## 

Contenido

[2.1. Recursos compartidos por los hilos. 2](#_Toc402941465)

[2.2. Estados de un hilo. Cambios de estado. 4](#_Toc402941466)

[2.3. Elementos relacionados con la programación de hilos. Librerías y clases. 7](#_Toc402941467)

[2.4. Creación, inicio y finalización. 16](#_Toc402941468)

[2.5. Información entre hilos. Intercambio. Gestión de hilos. 18](#_Toc402941469)

[2.6 Sincronización de hilos. 23](#_Toc402941470)

[2.7 Prioridades de los hilos. 33](#_Toc402941471)

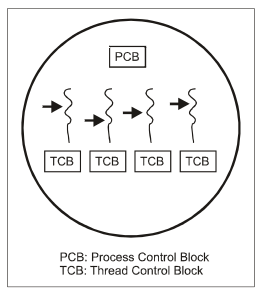
[2.8 Compartición de información entre hilos. 35](#_Toc402941472)

[2.9 Agrupación de hilos. 38](#_Toc402941473)

[2.10 Programación de aplicaciones multihilo. 42](#_Toc402941474)

## 2.1. Recursos compartidos por los hilos.

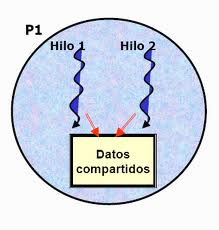
La Máquina Virtual Java (JVM) es un sistema multi-thread. Es decir, es capaz de ejecutar varias secuencias de ejecución (programas) simultáneamente. La JVM gestiona todos los detalles, asignación de tiempos de ejecución, prioridades, etc, de forma similar a como gestiona un Sistema Operativo múltiples procesos. **La diferencia básica entre un proceso de Sistema Operativo y un *Thread* Java es que los *Threads* corren dentro de la JVM, que es un proceso del Sistema Operativo y por tanto comparten todos los recursos, incluida la memoria y las variables y objetos allí definidos.** A este tipo de procesos donde se comparte los recursos se les llama a veces ***'procesos ligeros'*** (*lightweight process*). Por otro lado, un proceso pesado hijo podría sobrevivir sin la presencia del padre, no ocurre así con los hilos; la *‘muerte’* del padre supone la desaparición de los hilos hijos.



Java da soporte al concepto de *Thread* desde el mismo lenguaje, con algunas clases e interfaces definidas en el *package java.lang* y con métodos específicos para la manipulación de *Threads* en la clase *Object*.



Desde el punto de vista de las aplicaciones los *threads* son útiles porque permiten que el flujo del programa sea divido en dos o más partes, cada una ocupándose de alguna tarea. Por ejemplo un *Thread* puede encargarse de la comunicación con el usuario, mientras otros actúan en segundo plano, realizando la transmisión de un fichero, accediendo a recursos del sistema (cargar sonidos, leer ficheros ...), etc. De hecho todos los programas con interface gráfico (AWT o Swing) son *multithread* porque los eventos y las rutinas de dibujado de las ventanas corren en un *thread* distinto al principal.



Resumiendo, **cada hilo se ejecuta en forma estrictamente secuencial y tiene su propia pila, el estado de los registros de la CPU y su propio contador de programa. En cambio, comparten el mismo espacio de direcciones, lo que significa compartir también las mismas variables globales, el mismo conjunto de ficheros abiertos, procesos hijos (no hilos hijo), señales, semáforos y cualquier mecanismo de sincronización.**

**Un *thread* no puede existir independientemente de un programa, sino que se ejecuta dentro de un programa o proceso.**

**Compartición de datos**

Todos los hilos de proceso que pertenecen a un mismo proceso comparten un área común de datos que sirve para intercambiar información entre ellos. No es necesario acudir a técnicas de comunicación entre procesos tales como paso de mensajes, ya que todos los hilos son capaces de acceder directamente a los datos compartidos.

Por otro lado, la conmutación entre hilos de un mismo proceso es muy rápida, puesto que la cantidad de información que ha de ser salvada y/o restaurada por el sistema es mucho menor. Por eso, cuando se trata con hilos siempre se habla de cambios de contexto ligeros, en contraposición a los cambios de contexto pesados, que implican el manejo de procesos.

## 2.2. Estados de un hilo. Cambios de estado.

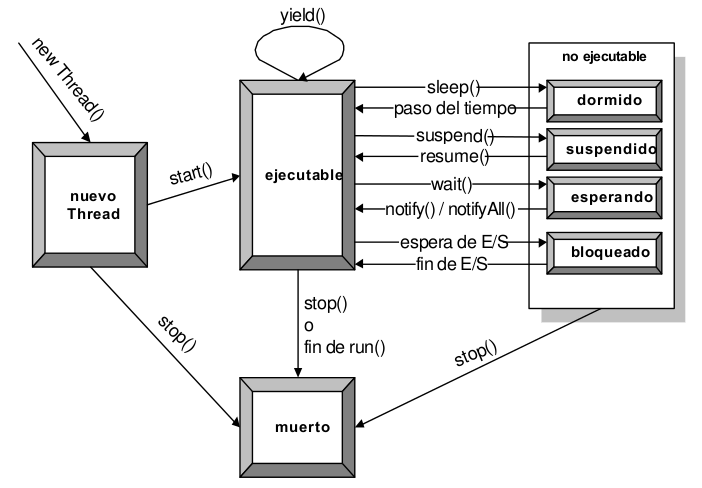
El comportamiento de un hilo depende del estado en que se encuentre, este estado define su modo de operación actual, por ejemplo, si está corriendo o no. A continuación proporcionamos la relación de estados en los que puede estar un hilo Java.

*• New*

*• Runnable*

*• Not running*

*• Dead*

****

**New**

Un hilo esta en el estado new la primera vez que se crea y hasta que el método *start* es llamado. Los hilos en estado new ya han sido inicializados y están listos para empezar a trabajar, pero aún no han sido notificados para que empiecen a realizar su trabajo.

**Runnable**

Cuando se llama al método *start* de un hilo nuevo, el método *run* es invocado y el hilo entra en el estado *runnable*. Este estado podría llamarse “*running*” porque la ejecución del método *run* significa que el hilo está corriendo. Sin embargo, debemos tener en cuenta la prioridad de los hilos. Aunque cada hilo está corriendo desde el punto de vista del usuario, en realidad todos los hilos, excepto el que en estos momentos está utilizando la CPU, están en el estado *runnable* (ejecutables, listos para correr) en cualquier momento dado. Uno puede pensar conceptualmente en el estado *runnable* como si fuera el estado “*running*”, sólo tenemos que recordar que todos los hilos tienen que compartir los recursos del sistema.

**Not running**

El estado *not running* se aplica a todos los hilos que están parados por alguna razón. Cuando un hilo está en este estado, está listo para ser usado y es capaz de volver al estado *runnable* en un momento dado. Los hilos pueden pasar al estado *not* *running* a través de varias vías.

A continuación se citan diferentes eventos que pueden hacer que un hilo esté parado de modo temporal.

• El método *suspend* ha sido llamado

• El método *sleep* ha sido llamado

• El método *wait* ha sido llamado

• El hilo está bloqueado por I/O

Para cada una de estas acciones que implica que el hilo pase al estado ***not running*** hay una forma para hacer que el hilo vuelva a correr. A continuación presentamos la lista de eventos correspondientes que pueden hacer que el hilo pase al estado *runnable*.

• Si un hilo está suspendido, la invocación del método *resume*

• Si un hilo está durmiendo, pasarán el número de milisegundos que se ha especificado que debe dormir

• Si un hilo está esperando, la llamada a *notify* o *notifyAll* por parte del objeto por el que espera

• Si un hilo está bloqueado por I/O, la finalización de la operación I/O en cuestión

**Dead**

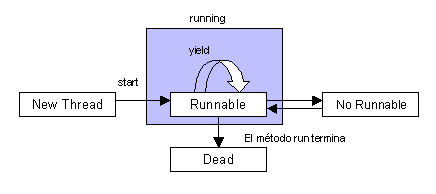
Un hilo entra en estado *dead* cuando ya no es un objeto necesario. Los hilos en estado *dead* no pueden ser resucitados y ejecutados de nuevo. Un hilo puede entrar en estado *dead* a través de dos vías:

• El método *run* termina su ejecución.

• El método *stop* es llamado.

