creación de threads implementando la interface Runnable.

Esta segunda forma también requiere que se defina el método ***run()***, pero además es necesario crear un objeto de la clase ***Thread*** para lanzar la ejecución del nuevo hilo. Al constructor de la clase ***Thread*** hay que pasarle una referencia del objeto de la clase que implementa la interface ***Runnable***. Posteriormente, cuando se ejecute el método ***start()*** del ***thread***, éste llamará al método ***run()*** definido en la nueva clase. A continuación se muestra el mismo estilo de clase que en el ejemplo anterior implementada mediante la interface ***Runnable***:

```
public class SimpleRunnable implements Runnable {

     // se crea un nombre

     String nameThread;
```

```
        // constructor
        public SimpleRunnable (String str) {
            nameThread = str;
        }

        // definición del método run()
        public void run() {
            for(int i=0;i<10;i++)
                System.out.println("Este es el thread: " + nameThread);
        }
    }
```

El siguiente código crea un nuevo ***thread*** y lo ejecuta por este segundo procedimiento:

```
SimpleRunnable p = new SimpleRunnable("Hilo de prueba");
// se crea un objeto de la clase Thread pasándolo el objeto Runnable como argumento
Thread miThread = new Thread(p);
// se arranca el objeto de la clase Thread
miThread.start();
```

Este segundo método cobra especial interés con las ***applets***, ya que cualquier ***applet*** debe heredar de la clase ***java.applet.Applet***, y por lo tanto ya no puede heredar de ***Thread***. Véase el siguiente ejemplo:

```
class ThreadRunnable extends Applet implements Runnable {
    private Thread runner=null;
    // se redefine el método start() de Applet
    public void start() {
        if (runner == null) {
            runner = new Thread(this);
            runner.start(); // se llama al método start() de Thread
        }
    }
    // se redefine el método stop() de Applet
    public void stop(){
```

```
            runner = null; // se libera el objeto runner
        }
    }
```

En este ejemplo, el argumento ***this*** del constructor de ***Thread*** hace referencia al objeto ***Runnable*** cuyo método ***run()*** debería ser llamado cuando el hilo ejecutado es un objeto de ***ThreadRunnable***.

La elección de una u otra forma -derivar de ***Thread*** o implementar ***Runnable***- depende del tipo de clase que se vaya a crear. Así, si la clase a utilizar ya hereda de otra clase (por ejemplo un ***applet***, que siempre hereda de ***Applet***), no quedará más remedio que implementar ***Runnable***, aunque normalmente es más sencillo heredar de ***Thread***.

## 3. CICLO DE VIDA DE UN THREAD.

En el apartado anterior se ha visto cómo crear nuevos objetos que permiten incorporar en un programa la posibilidad de realizar varias tareas simultáneamente. Un ***thread*** puede presentar cuatro estados distintos:

1. *Nuevo* (***New***): El ***thread*** ha sido creado pero no inicializado, es decir, no se ha ejecutado todavía el método ***start()***. Se producirá un mensaje de error (***IllegalThreadStateException***) si se intenta ejecutar cualquier método de la clase ***Thread*** distinto de ***start()***.
2. *Ejecutable* (***Runnable***): El ***thread*** puede estar ejecutándose, siempre y cuando se le haya asignado un determinado tiempo de CPU. En la práctica puede no estar siendo ejecutado en un instante determinado en beneficio de otro ***thread***.
3. *Bloqueado* (***Blocked*** o ***Not Runnable***): El ***thread*** podría estar ejecutándose, pero hay alguna actividad interna suya que lo impide, como por ejemplo una espera producida por una operación de escritura o lectura de datos por teclado (E/S). Si un t***hread*** está en este estado, no se le asigna tiempo de CPU.
4. *Muerto* (***Dead***): La forma habitual de que un ***thread*** muera es finalizando el método ***run()***.

También puede llamarse al método ***stop()*** de la clase ***Thread***, aunque dicho método es considerado "peligroso" y no se debe utilizar.

A continuación se explicarán con mayor detenimiento los puntos anteriores.

### 3.1. Ejecución de un nuevo thread.

La creación de un nuevo ***thread*** no implica necesariamente que se empiece a ejecutar algo. Hace falta iniciarlo con el método ***start()***, ya que de otro modo, cuando se intenta ejecutar cualquier método del ***thread*** -distinto del método ***start()***- se obtiene en tiempo de ejecución el error ***IllegalThreadStateException***.