La **primera opción** es el modo natural de que un hilo muera. Uno puede pensar en la muerte de un hilo cuando su método *run* termina la ejecución como una muerte por causas naturales.

En contraste a esto, está la muerte de un hilo “por causa” de su método *stop*. Una llamada al método *stop* mata al hilo de modo asíncrono.

****

Aunque **la segunda opción** suene un poco brusca, a menudo es muy útil. Por ejemplo, es bastante común que los *applets* maten sus hilos utilizando el método *stop* cuando el propio método *stop* del *applet* ha sido invocado. La razón de esto es que el método *stop* del *applet* es llamado normalmente como respuesta al hecho de que el usuario ha abandonado la página web que contenía el *applet* y no es adecuado dejar hilos de un *applet* corriendo cuando el *applet* no está activo, así que es deseable matar los hilos.

## 2.3. Elementos relacionados con la programación de hilos. Librerías y clases.

El lenguaje de programación Java proporciona soporte para hilos a través de una simple interfaz y un conjunto de clases. La interfaz de Java y las clases que incluyen funcionalidades sobre hilos son las siguientes:

*• Thread*

*• Runnable*

*• ThreadDeath*

*• ThreadGroup*

*• Object*

Todas estas clases son parte del paquete ***Java.lang.***

**2.3.1 Atributos y Métodos de la clase *Thread*.**

La clase *Thread* es la clase responsable de producir hilos funcionales para otras clases. Para añadir la funcionalidad de hilo a una clase simplemente se deriva la clase de *Thread* y se ignora el método *run*. Es en este método *run* donde el procesamiento de un hilo toma lugar, y a menudo se refieren a él como el cuerpo del hilo. La clase *Thread* también define los métodos *start* y stop, los cuales te permiten comenzar y parar la ejecución del hilo, además de un gran número de métodos útiles.

Al final de este documento, en Apéndice A: Funciones Miembro de la Clase *Thread* (se ha incluido una relación de métodos de esta clase con una breve descripción.

En las siguientes tablas se ven algunos atributos y métodos de la clase *Thread* que consideramos de interés para el curso. Este no es un listado exhaustivo.

**ATRIBUTOS**

**public static final int MIN\_PRIORITY**

La prioridad mínima que un hilo puede tener. Vale 1.

**public static final int NORM\_PRIORITY**

La prioridad por defecto que se le asigna a un hilo. Vale 5.

**public static final int MAX\_PRIORITY**

La prioridad máxima que un hilo puede tener. Vale 10.

**CONSTRUCTORES**

**public Thread ()**

Crea un nuevo objeto *Thread*. Este constructor tiene el mismo efecto que *Thread* (null, null, gname), donde gname es un nombre generado automáticamente y que tiene la forma "*Thread*-"+n, donde n es un entero asignado consecutivamente.

**public Thread (String name)**

Crea un nuevo objeto *Thread*, asignándole el nombre *name*.

**public Thread (Runnable target)**

Crea un nuevo objeto *Thread*. Target es el objeto que contie­ne el método *run*() que será invocado al lanzar el hilo con *start*().

**public Thread (Runnable target, String name)**

Crea un nuevo objeto *Thread*, asignándole el nombre *name*. Target es el objeto que contiene el método *run*() que será invocado al lanzar el hilo con *start*().

**MÉTODOS**

**public static Thread currentThread ()**

Retorna una referencia al hilo que se está ejecutando actualmente.

**public static void dumpStack ()**

Imprime una traza del hilo actual. Usado sólo con propósitos de depuración.

**public String getName ()**

Retorna el nombre del hilo.

**int getPriority ()**

Retorna la prioridad del hilo.

**public final boolean isAlive ()**

Chequea si el hilo está vivo. Un hilo está vivo si ha sido lanzado con *start* y no ha muerto todavía.

**public final void isDaemon ()**

Devuelve verdadero si el hilo es *daemon*.

**public foral void join () throws InterruptedException**

Espera a que este hilo muera.

**public final void join (long millis) throws InterruptedException**

Espera como mucho *millis* milisegundos para que este hilo muera.

**public final void join (long millis, int nonos) throws InterruptedException**

Permite afinar con los nanosegundos nanos el tiempo a esperar.

**public void resume ()**

Se usa para recomenzar un hilo que está suspendido.

NOTA: resume=reanudar.

**public void run ()**

Si este hilo fue construido usando un objeto que implementaba *Runnable*, entonces el método *run* de ese objeto es llamado. En cualquier otro caso este método no hace nada y retorna.

**public final void setDaemon (boolean on)**

Marca este hilo como *daemon* si el parámetro *on* es verdadero o como hilo de usuario si es falso. El método debe ser llamado antes de que el hilo sea lanzado.

**public foral void setName (String name)**

Cambia el nombre del hilo por *name*.

**public final void setPriority (int newPriority)**

Asigna la prioridad *newPriority* a este hilo.

NOTA: acepta valores de 1 a 10.

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

Hace que el hilo que se está ejecutando actualmente cese su ejecución por los milisegundos especificados en milisegundos. Pasa al estado dormido. El hilo no pierde la propiedad de ningún cerrojo que tuviera adquirido con *synchronized*.

**public static void sleep (long millis, int manos) throws InterruptedException**

Permite afinar con los nanosegundos nanos el tiempo a estar dormido.

**public void start ()**

Hace que el hilo comience su ejecución. La MVJ llamará al método *run* de este hilo.

**public void stop ()**

Hace que el hilo termine su ejecución.

**public void suspend ()**

Hace que el hilo interrumpa temporalmente su ejecución.

**public String toString ()**

Devuelve una representación en forma de cadena de este hilo, incluyendo su nombre, su prioridad y su grupo.

**public static void yield ()**

Hace que el hilo que se está ejecutando actualmente pase al estado listo, permitiendo a otro hilo ganar el procesador.

NOTA: yield=ceder el paso.

Por otro lado, los métodos de la clase *Thread* se pueden agrupar, **según su uso**, como:

**Métodos de clase**

Estos son los métodos estáticos que deben llamarse de manera directa en la clase *Thread*.

**currentThread**: el método estático devuelve el objeto *Thread* que representa al hilo que se ejecuta actualmente.

**yield**: este método hace que el intérprete cambie de contexto entre el hilo actual y el siguiente hilo ejecutable disponible. Es una manera de asegurar que los hilos de menor prioridad no sufran inanición.

**Sleep(int n):** el método *sleep* provoca que el intérprete ponga al hilo en curso a dormir durante n milisegundos. Una vez transcurridos los n milisegundos, dicho hilo volverá a estar disponible para su ejecución. Los relojes asociados a la mayor parte de los intérpretes Java nos serán capaces de obtener precisiones mayores de 10 milisegundos.

**activeCount():** este método devuelve el número de *threads* activos dentro del grupo donde está incluido el *thread* en ejecución.

**Métodos de instancia**

**start**: indica al intérprete de Java que cree un contexto de hilo del sistema y comience a ejecutarlo. A continuación, el método *run* objeto de este hilo será llamado en el nuevo contexto del hilo. Debe tomarse la precaución de no llamar al método *start* más de una vez sobre un objeto hilo dado.

**run**: constituye el cuerpo de un hilo en ejecución. Este es el único método de la interfaz *Runnable*. Es llamado por el método *start* después de que el hilo apropiado del sistema se haya inicializado. Siempre que el método *run* devuelva el control, el hilo actual se detendrá.

**stop**: provoca que el hilo se detenga de manera inmediata. A menudo constituye una manera brusca de detener un hilo, especialmente si este método se ejecuta sobre el hilo en curso. En tal caso, la línea inmediatamente posterior a la llamada al método stop no llega a ejecutarse jamás, pues el contexto del hilo muere antes de que stop devuelva el control.

**suspend**: es distinto de stop. *suspend* toma el hilo y provoca que se detenga su ejecución sin destruir el hilo de sistema subyacente, ni el estado del hilo anteriormente en ejecución. Si la ejecución de un hilo se suspende, puede llamarse a resume sobre el mismo hilo para lograr que vuelva a ejecutarse de nuevo.

**resume**: se utiliza para revivir un hilo suspendido.

**setPriority(int p):** asigna al hilo la prioridad indicada por el valor entero pasado como parámetro.

**getPriority**: devuelve la prioridad del hilo en curso, que es un valor entre 1 y 10.

**setName(String nombre):** identifica al hilo con un nombre mnemónico. De esta manera se facilita la depuración de programas multihilo.