El método ***start()*** se encarga de llamar al método ***run()*** de la clase ***Thread***. Si el nuevo ***thread*** se ha creado heredando de la clase ***Thread*** la nueva clase deberá redefinirir el método ***run()*** heredado. En el caso de utilizar una clase que implemente la interface ***Runnable***, el método ***run()***de la clase ***Thread*** se ocupa de llamar al método ***run()*** de la nueva clase.

Una vez que el método ***start()*** ha sido llamado, se puede decir ya que el ***thread*** está "corriendo" (***running***), lo cual no quiere decir que se esté ejecutando en todo momento, pues ese ***thread*** tiene que compartir el tiempo de la CPU con los demás ***threads*** que también estén ***running***. Por eso más bien se dice que dicha ***thread*** es ***runnable***.

### 3.2. Detener un Thread temporalmente.

El sistema operativo se ocupa de asignar tiempos de CPU a los distintos ***threads*** que se estén ejecutando simultáneamente. Aun en el caso de disponer de un ordenador con más de un procesador (2 ó más CPUs), el número de ***threads*** simultáneos suele siempre superar el número de CPUs, por lo que se debe repartir el tiempo de forma que parezca que todos los procesos corren a la vez (quizás más lentamente), aun cuando sólo unos pocos pueden estar ejecutándose en un instante de tiempo.

Los tiempos de CPU que el sistema continuamente asigna a los distintos ***threads*** en estado ***runnable*** se utilizan en ejecutar el método ***run()*** de cada ***thread***. Por diversos motivos, un ***thread*** puede en un determinado momento renunciar "voluntariamente" a su tiempo de CPU y otorgárselo al sistema para que se lo asigne a otro ***thread***. Esta "renuncia" se realiza mediante el método ***yield()***. Es importante que este método sea utilizado por las actividades que tienden a "monopolizar" la CPU. El método ***yield()*** viene a indicar que en ese momento no es muy importante para ese ***thread*** el ejecutarse continuamente y por lo tanto tener ocupada la CPU. En caso de que ningún ***thread*** esté requiriendo la CPU para una actividad muy intensiva, el sistema volverá casi de inmediato a asignar nuevo tiempo al ***thread*** que fue "generoso" con los demás. Por ejemplo, en un Pentium II 400 Mhz es posible llegar a más de medio millón de llamadas por segundo al método ***yield()***, dentro del método ***run()***, lo que significa que llamar al método ***yield()*** apenas detiene al ***thread***, sino que sólo ofrece el control de la CPU para que el sistema decida si hay alguna otra tarea que tenga mayor prioridad.

Si lo que se desea es parar o bloquear temporalmente un ***thread*** (pasar al estado ***Not Runnable***), existen varias formas de hacerlo:

1. Ejecutando el método ***sleep()*** de la clase ***Thread***. Esto detiene el ***thread*** un tiempo pre-establecido. De ordinario el método ***sleep()*** se llama desde el método ***run()***.
2. Ejecutando el método ***wait()*** heredado de la clase ***Object***, a la espera de que suceda algo que es necesario para poder continuar. El ***thread*** volverá nuevamente a la situación de ***runnable*** mediante los métodos ***notify()*** o ***notifyAll()***, que se deberán ejecutar cuando cesa la condición que tiene detenido al thread.
3. Cuando el ***thread*** está esperando para realizar operaciones de Entrada/Salida o Input/Output (E/S ó I/O).
4. Cuando el ***thread*** está tratando de llamar a un método ***synchronized*** de un objeto, y dicho objeto está bloqueado por otro ***thread.***

Un ***thread*** pasa automáticamente del estado ***Not Runnable*** a ***Runnable*** cuando cesa alguna de las condiciones anteriores o cuando se llama a ***notify()*** o ***notifyAll()***.

La clase ***Thread*** dispone también de un método ***stop()***, pero ***no se debe utilizar*** ya que puede provocar bloqueos del programa (***deadlock***). Hay una última posibilidad para detener un ***thread***, que consiste en ejecutar el método ***suspend()***. El ***thread*** volverá a ser ejecutable de nuevo ejecutando el método ***resume()***. Esta última forma también se desaconseja, por razones similares a la utilización del método ***stop()***.