**getName:** devuelve el valor actual, de tipo cadena, asignado como nombre al hilo mediante *setName*.

**2.3.2 La clase Runnable.**

Java no soporta herencia múltiple de forma directa, es decir, no se puede derivar una clase de varias clases padre. Esto nos plantea la duda sobre cómo podemos añadir la funcionalidad de Hilo a una clase que deriva de otra clase, siendo ésta distinta de *Thread*. Para lograr esto se utiliza la interfaz *Runnable*. La interfaz *Runnable* proporciona la capacidad de añadir la funcionalidad de un hilo a una clase simplemente implementando la interfaz, en lugar de derivándola de la clase *Thread*.

Las clases que implementan la interfaz *Runnable* proporcionan un método *run* que es ejecutado por un objeto hilo asociado que es creado aparte. Esta es una herramienta muy útil y a menudo es la única salida que tenemos para incorporar multihilo dentro de las clases. Esta cuestión será tratada más ampliamente en el apartado de creación de hilos.

**2.3.3 Clase ThreadDead.**

La clase de error *ThreadDeath* proporciona un mecanismo que permite hacer limpieza después de que un hilo haya sido finalizado de forma asíncrona. Se llama a *ThreadDeath* una clase error porque deriva de la clase Error, la cual proporciona medios para manejar y notificar errores. Cuando el método stop de un hilo es invocado, una instancia de *ThreadDeath* es lanzada por el moribundo hilo como un error. Sólo se debe recoger el objeto *ThreadDeath* si se necesita para realiza una limpieza específica a la terminación asíncrona, lo cual es una situación bastante inusual. Si se recoge el objeto, debe ser relanzado para que el hilo realmente muera.

**2.3.4 Atributos y métodos de la clase ThreadGroup.**

La clase ThreadGroup es la implementación del concepto de grupo de hilos en Java. Ofrece, por tanto, la funcionalidad necesaria para la manipulación de grupos de hilos para las aplicaciones Java. Un objeto ThreadGroup puede contener cualquier número de hilos. Los hilos de un mismo grupo generalmente se relacionan de algún modo, ya sea por quién los creó, por la función que llevan a cabo, o por el momento en que deberían arrancarse y parar.

El grupo de hilos de más alto nivel en una aplicación Java es el grupo de hilos denominado *main.*

La clase *ThreadGroup* tiene métodos que pueden ser clasificados como sigue:

• ***Collection Managment Methods*** (Métodos de administración del grupo): métodos que manipulan la colección de hilos y subgrupos contenidos en el grupo de hilos.

• ***Methods That Operate on the Group*** (Métodos que operan sobre el grupo): estos métodos establecen u obtienen atributos del objeto *ThreadGroup*.

• ***Methods That Operate on All Threads within a Group*** (Métodos que operan sobre todos los hilos dentro del grupo): este es un conjunto de métodos que desarrollan algunas operaciones, como inicio y reinicio, sobre todos los hilos y subgrupos dentro del objeto *ThreadGroup*.

• ***Access Restriction Methods*** (Métodos de restricción de acceso): *ThreadGroup* y *Thread* permiten al administrador de seguridad restringir el acceso a los hilos en base a la relación de miembro/grupo con el grupo.

**Métodos de administración del grupo**

La clase *ThreadGroup* proporciona un conjunto de métodos que manipulan los hilos y los subgrupos que pertenecen al grupo y permiten a otros objetos solicitar información sobre sus miembros. Por ejemplo, puedes llamar al método *activeCount* de *ThreadGroup* para conocer el número de hilos activos que actualmente hay en el grupo. El método *activeCount* se usa generalmente con el método *enumerate* para obtener un vector (array) que contenga las referencias a todos los hilos activos en un *ThreadGroup*. Por ejemplo, el método *listCurrentThreads* en el siguiente ejemplo rellena un vector con todos los hilos activos en el grupo de hilos actual e imprime sus nombres:

public class EnumerateTest {

public void listCurrentThreads() {

ThreadGroupcurrentGroup=

Thread.currentThread().getThreadGroup();

Int numThreads =

currentGroup.activeCount();

Thread[] listOfThreads = new

Thread[numThreads];

currentGroup.enumerate(listOfThreads);

for (int i = 0; i < numThreads;

i++)

System.out.println("Thread #"

+ i + " = " +

listOfThreads[i].getName()); } }

Otro conjunto de métodos de administración del grupo proporcionado por la clase *ThreadGroup* incluye los métodos *activeGroupCount* y *list*.

**Métodos que operan sobre el grupo**

La clase *ThreadGroup* da soporte a varios atributos que son establecidos y recuperados de un grupo de forma global (hacen referencia al concepto de grupo, no a los hilos individualmente).

Se incluyen atributos como la prioridad máxima que cualquiera de los hilos del grupo puede tener, el carácter “*daemon*” o no del grupo, el nombre del grupo, y el padre del grupo.

Los métodos que recuperan y establecen los atributos de *ThreadGroup* operan a nivel de grupo. Consultan o cambian el atributo del objeto de la clase *ThreadGroup*, pero no hacen efecto sobre ninguno de los hilos pertenecientes al grupo. La siguiente es una lista de métodos de *ThreadGroup* que operan a nivel de grupo:

- *getMaxPriority y setMaxPriority*

*- getDaemon y SetDaemon*

*- getName*

*- getParent y parentOf*

*- toString*

Por ejemplo, cuando utilizas setMaxPriority para modificar la máxima prioridad de un grupo, sólo estás modificando el atributo para el objeto grupal; no estás modificando, individualmente, la prioridad de ninguno de los hilos definidos dentro del grupo. Esto puede dar lugar a situaciones peculiares: si rebajamos la prioridad máxima del grupo después de asignar la prioridad de un hilo miembro, podría quedar el hilo con mayor prioridad que la máxima del grupo.

**Métodos que operan sobre todos los hilos de un grupo**

La clase ThreadGroup tiene tres métodos que te permiten modificar el estado actual de todos los hilos pertenecientes al grupo:

*- resume*

*- stop*

*- suspend*

Estos métodos suponen el cambio correspondiente de estado para todos y cada uno de los hilos del grupo, así como los de sus subgrupos. No se aplican, por tanto, a un nivel de grupo, sino que se aplican individualmente a todos los miembros.

**Métodos de restricción de acceso**

La clase *ThreadGroup* no impone ninguna restricción de acceso por sí sola, como permitir a los hilos de un grupo consultar o modificar los hilos de un grupo diferente. En lugar de esto, las clases *Thread* y *ThreadGroup* cooperan con los administradores de seguridad (subclases de la clase *SecurityManager*), la cual impone las restricciones de acceso basándose en la pertenencia de los hilos a los grupos.

Ambas clases, *Thread* y *ThreadGroup*, tienen un método, *checkAccess*, la cual invoca al método *checkAccess* del administrador de seguridad actual. El administrador de seguridad decide si debe permitir el acceso basándose en la relación de pertenencia entre el grupo y los hilos implicados. Si no se permite el acceso, le método *checkAccess* lanza una excepción de tipo *SecurityException*. En otro caso, *checkAccess* simplemente retorna.

La siguiente es una lista de métodos de *ThreadGroup* que invocan al método *checkAccess* de la propia clase antes de llevar a cabo la acción del método en cuestión.

Estos son los que se conocen como accesos regulados, esto es, accesos que deben ser aprobados por el administrador de seguridad antes de completarse.

*- ThreadGroup (ThreadGroup parent, String name)*

*- setDameon(boolena isDaemon)*

*- setMaxPriority(int maxPriority)*

*- stop*

*- suspend*

*- resume*

*- destroy*

Igualmente, los siguientes métodos de la clase *Thread* invocan al método *checkAccess* de la clase antes de completarse la acción que desarrollan:

*- constructores que especifican un grupo de hilos*

*- stop*

*- supend*

*- resume*

*- setPriority(int priority)*

*- setName(String name)*

*- setDaemon(boolean isDaemon)*

Una aplicación “*stand-alone*” de Java no tiene, por defecto, un administrador de seguridad; no se impone ninguna restricción de acceso y cualquier hilo puede consultar o modificar cualquier otro hilo, sea cual sea el grupo en el que se encuentren. Puedes definir e implementar tus propias restricciones de acceso para los grupos de hilos mediante la creación de subclases de la clase *SecurityManager*, rescribiendo los métodos apropiados, e imponiendo ese *SecurityManager* como el actual administrador de seguridad de tu aplicación.

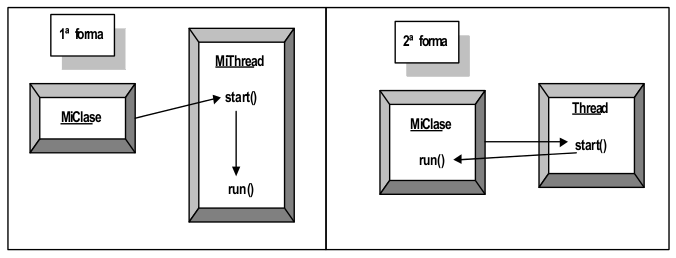
**2.3.5 Clase Object.**

Aunque, estrictamente hablando, no es una clase de apoyo a los hilos, la clase objeto proporciona unos cuantos métodos cruciales dentro de la arquitectura multihilo de Java. Estos métodos son ***wait, notify*** y ***notifyAll***. El método *wait* hace que el hilo de ejecución espere en estado dormido hasta que se le notifique que continúe. Del mismo modo, el método *notify* informa a un hilo en espera de que continúe con su ejecución. El método *notifyAll* es similar a *notify* excepto que se aplica a todos los hilos en espera. Estos tres métodos solo pueden ser llamados desde un método o bloque sincronizado (o bloque de sincronización).

## 2.4. Creación, inicio y finalización.

En Java, los hilos comparten el mismo espacio de memoria. Incluso comparten gran parte del entorno de ejecución, de modo que la creación de nuevos hilos es mucho más rápida que la creación de nuevos procesos. La ventaja que proporcionan los hilos es la capacidad de tener más de un camino de ejecución en un mismo programa. Así, con un único proceso, ejecutándose una JVM (Java Virtual Machine), habrá siempre más de un hilo, cada uno con su propio camino de ejecución.

En cuanto al proceso de **creación de hilos**, son **dos los mecanismos** que nos permiten llevarlo a cabo en Java: **extendiendo la clase *Thread****, o,* **implementando la interfaz *Runnable*.**



**Lo más habitual es crear hilos implementando la interfaz *Runnable***, dado que las interfaces representan una forma de encapsulamiento del trabajo que una clase debe realizar. Así, se utilizan para el diseño de requisitos comunes a todas las clases que se tiene previsto implementar. La interfaz define el trabajo, la funcionalidad que debe cubrirse, mientras que la clase o clases que implementan la interfaz realizan dicho trabajo (cumplen esa funcionalidad). Todas las clases o grupos de clases que implementen una cierta interfaz deberán seguir las mismas reglas de funcionamiento.

**El otro mecanismo de creación de hilos, como ya hemos dicho, consistiría en la creación previa de una subclase de la clase *Thread***, la cual podríamos instanciar después.

Por ejemplo:

class MiThread extends Thread {

public void run() {

. . .

}

}

Se corresponde con la declaración de una clase, *MiThread*, que extiende la clase *Thread*, sobrecargando el método *Thread.run* heredado con su propia implementación.

Es en el método *run* donde se implementa el código correspondiente a la acción (la tarea) que el hilo debe desarrollar. El método *run* no es invocado directa o explícitamente (a menos que no quieras que se ejecute dentro de su propio hilo). En lugar de esto, los hilos se arrancan con el método *start*, se suspenden con el método *suspend*, se reanudan con el método resume, y se detienen con el método stop (el cual supone también la muerte del hilo y la correspondiente excepción *ThreadDeath*), como ya explicaremos en el apartado de Estado y Control de Hilos. Un hilo suspendido puede reanudarse en la instrucción del método *run* en la que fue suspendido.

En el caso de crear un hilo extendiendo la clase *Thread*, se pueden heredar los métodos y variables de la clase padre. Si es así, una misma subclase solamente puede extender o derivar una vez de la clase padre *Thread*. Esta limitación de Java puede ser superada a través de la implementación de *Runnable*. Veamos el siguiente ejemplo:

public class MiThread implements Runnable {

Thread t;

public void run() {

// Ejecución del thread una vez creado

}

}

En este caso necesitamos crear una instancia de *Thread* antes de que el sistema pueda ejecutar el proceso como un hilo. Además, el método abstracto *run* que está definido en la interfaz *Runnable* tiene que implementarse en la nueva clase creada.

La **diferencia entre ambos métodos de creación de hilos** en Java radica en la flexibilidad con que cuenta el programador, que es mayor en el caso de la utilización de la interfaz *Runnable*. Sobre la base del ejemplo anterior, se podría extender la clase *MiThread* a continuación, si fuese necesario. La mayoría de las clases creadas que necesiten ejecutarse como un hilo implementarán la interfaz *Runnable*, ya que así queda cubierta la posibilidad de que sean extendidas por otras clases.

Por otro lado, es un error pensar que la interfaz *Runnable* está realizando alguna tarea mientras un hilo de alguna clase que la implemente se está ejecutando. Es una interfaz, y como tal, sólo contiene **funciones abstractas** (concretamente una única, *run*), proporcionando tan solo una idea de diseño, de infraestructura, de la clase *Thread*, pero ninguna funcionalidad. Su declaración en Java tiene el siguiente aspecto:

package Java.lang;

public interfaz Runnable {

public abstract void run() ;

}

## 2.5. Información entre hilos. Intercambio. Gestión de hilos.

**2.5.1. Paso de parámetros a los hilos.**

El método *run()* no toma argumentos, sin embargo, en muchas ocasiones es necesario pasarle parámetros a un *thread*. Esto se puede conseguir encapsulando el *thread* en una clase y pasando los parámetros al constructor de la misma para almacenarlos en variables locales. En el siguiente ejemplo consideramos una aplicación que crea un nuevo esclavo en un *thread* por cada nuevo trabajo que le llega.

class Parametros implements Runnable

{

private int parm1;

private int parm2;

private Thread me;

public Parametros (int p1, int p2, String nombre)

{

parm1 = p1;

parm2 = p2;

me = new Thread (this,nombre);

me.start();

}

public void run()

{

if (me == Thread.currentThread())

{

System.out.println(Thread.currentThread().getName()

+"\n\tp1 = "+ parm1 + "\n\tp2 = " + parm2);

}

}

}

public static void main (String[] args)

{

Parametros p1 = new Parametros(4,5,"Hilo 1");

Parametros p2 = new Parametros(7,9,"Hilo 2");

}

Esto imprimiría:

*Hilo 1*

*p1 = 4*

*p2 = 5*

*Hilo 2*

*p1 = 7*

*p2 = 9*

**2.5.2. Operaciones básicas sobre *Threads*.**

Existen una serie de métodos auxiliares que nos permiten saber el estado del hilo y modificar el mismo.

***public static Thread currentThread();***

Devuelve el *thread* que se está ejecutando *acutalmente*. Ver ejemplo enel punto siguiente.

***public final boolean isAlive();***

Devuelve ***false*** si el *thread* está en el estado nuevo *thread* o muerto y ***true*** en caso contrario.

***Ejemplo:***

class MiThread extends Thread {

public void run() {

try {

sleep(2000);

} catch (InterruptedException e) { }

}

}

class Vivo {

public static void main(String args[]) {

MiThread t = new MiThread();

System.out.println("isAlive() antes de iniciar: "+

t.isAlive());

t.start();

System.out.println("isAlive() en ejecución: "+

t.isAlive());

try {

// Hace "dormir" al thread actual

Thread.currentThread().sleep(3000);

// de esta forma, le da tiempo a terminar al

// thread t

} catch (InterruptedException e) { }

System.out.println("isAlive() después de ejecución: "+

t.isAlive());

}

}

La salida correspondiente al programa es:

*Estado antes de iniciar: false*

*Estado en ejecución: true*

*Estado después de ejecución: false*

***public static void yield();***

Transfiere el control al siguiente *Thread* en el *scheduler* (con la misma prioridad que el actual) que se encuentre en estado ejecutable.