El método ***sleep()*** de la clase ***Thread*** recibe como argumento el tiempo en *milisegundos* que ha de permanecer detenido. Adicionalmente, se puede incluir un número entero con un tiempo adicional en *nanosegundos*. Las declaraciones de estos métodos son las siguientes:

```
public static void sleep(long millis) throws InterruptedException

public static void sleep(long millis, int nanosecons) throws InterruptedException
```

Considérese el siguiente ejemplo:

```
System.out.println ("Contador de segundos");

int count=0;

public void run () {

     try {

          sleep(1000);

          System.out.println(count++);

     } catch (InterruptedException e){}

}
```

Se observa que el método ***sleep()*** puede lanzar una ***InterruptedException*** que ha de ser capturada. Así se ha hecho en este ejemplo, aunque luego no se gestiona esa excepción.

La forma preferible de detener temporalmente un ***thread*** es la utilización conjunta de los métodos ***wait()*** y ***notifyAll()***. La principal ventaja del método ***wait()*** frente a los métodos anteriormente descritos es que libera el bloqueo del objeto. por lo que el resto de threads que se encuentran esperando para actuar sobre dicho objeto pueden llamar a sus métodos. Hay dos formas de llamar a ***wait()***:

1. Indicando el tiempo máximo que debe estar parado (en *milisegundos* y con la opción de indicar también *nanosegundos*), de forma análoga a ***sleep()***. A diferencia del método ***sleep()***, que simplemente detiene el ***thread*** el tiempo indicado, el método ***wait()*** establece el tiempo máximo que debe estar parado. Si en ese plazo se ejecutan los métodos ***notify()*** o ***notifyAll()*** que indican la liberación de los objetos bloqueados, el ***thread*** continuará sin esperar a concluir el tiempo indicado. Las dos declaraciones del método ***wait()*** son como siguen:

   ```
   public final void wait(long timeout) throws InterruptedException

   public final void wait(long timeout, int nanos) throws InterruptedException
   ```

2. Sin argumentos, en cuyo caso el ***thread*** permanece parado hasta que sea reinicializado explícitamente mediante los métodos ***notify()*** o ***notifyAll()***.

   public final void wait() throws InterruptedException

Los métodos ***wait()*** y ***notify()*** han de estar incluidas en un método ***synchronized,*** ya que de otra forma se obtendrá una excepción del tipo ***IllegalMonitorStateException*** en tiempo de ejecución. El uso típico de ***wait()*** es el de esperar a que se cumpla alguna determinada condición, ajena al propio ***thread***. Cuando ésta se cumpla, se utilizará el método ***notifyAll()*** para avisar a los distintos ***threads*** que pueden utilizar el objeto.

### 3.3. Finalizar un Thread.

Un ***thread*** finaliza cuando el método ***run()*** devuelve el control, por haber terminado lo que tenía que hacer (por ejemplo, un bucle ***for*** que se ejecuta un número determinado de veces) o por haberse dejado de cumplir una condición (por ejemplo, por un bucle ***while*** en el método ***run()***). Es habitual poner las siguientes sentencias en el caso de ***Applets Runnables***:

```
public class MyApplet extends Applet implements Runnable {
     // se crea una referencia tipo Thread
     private Thread AppletThread;
     ...
     // método start() del Applet
     public void start() {
          if(AppletThread == null){ // si no tiene un objeto Thread asociado
               AppletThread = new Thread(this, "El propio Applet");
               AppletThread.start(); // se arranca el thread y llama a run()
          }
     }
     // método stop() del Applet
     public void stop() {
          AppletThread = null; // iguala la referencia a null
     }
     // método run() por implementar Runnable
     public void run() {
          Thread myThread = Thread.currentThread();
          while (myThread == AppletThread) { // hasta que se ejecute stop()
               ...                                                    // código a
               ejecutar
          }
     }
} // fin de la clase MyApplet
```

donde ***AppletThread*** es el ***thread*** que ejecuta el método ***run()*** MyApplet. Para finalizar el thread basta poner la referencia ***AppletThread*** a ***null***. Esto se consigue en el ejemplo con el método ***stop()*** del ***applet*** (distinto del método ***stop()*** de la clase ***Thread***, que no conviene utilizar).

Para saber si un ***thread*** está "vivo" o no, es útil el método ***isAlive()*** de la clase ***Thread***, que devuelve ***true*** si el ***thread*** ha sido inicializado y no parado, y ***false*** si el ***thread*** es todavía nuevo (no ha sido inicializado) o ha finalizado.