***Ejemplo:***

class UnThreadDosInstanciasAmables {

public static void main(String args[] ) {

SiNoThreadAmable s = new SiNoThreadAmable("SI");

SiNoThreadAmable n = new SiNoThreadAmable("NO");

s.start();

n.start();

}

}

class SiNoThreadAmable extends Thread {

private String SiNo;

static int Contador=0;

public SiNoThreadAmable(String s) {

super();

SiNo=s;

}

public void run() {

int i;

for (i=1;i<=20; i++) {

System.out.print(++Contador+":"+SiNo+" ");

yield();

}

}

}

En este caso hay dos *threads* que se ejecutan concurrentemente. Para cada salida por pantalla, (*System.out.println(++Contador+”:”SiNo+” “);* ) el *thread* cede la CPU al otro *thread* mediante la llamada a *yield().* Esto no significa que el otro *thread* continúe su ejecución hasta llegar a su propio *yield(),* ya que puede perder la CPU antes y por consiguiente no tienen porqué salir los Síes y Noes alternativamente, pero sí que está más repartida la CPU:

*1:NO 2:SI 3:SI 4:NO 5:SI 6:NO 7:NO 8:SI 9:NO 10:SI 11:NO 12:NO 14:NO 13:SI 15:NO 16:SI 17:SI 18:NO 19:SI 20:NO 21:SI 22:NO 23:SI 24:SI 25:NO 26:SI 27:SI 28:NO 29:NO 30:NO 31:SI 32:NO 33:SI 34:SI 35:SI 36:NO 37:SI 38:NO 39:SI 40:NO*

**2.5.3. Join.**

***public final void join() throws InterruptedException;***

Hace que el *thread* que se está ejecutando actualmente pase al estado “*esperando*” indefinidamente hasta que muera el *thread* sobre el que se realiza el join().

***Ejemplo:***

class MiThread extends Thread {

public void run() {

int i;

for (i=1;i<=100; i++)

System.out.print(i+" ");

}

}

class Join1 {

public static void main (String args[]) throws InterruptedException {

MiThread t = new MiThread();

t.start();

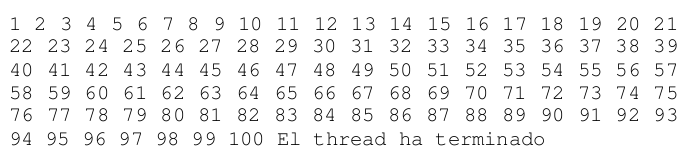
t.join();

System.out.println("El thread ha terminado");

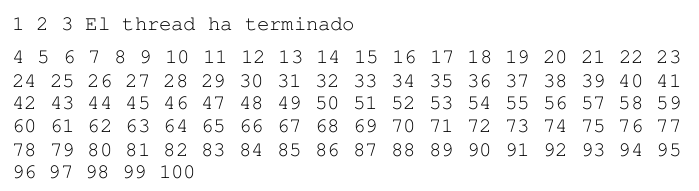
}

}

El *thread* de la clase Join1 espera indefinidamente hasta que finalice el *thread t*. La salida por pantalla es:



Si no se hubiera realizado el *t.join(),* el *thread* de la clase *Join1* no habría esperado la finalización de t y la salida por pantalla habría sido otra:



Existen dos métodos join() más:

***public final synchronized void join(long miliseg) throws InterruptedException;***

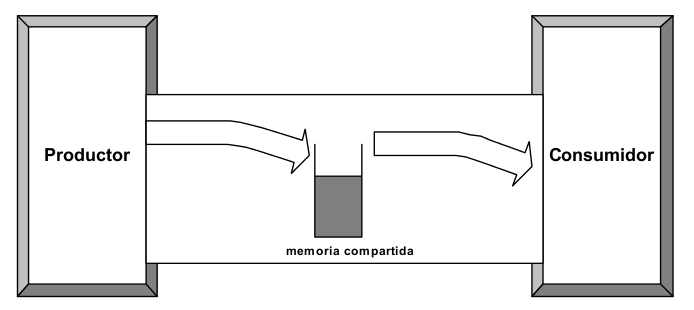
***public final synchronized void join(long miliseg, int nanoseg) throws InterruptedException;***

En estos dos casos, el *thread* actual no espera indefinidamente sino que reinicia su ejecución en el instante en que se finalice el *thread* sobre el que se hace el *join()* o pase el tiempo especificado por los parámetros *miliseg* (milisegundos) y opcionalmente *nanoseg* (nanosegundos).

## 2.6 Sincronización de hilos.

Hasta el momento se ha estado tratando con *threads* asíncronos, cada uno de ellos se ejecutaba independientemente de los demás, que se ejecutaban concurrentemente. Existen, sin embargo, situaciones en las que es imprescindible sincronizar en cierta forma la ejecución de distintos *threads* que buscan un objetivo común, cooperando entre ellos para su correcto funcionamiento.

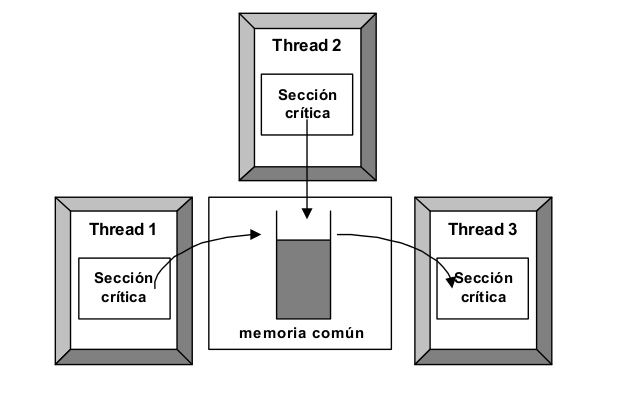
El paradigma productor/consumidor consiste en la ejecución de un *thread* que produce un flujo de datos que deben ser consumidos por otro *thread*. Este flujo de datos puede ser almacenado en una zona de memoria compartida por ambos *threads*. En este caso, la sincronización es vital, ya que de lo contrario, si el productor genera los datos más deprisa de lo que el consumidor es capaz de tratarlos, puede producirse un desbordamiento de la memoria asignada al efecto; por otra parte, si el consumidor es más rápido que el productor, puede encontrarse con que no hay datos que consumir o que se consuman más de una vez (si no elimina los datos consumidos, sino que los elimina el productor).



En este caso, debe existir un mecanismo que “frene” al productor o al consumidor en los casos necesarios.

**2.6.1 Sección crítica.**

Se llama sección crítica a los segmentos de código, dentro de un programa, que acceden a zonas de memoria comunes desde distintos *threads* que se ejecutan concurrentemente.



En Java, las secciones críticas se marcan con la palabra reservada *synchronized*. Aunque está permitido marcar bloques de código más pequeños que un método como *synchronized*, para seguir una buena metodología de programación, es preferible hacerlo a nivel de método.

Ejemplo, si tenemos las siguientes clases:

public class Contador {

private int valor=0;

//public synchronized void incrementa() {

public void incrementa() {

int aux;

aux = valor;

aux++;

valor = aux;

//valor++;

}

public long getValor() {

return valor;

}

public class Contable extends Thread {

Contador contador;

public Contable (Contador c, String nombre) {

contador=c;

this.setName(nombre);

}

public void run () {

int i;

long aux;

for (i=1;i<=10000; i++) {

contador.incrementa();

}

System.out.println("Contado hasta ahora ("+ this.getName() + "): " + contador.getValor());

}

}

public class EjemploSynchronized {

public static void main(String[] args) {

Contable c1 , c2 ;

Contador c = new Contador();

c1 = new Contable(c, "C1");

c2 = new Contable(c, "C2");

c1.start();

c2.start();

}

}

Para realizar el incremento en el método ***incrementa(),*** a efectos didácticos, se ha empleado una variable auxiliar intermedia (*aux*), aunque el efecto de haber codificado *valor++* habría sido similar.

En este caso, la sección crítica es la línea: ***contador.incrementa()***; ya que desde esta instrucción se accede a un objeto (contador) que es compartido por ambos *threads* (c1 y c2).