## 4. SINCRONIZACIÓN.

La ***sincronización*** nace de la necesidad de evitar que dos o más ***threads*** traten de acceder a los mismos recursos al mismo tiempo. Así, por ejemplo, si un ***thread*** tratara de escribir en un fichero, y otro ***thread*** estuviera al mismo tiempo tratando de borrar dicho fichero, se produciría una situación no deseada. Otra situación en la que hay que sincronizar ***threads*** se produce cuando un ***thread*** debe esperar a que estén preparados los datos que le debe suministrar el otro ***thread***. Para solucionar estos tipos de problemas es importante poder ***sincronizar*** los distintos ***threads***.

Las secciones de código de un programa que acceden a un mismo recurso (un mismo objeto de una clase, un fichero del disco, etc.) desde dos ***threads*** distintos se denominan ***secciones críticas (critical sections)***. Para sincronizar dos o más ***threads***, hay que utilizar el modificador ***synchronized*** en aquellos métodos del ***objeto-recurso*** con los que puedan producirse situaciones conflictivas. De esta forma, ***Java*** bloquea (asocia un ***bloqueo*** o ***lock***) con el recurso sincronizado. Por ejemplo:

```
public synchronized void metodoSincronizado() {
     ...// accediendo por ejemplo a las variables de un objeto
     ...  }
```

La ***sincronización*** previene las interferencias solamente sobre un tipo de recurso: la memoria reservada para un objeto. Cuando se prevea que unas determinadas variables de una clase pueden tener problemas de sincronización, se deberán declarar como ***private*** (o ***protected***). De esta forma sólo estarán accesibles a través de métodos de la clase, que deberán estar ***sincronizados***.

Es muy importante tener en cuenta que si se sincronizan algunos métodos de un objeto pero otros no, el programa puede no funcionar correctamente. La razón es que los métodos no sincronizados pueden acceder libremente a las variables miembro, ignorando el bloqueo del objeto. Sólo los métodos sincronizados comprueban si un objeto está bloqueado. Por lo tanto, todos los métodos que accedan a un recurso compartido deben ser declarados ***synchronized***. De esta forma, si algún método accede a un determinado recurso, ***Java*** bloquea dicho recurso, de forma que el resto de ***threads*** no puedan acceder al mismo hasta que el primero en acceder termine de realizar su tarea. ***Bloquear un recurso u objeto*** significa que sobre ese objeto no pueden actuar simultáneamente dos ***métodos sincronizados***.

Existen dos niveles de bloqueo de un recurso. El primero es ***a nivel de objetos***, mientras que el segundo es ***a nivel de clases***. El primero se consigue declarando todos los métodos de una clase como ***synchronized***. Cuando se ejecuta un método ***synchronized*** sobre un objeto concreto, el sistema bloquea dicho objeto, de forma que si otro ***thread*** intenta ejecutar algún método sincronizado de ese objeto, este segundo método se mantendrá a la espera hasta que finalice el anterior (y desbloquee por lo tanto el objeto). Si existen varios objetos de una misma clase, como los bloqueos se producen a nivel de objeto, es posible tener distintos ***threads*** ejecutando métodos sobre diversos objetos de una misma clase.

El bloqueo de recursos ***a nivel de clases*** se corresponde con los ***métodos de clase*** o ***static***, y por lo tanto con las ***variables de clase*** o ***static***. Si lo que se desea es conseguir que un método bloquee simultáneamente una clase entera, es decir todos los objetos creados de una clase, es necesario declarar este método como ***synchronized static***. Durante la ejecución de un método declarado de esta segunda forma ningún método sincronizado tendrá acceso a ningún objeto de la clase bloqueada.

La sincronización puede ser problemática y generar errores. Un ***thread*** podría bloquear un determinado recurso de forma indefinida, impidiendo que el resto de ***threads*** accedieran al mismo. Para evitar esto último, habrá que utilizar la sincronización sólo donde sea estrictamente necesario.

Es necesario tener presente que si dentro un método sincronizado se utiliza el método ***sleep()*** de la clase ***Thread***, el objeto bloqueado permanecerá en ese estado durante el tiempo indicado en el argumento de dicho método. Esto implica que otros ***threads*** no podrán acceder a ese objeto durante ese tiempo, aunque en realidad no exista peligro de simultaneidad ya que durante ese tiempo el thread que mantiene bloqueado el objeto no realizará cambios. Para evitarlo es conveniente sustituir ***sleep()*** por el método ***wait()*** de la clase ***java.lang.Object*** heredado automáticamente por todas las clases. Cuando se llama al método ***wait()*** (siempre debe hacerse desde un método o bloque ***synchronized***) se libera el bloqueo del objeto y por lo tanto es posible continuar utilizando ese objeto a través de métodos sincronizados. El método ***wait()*** detiene el ***thread*** hasta que se llame al método ***notify()*** o ***notifyAll()*** del objeto, o finalice el tiempo indicado como argumento del método ***wait()***. El método ***unObjeto.notify()*** lanza una señal indicando al sistema que puede activar uno de los ***threads*** que se encuentren bloqueados esperando para acceder al objeto ***unObjeto***. El método ***notifyAll()*** lanza una señal a todos los ***threads*** que están esperando la liberación del objeto.

Los métodos ***notify()*** y ***notifyAll()*** deben ser llamados desde el ***thread*** que tiene bloqueado el objeto para activar el resto de threads que están esperando la liberación de un objeto. Un ***thread*** se convierte en propietario del bloqueo de un objeto ejecutando un método sincronizado del objeto. Los bloqueos de tipo clase, se consiguen ejecutando un ***método de clase sincronizado*** (***synchronized static***). Véanse las dos funciones siguientes, de las que ***put()*** inserta un dato y ***get()*** lo recoge:

```
public synchronized int get() {
     while (available == false) {
          try {
             // Espera a que put() asigne el valor y lo comunique con notify()
             wait();
          } catch (InterruptedException e) { }
     }
     available = false;
     // notifica que el valor ha sido leído
     notifyAll();
     // devuelve el valor
     return contents;
}
```

```
public synchronized void put(int value) {
     while (available == true) {
          try {
             // Espera a que get() lea el valor disponible antes de darle otro
             wait();
          } catch (InterruptedException e) { }
     }
     // ofrece un nuevo valor y lo declara disponible
     contents = value;
     available = true;
     // notifica que el valor ha sido cambiado
     notifyAll();
}
```

El bucle ***while*** de la función ***get()*** continúa ejecutándose (*avalaible == false*) hasta que el método ***put()*** haya suministrado un nuevo valor y lo indique con *avalaible = true*. En cada iteración del ***while*** la función ***wait()*** hace que el hilo que ejecuta el método ***get()*** se detenga hasta que se produzca un mensaje de que algo ha sido cambiado (en este caso con el método ***notifAll()*** ejecutado por ***put()***). El método ***put()*** funciona de forma similar.

Existe también la posibilidad de sincronizar una parte del código de un método sin necesidad de mantener bloqueado el objeto desde el comienzo hasta el final del método. Para ello se utiliza la palabra clave ***syncronized*** indicando entre paréntesis el objeto que se desea sincronizar (*synchronized(objetoASincronizar)*). Por ejemplo si se desea sincronizar el propio ***thread*** en una parte del método ***run()***, el código podría ser:

```
public void run() {
     while(true) {
          ...
          syncronized(this) { // El objeto a sincronizar es el propio thread
          ... // Código sincronizado
          }
          try {
              sleep(500); /* Se detiene el thread durante 0.5 segundos pero el
                        objeto es accesible por otros threads al no estar
                        sincronizado */
          } catch(InterruptedException e) {}
      }
}
```

Un ***thread*** puede llamar a un método sincronizado de un objeto para el cual ya posee el bloqueo, volviendo a adquirir el bloqueo. Por ejemplo:

```
public class VolverAAdquirir {

     public synchronized void a() {

          b();

          System.out.println("Estoy en a()");

     }

     public synchronized void b() {

          System.out.println("Estoy en b()");

     }

}
```

El anterior ejemplo obtendrá como resultado:

```
Estoy en b()
Estoy en a()
```

debido a que se ha podido acceder al objeto con el método ***b()*** al ser el ***thread*** que ejecuta el método ***a()*** "propietario" con anterioridad del bloqueo del objeto.

La sincronización es un proceso que lleva bastante tiempo a la CPU, luego se debe minimizar su uso, ya que el programa será más lento cuanta más sincronización incorpore.

## 5. PRIORIDADES.

Con el fin de conseguir una correcta ejecución de un programa se establecen ***prioridades*** en los ***threads***, de forma que se produzca un reparto más eficiente de los recursos disponibles. Así, en un determinado momento, interesará que un determinado proceso acabe lo antes posible sus cálculos, de forma que habrá que otorgarle más recursos (más tiempo de CPU). Esto no significa que el resto de procesos no requieran tiempo de CPU, sino que necesitarán menos. La forma de llevar a cabo esto es gracias a las prioridades.