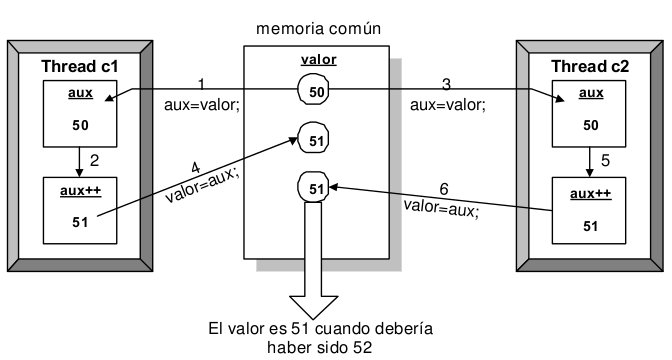
La salida por pantalla es:

Contado hasta ahora: 124739

Contado hasta ahora: 158049

Cuando debería haber sido: en la primera línea un número comprendido entre 100000 y 200000; y en la segunda línea el valor 200000.

Lo que ha ocurrido en realidad para que se produzca este resultado equivocado es lo siguiente:



Evidentemente, esto mismo ha ocurrido un gran número de veces durante la ejecución de los *threads*.

Simplemente cambiando la línea de código:

***public void incrementa() {***

por

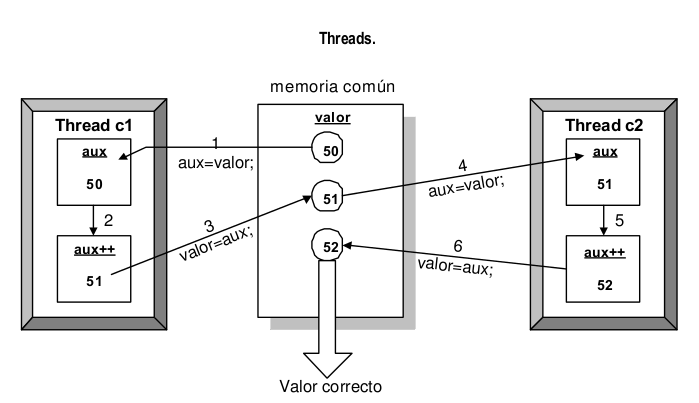
***public synchronized void incrementa() {***

Se habría conseguido bloquear el método incrementa, de forma que no pudieran ejecutarlo simultáneamente dos *threads* sobre el mismo objeto (contador).

La salida del programa habría sido (correcta):

Contado hasta ahora: 186837

Contado hasta ahora: 200000



**2.6.2. Wait y Notify.**

En realidad, el método wait() tiene varios hermanos, y el método notify() también. Todos ellos son finales:

**wait(long milis):** espera hasta una notificación, o hasta que pasen <milis> milisegundos. Si el parámetro es 0, la espera es infinita.

**wait(long tiempo, int nanos):** igual que la anterior, pero con precisión de nanosegundos.

**wait():** igual que wait(0). Es la explicada anteriormente.

**notify():** despierta a una hebra de las que están esperando.

**notifyAll():** despierta a todas las hebras que están esperando.

Todas las variantes de *wait(...)* pueden lanzar la excepción *InterruptedException*, al igual que ocurría con *sleep().* La excepción se lanzará si se llama al método *interrupt()* de la hebra que llamó al *wait(...).*

*notify()* y *notifyAll()* son semejantes, con la diferencia de que *notifyAll()* despierta a todas las hebras. Además, *notify()* es atómica, mientras que *notifyAll()* no. Al llamar a esta última, todas las hebras suspendidas pasarán a estar preparadas para la ejecución, y competirán todas por conseguir el cerrojo que liberaron en la llamada a *wait(),* sin que exista ningún tipo de preferencia por ninguna de ellas.

Se aconseja más el uso de *notifyAll()* que el de *notify().* Si el programa está diseñado de tal forma que se asegura que sólo una hebra estará esperando a un *notify()* cual se use es indiferente. Sin embargo si no es así, es preferible utilizar *notifyAll().* Más aún, es aconsejable utilizar siempre *notifyAll(),* pues no puede saberse con seguridad si en el futuro siempre habrá una única hebra, o será necesario modificar el comportamiento y cabrá la posibilidad de que haya más. Si en un momento dado hay más de una hebra y fuera necesario un *notifyAll()* en lugar de un *notify()* podrían aparecer problemas bastante difíciles de entender.

Realmente el uso de *notifyAll()* está pensado para la posibilidad de que se utilice la espera para que se cumplan más de una condición. Podríamos tener una hebra esperando a que se cumpla una condición, y otra esperando a que se cumpla una diferente. Si se llama a *notify()* cuando un método hace cierta una de esas condiciones, la máquina virtual podría despertar a la hebra que está a la espera del cumplimiento de la otra condición, y al no cumplirse volvería a llamar a *wait(),* y la hebra que podría ya ejecutarse seguirá dormida, quizá indefinidamente. Naturalmente en este caso es aún más importante el uso de la estructura *while(!condicion) wait();* en lugar de *if(!condicion) wait()*

En este ejemplo vamos a tener 3 clases: una clase *Main*, una clase *Saludo* y una clase *Personal* (Hilo).

Clase Main:

public class Main {

public static void main(String[] args) {

// Objeto en comun, se encarga del wait y notify

Saludo s = new Saludo();

/\*Instancio los hilos y le paso como parametros:

\*

\* El Nombre del Hilo(en este caso es el nombre del personal)

\* ----El objeto en comun (Saludo)----

\* Un booleano para verificar si es jefe o empleados

\*

\*/

Personal Empleado1 = new Personal("Pepe", s, false);

Personal Empleado2 = new Personal("José", s, false);

Personal Empleado3 = new Personal("Pedro", s, false);

Personal Jefe1 = new Personal("JEFE", s, true);

//Lanzo los hilos

Empleado1.start();

Empleado2.start();

Empleado3.start();

Jefe1.start();

}

}

Clase Personal:

import java.util.logging.Level;

import java.util.logging.Logger;

public class Personal extends Thread{

String nombre;

Saludo saludo;

boolean esJefe;

public Personal(String nombre, Saludo salu, boolean esJefe){

this.nombre = nombre;

this.saludo = salu;

this.esJefe = esJefe;

}

public void run(){

System.out.println(nombre + " llegó.");

try {

Thread.sleep(1000);

//Verifico si es personal que esta es jefe o no

if(esJefe){

saludo.saludoJefe(nombre);

}else{

saludo.saludoEmpleado(nombre);

}

} catch (InterruptedException ex) {

Logger.getLogger(Personal.class.getName()).log(Level.SEVERE, null, ex);

}

}

}

Clase Saludo:

import java.util.logging.Level;

import java.util.logging.Logger;

public class Saludo {

public Saludo(){

}

/\* Si no es jefe, el empleado va a quedar esperando a que llegue el jefe

Se hace wait de el hilo que esta corriendo y se bloquea, hasta que

se le avise que ya puede saludar\*/

public synchronized void saludoEmpleado(String nombre){

try {

wait();

System.out.println("\n"+nombre.toUpperCase() + "-: Buenos días jefe.");

} catch (InterruptedException ex) {

Logger.getLogger(Saludo.class.getName()).log(Level.SEVERE, null, ex);

}

}

//Si es jefe, saluda y luego avisa a los empleados para que saluden

// El notifyAll despierta a todos los hilos que esten bloqueados

public synchronized void saludoJefe(String nombre){

System.out.println("\n\*\*\*\*\*\* "+nombre + "-: Buenos días empleados. \*\*\*\*\*\*");

notifyAll();

}

}

Salida del programa por consola:

*Pepe llegó.*

*José llegó.*

*Pedro llegó.*

*JEFE llegó.*

*\*\*\*\*\*\* JEFE-: Buenos días empleados. \*\*\*\*\*\**

*PEPE-: Buenos días jefe.*

*JOSÉ-: Buenos días jefe.*

*PEDRO-: Buenos días jefe.*

**2.6.3. Semáforos.**

Un semáforo binario es un indicador (S) de condición que registra si un recurso está disponible o no. Un semáforo binario sólo puede tomar dos valores: 0 y 1. Si, para un semáforo binario, S = 1 entonces el recurso está disponible y la tarea lo puede utilizar; si S = 0 el recurso no está disponible y el proceso debe esperar.

Los semáforos se implementan con una cola de tareas o de condición a la cual se añaden los procesos que están en espera del recurso. Sólo se permiten tres operaciones sobre un semáforo:

Inicializar

Espera (wait)

Señal (signal)

En algunos textos, se utilizan las notaciones P y V para las operaciones de espera y señal respectivamente, ya que ésta fue la notación empleada originalmente por Dijkstra (el creador de la solución) para referirse a las operaciones.