Cuando se crea un nuevo ***thread***, éste hereda la prioridad del ***thread*** desde el que ha sido inicializado. Las prioridades viene definidas por variables miembro de la clase ***Thread***, que toman valores enteros que oscilan entre la máxima prioridad ***MAX_PRIORITY*** (normalmente tiene el valor 10) y la mínima prioridad ***MIN_PRIORITY*** (valor 1), siendo la prioridad por defecto ***NORM_PRIORITY*** (valor 5). Para modificar la prioridad de un ***thread*** se utiliza el método ***setPriority()***. Se obtiene su valor con ***getPriority()***.

El algoritmo de distribución de recursos en ***Java*** escoge por norma general aquel ***thread*** que tiene una prioridad mayor, aunque no siempre ocurra así, para evitar que algunos procesos queden "dormidos". Cuando hay dos o más ***threads*** de la misma prioridad (y además, dicha prioridad es la más elevada), el sistema no establecerá prioridades entre los mismos, y los ejecutará alternativamente dependiendo del sistema operativo en el que esté siendo ejecutado. Si dicho SO soporta el "time-slicing" (reparto del tiempo de CPU), como por ejemplo lo hace Windows 95/98/NT, los ***threads*** serán ejecutados alternativamente.

Un ***thread*** puede en un determinado momento renunciar a su tiempo de CPU y otorgárselo a otro ***thread*** de la misma prioridad, mediante el método ***yield()***, aunque en ningún caso a un ***thread*** de prioridad inferior.

## 6. GRUPOS DE THREADS.

Todo hilo de ***Java*** debe formar parte de un grupo de hilos (***ThreadGroup***). Puede pertenecer al grupo por defecto o a uno explícitamente creado por el usuario. Los grupos de ***threads*** proporcionan una forma sencilla de manejar múltiples ***threads*** como un solo objeto. Así, por ejemplo es posible parar varios ***threads*** con una sola llamada al método correspondiente. Una vez que un ***thread*** ha sido asociado a un ***threadgroup***, no puede cambiar de grupo.

Cuando se arranca un programa, el sistema crea un ***ThreadGroup*** llamado ***main***. Si en la creación de un nuevo ***thread*** no se especifica a qué grupo pertenece, automáticamente pasa a pertenecer al ***threadgroup*** del ***thread*** desde el que ha sido creado (conocido como ***current thread group*** y ***current thread***, respectivamente). Si en dicho programa no se crea ningún ***ThreadGroup*** adicional, todos los ***threads*** creados pertenecerán al grupo ***main*** (en este grupo se encuentra el método ***main()***).

Para conseguir que un **thread** pertenezca a un grupo concreto, hay que indicarlo al crear el nuevo ***thread***, según uno de los siguientes constructores:

*public Thread (ThreadGroup grupo, Runnable destino)*

*public Thread (ThreadGroup grupo, String nombre)*
*public Thread (ThreadGroup grupo, Runnable destino, String nombre)*

A su vez, un ***ThreadGroup*** debe pertenecer a otro ***ThreadGroup***. Como ocurría en el caso anterior, si no se especifica ninguno, el nuevo grupo pertenecerá al ***ThreadGroup*** desde el que ha sido creado (por defecto al grupo ***main***). La clase ***ThreadGroup*** tiene dos posibles constructores:

*ThreadGroup(ThreadGroup parent, String nombre);*

*ThreadGroup(String name);*

el segundo de los cuales toma como ***parent*** el ***threadgroup*** al cual pertenezca el ***thread*** desde el que se crea (***Thread.currentThread()***). Para más información acerca de estos constructores, dirigirse a la documentación del API de ***Java*** donde aparecen numerosos métodos para trabajar con ***grupos de threads*** a disposición del usuario (***getMaxPriority()***, ***setMaxPriority()***, ***getName()***, ***getParent()***, ***parentOf()***).

En la práctica los ***ThreadGroups*** no se suelen utilizar demasiado. Su uso práctico se limita a efectuar determinadas operaciones de forma más simple que de forma individual. En cualquier caso, véase el siguiente ejemplo:

```
ThreadGroup miThreadGroup = new ThreadGroup("Mi Grupo de Threads");

Thread miThread = new Thread(miThreadGroup, ”un thread para mi grupo");
```

donde se crea un grupo de ***threads*** (***miThreadGroup***) y un ***thread*** que pertenece a dicho ***grupo*** (***miThread)***.