En JAVA usamos la clase ***Semaphore***, del paquete ***java.util.concurrent***. Los métodos serán:

* Lo primero es el método ***acquire()*** de la clase *Semaphore*. Este método bloquea el semáforo (*wait*).
* El método ***release()*** de la clase *Semaphore*, libera el semáforo para los demás procesos (*signal*).

**Ejemplo:**

Tenemos la clase hilo que contiene una sincronización con semáforos:

public class Hilo extends Thread {

private static Semaphore sem = new Semaphore(1, true);

@Override

public void run(){

int espera=0;

while(true){

try {

sem.acquire();

System.out.println("SemÃ¡foro adquirido por " + this.getName());

espera = (int) (Math.random() \* 300);

System.out.println(this.getName() + " esperando " + espera);

System.out.println("SemÃ¡foro liberado por " + this.getName());

//Thread.sleep(espera);

sem.release();

espera = (int) (Math.random() \* 300);

//Thread.sleep(espera);

} catch (InterruptedException ex) {

Logger.getLogger(Hilo.class.getName()).log(Level.SEVERE, null, ex);

}

}

}

}

El programa principal:

public class EjemploSemaforos {

/\*\*

\* @param args the command line arguments

\*/

public static void main(String[] args) {

Hilo h1= new Hilo();

Hilo h2=new Hilo();

h1.start();

h2.start();

}

}

En la declaración de semáforos podemos observar:

Semaphore sem = new Semaphore(1, true);

El número indica la cantidad de *cerrojos* que se pueden conceder, es decir, la cantidad de accesos a la sección crítica. Cuando un hilo realiza un *acquire* resta uno a la cantidad de accesos disponibles; si no quedan cerrojos disponibles el hilo se quedará bloqueado en el *acquire* hasta que otro hilo realice un *release* sobre el semáforo, liberando un acceso a la sección crítica.

El segundo argumento lógico es optativo, si el valor es *true* el semáforo intentará cederse lo más equitativamente posible a los hilos que lo soliciten.

## 2.7 Prioridades de los hilos.

Hasta ahora se ha supuesto que todos los *threads* se repartían el tiempo de la CPU por igual, pero esto no es generalmente así. Se repartirán la CPU por igual sólo si la prioridad de los distintos *thread* es la misma.

En todos los ejemplos que se han visto anteriormente los *threads* tenían la misma prioridad porque cuando se crea un *thread*, éste hereda la prioridad del thread que lo creó, sin embargo esta prioridad puede ser modificada en cualquier momento llamando al método *setPriority().* Para conocer la prioridad de un *thread* se utiliza el método *getPriority().*

**public final static int MAX\_PRIORITY = 10;**  prioridad máxima de un thread

**public final static int MIN\_PRIORITY = 1;**  prioridad mínima de un thread

**public final static int NORM\_PRIORITY = 5;** prioridad normal (por defecto) de un thread

**public final void setPriority(int newPriority);**

**public final int getPriority();**

El algoritmo de planificación (*scheduling*) es el siguiente: la CPU se asigna al *thread* en estado ejecutable de mayor prioridad y si hay varios, se reparten el tiempo mediante *round-robin*. Además este algoritmo es expulsivo, es decir, si se está ejecutando un *thread* y en ese momento pasa a estado ejecutable un *thread* de mayor prioridad, se le asigna directamente la CPU. Este algoritmo tiene un peligro: que un *thread* nunca se ejecute porque siempre existan *threads* de mayor prioridad. Java evita esto asignando la CPU también a *threads* de menor prioridad aunque cada mucho menos tiempo que a los *threads* de mayor prioridad. De todas formas, no debe confiarse la “no inanición” a la bondad del Java.

**Ejemplo:**

class ThreadConPrioridad extends Thread {

String nombre;

public ThreadConPrioridad(String n , int pri) {

super();

nombre=n;

setPriority(pri);

}

public void run() {

int i;

for(i=1;i<100;i++) {

System.out.print(nombre);

yield();

}

}

}

class Prioridad1 {

public static void main(String args[]) {

ThreadConPrioridad t1 = new ThreadConPrioridad("1",5);

ThreadConPrioridad t2 = new ThreadConPrioridad("2",5);

t1.start();

t2.start();

}

}

Salida por pantalla:

221211122122212121212121112221212121212121212121212121

212121212121212121212122121212121212121121212121221212

121212121212121212112121212121212121212121221212211212

121212121211212121212121212121221111

En este caso, los *threads* t1 y t2 tienen la misma prioridad (5), por lo que al llamar al método *yield()* se cede la ejecución al otro *thread*.

Si las prioridades hubieran sido otras. Por ejemplo:

ThreadConPrioridad t1 = new ThreadConPrioridad("1",5);

ThreadConPrioridad t2 = new ThreadConPrioridad("2",4);

La salida por pantalla habría sido diferente:

111111111111111111111111111111111111111111111111111111

111111111111111111121212121212121212111111111111111111

222222222222222222222222222222222222222222222222222222

222222222222222222222222222222222222

Como puede observarse, la mayoría del tiempo de la CPU es acaparada por el thread de mayor prioridad.

## 2.8 Compartición de información entre hilos.

Las tuberías se utilizan para intercambiar información entre hilos sin tener que recurrir a otros elementos como, por ejemplo, ficheros temporales, objetos creados y pasados como argumento en los constructores (secciones críticas) o variables estáticas.

Java proporciona las clases ***PipedReader*** y ***PipedWriter*** (y sus homólogos para bytes, ***PipedInpuntStream*** y ***PipedOutputStream***) para trabajar con tuberías a través de las cuales fluirán flujos de caracteres. La primera representa el extremo por el cual un hilo obtiene datos y la segunda el extremo por el cual un hilo envía datos.

La forma de crear una tubería es:

PipedWriter emisor = new PipedWriter();

PipedReader receptor = new PipedReader(emisor);

O

PipedReader receptor = new PipedReader();

PipedWriter emisor = new PipedWriter(receptor);

Vamos a realizar un ejemplo en el que un hilo genera texto para otro hilo y se lo envía mediante una tubería.

El programa principal definirá ambos extremos de una tubería que se pasarán a los hilos correspondientes; el código es el siguiente:

public static void main(String[] args) throws IOException {

PipedWriter emisor = new PipedWriter();

PipedReader receptor = new PipedReader(emisor);

Productor p = new Productor(emisor);

Consumidor c = new Consumidor(receptor);

p.start();

c.start();

}

En el hilo productor necesitamos dos atributos: la tubería y un atributo de tipo *PrintWriter* que nos proporcionará el método *println*, que *PipedWriter* no tiene, para enviar mensajes por la tubería. Tendrá el siguiente código:

public class Productor extends Thread {

private PipedWriter emisor;

private PrintWriter flujoS;

public Productor(PipedWriter em){

emisor = em;

flujoS = new PrintWriter(emisor);

}

@Override

public void run(){

int espera;

while(true){

almacenarMensaje();

try{

espera = (int)(Math.random()\*10);

sleep(espera);

}

catch(Exception e){

}

}

}

private synchronized void almacenarMensaje() {

int num\_MS;

String Texto\_MS;

num\_MS = (int)(Math.random()\*100);

Texto\_MS = "mensaje #" + num\_MS;

flujoS.println(Texto\_MS); //enviamos el mensaje por la tubería.

System.out.println("Productor " + getName() + " almacena: " + Texto\_MS);

}

protected void finalize() throws IOException{

if (flujoS!=null) {

flujoS.close();

flujoS=null;

}

if (emisor!=null){

emisor.close();

emisor=null;

}

}

}

En el hilo consumidor necesitamos, asimismo, otros dos atributos: la tubería y un atributo de tipo *BufferedReader* que nos proporcionará el método *readLine*, que *PipedReader* no tiene, para obtener los mensajes de la tubería. Tendrá el siguiente código:

public class Consumidor extends Thread {

private PipedReader receptor;

private BufferedReader flujoE;

public Consumidor(PipedReader re){

receptor = re;

flujoE = new BufferedReader(receptor);

}

@Override

public void run(){

while(true){

obtenerMensaje();

}

}

private void obtenerMensaje() {

String MS = null;

try{

MS = flujoE.readLine(); //Obtenemos mensaje de la tubería

System.out.println("Consumidor: " + getName() + " obtuvo: " + MS);

}

catch(Exception e){

}

}

protected void finalize() throws IOException{

if (flujoE!=null){

flujoE.close();

flujoE = null;

}

if (receptor != null){

receptor.close();

receptor=null;

}

}

}

**Ejercicio propuesto:** Reescribe el código anterior de forma que, mediante semáforos, se establezca una sincronización de forma que el emisor escribe en la tubería y el consumidor obtiene el texto de forma alternativa.

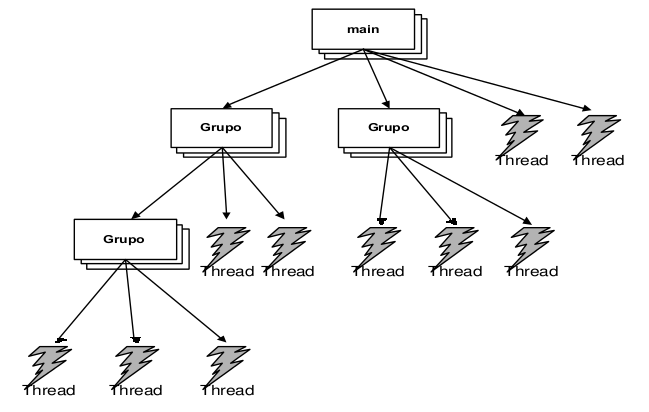
**Ejercicio propuesto 2:** Un hilo crea un listado de números y el receptor los obtiene y los ordena, después se muestra el listado ordenado.

## 2.9 Agrupación de hilos.

Todo *thread* creado en Java pertenece a un **Grupo de *threads***. Los grupos de *threads* sirven para manejar como una unidad a un conjunto de *threads*; por ejemplo: iniciar o detener varios *threads* mediante una sola operación. Hay que señalar que los grupos de *threads* no añaden ningún tipo de funcionalidad, únicamente sirven para que sea más cómodo el manejo muchos *threads* en un programa. Para ello se utiliza la clase *ThreadGroup* que pertenece al *paquete java.lang.*

Cuando se inicia la ejecución de un programa, Java crea un grupo de *threads* llamado *main*. A no ser que se especifique lo contrario, cuando se crea un *thread*, se añade al grupo al que pertenece el *thread* que lo creó. Esa es la razón por la cual todos los *threads* creados hasta el momento pertenecen al grupo *main*.

Un grupo de *threads* puede contener *threads*, pero también grupos, y todos ellos pertenecen a un grupo raíz: *main*.



**Creación de un grupo de Threads** mediante llamadas al conductor:

public ThreadGroup(String nombre);

Ejemplo: ThreadGroup tg = new ThreadGroup(“MiGrupo”);

public ThreadGroup(ThreadGroup padre, String nombre);

Ejemplo: ThreadGroup tg2 = new ThreadGroup( tg , “MiGrupo2”);

Es posible **obtener el grupo al que pertenece un thread** mediante la llamada al método getThreadGroup() de la clase Thread:

class Grupo1 {

public static void main(String args[]) {

ThreadGroup tg =Thread.currentThread().getThreadGroup();

System.out.println(tg.getName());

}

}

El método getName() devuelve un String que contiene el nombre del grupo.

La salida por pantalla, evidentemente es:

main

**Asignación de un Thread a un grupo.**

Como se ha visto, los *threads* se insertan por defecto en el mismo grupo que el *thread* actual (el *thread* que lo crea) a no ser que se indique explícitamente a qué grupo debe pertenecer. Esto se realiza en la llamada al constructor del *thread*:

**public Thread(ThreadGroup grupo, String nombreDelThread);**

**public Thread(ThreadGroup grupo, Runnable destino, String nombreDelThread);**

**Operaciones sobre un conjunto de redes.**

**public final void suspend();** Pasan al estado de “suspendido” todos los threads descendientes de este grupo (todos los threads pertenecientes a este grupo) y se llama al método suspend() de todos los grupos pertenecientes a este grupo (valga la redundancia).

**public final void resume();** Pasan al estado de “ejecutable” todos los threads descendientes de este grupo que estaban en estado “suspendido”.

**public final void stop();** Pasan al estado “muerto” todos los threads descendientes de este grupo.

**Operaciones sobre grupos.**

**public final void setDaemon(boolean daemon);** Convierte al grupo en daemon si el parámetro es true43 . Un grupo daemon es destruido automáticamente cuando no contiene ningún thread o grupo.

**public final boolean isDaemon();** Devuelve true si el grupo es un daemon y false en caso contrario.

**public final void setMaxPriority(int pri);** Establece la máxima prioridad que puede tener un thread descendiente de este grupo, aunque los threads descendientes que ya pertenecían al grupo antes de llamar al método setMaxPriority() pueden conservar su prioridad mayor que el parámetro (pri).

**public final void destroy() throws IllegalThreadStateException;** Destruye el grupo (pero no los threads descendientes). Este grupo debe estar vacío: no debe contener threads activos, de lo contrario se genera una excepción de la clase IllegalThreadStateException. Si el grupo no está vacío, hay que llamar antes al método stop() del grupo para que todos los threads descendientes pasen al estado “muerto”.

**public final boolean parentOf(ThreadGroup g);** Devuelve true si este grupo es padre (o es el mismo) que el grupo que se pasa como parámetro (g) y false en caso contrario.

**Obtención del contenido de un grupo.**

**public int enumerate(Thread lista[]);**

**public int enumerate(Thread lista[], boolean recurr);**

**public int enumerate(ThreadGroup lista[]);**

**public int enumerate(ThreadGroup lista[], boolean recurr);**

El método *enumerate* puede obtener un vector con los *threads* pertenecientes al grupo (los casos 1 y 2) o un vector con los grupos pertenecientes a un grupo (casos 3 y 4) se almacena en el parámetro lista[].

Por defecto, se obtienen listas de *threads* (y grupos) recursivas, añadiendo los threads o grupos del grupo sobre el que se realiza la llamada y de los grupos descendientes del mismo. Para obtener únicamente los *threads* pertenecientes al grupo en cuestión es necesario pasar un nuevo parámetro de tipo *boolean* (*recurr*) con el valor false.

**Ejemplo:**

class MiThread extends Thread {

public MiThread(ThreadGroup tg, String nom) {

super(tg,nom);

}

public void run() {

System.out.println("Ejecutado "+getName());

}

}

class Grupo2 {

public static void main(String args[]) throws InterruptedException {

ThreadGroup tg = new ThreadGroup("grupo uno");

ThreadGroup tg2 = new ThreadGroup(tg,"grupo dos");

MiThread t1 = new MiThread(tg,"Thread 1.1");

MiThread t12 = new MiThread(tg,"Thread 1.2");

MiThread t21 = new MiThread(tg2,"Thread 2.1");

MiThread t22 = new MiThread(tg2,"Thread 2.2");

Thread t[] = new Thread[4];

int i;

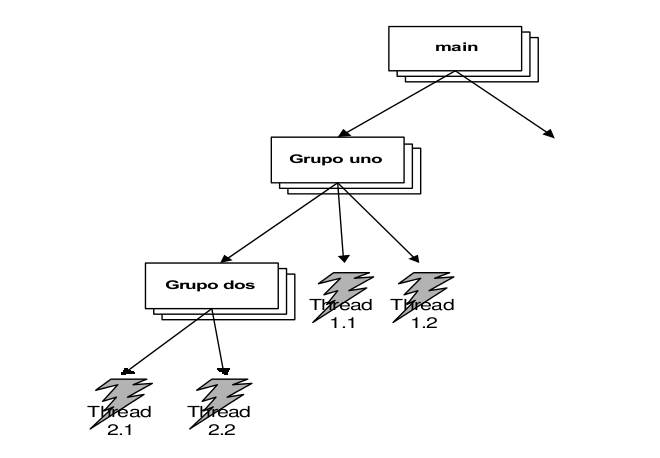
tg.enumerate(t);

for (i=0;i<t.length;i++)

System.out.println(t[i]);

}

}



En este caso, la salida por pantalla es la siguiente:

Thread[Thread 1.1,5,grupo uno]

Thread[Thread 1.2,5,grupo uno]

Thread[Thread 2.1,5,grupo dos]

Thread[Thread 2.2,5,grupo dos]

Si en lugar de *enumerate(t)* se hubiera utilizado la fórmula *enumerate(t, false),* la salida por pantalla habría sido:

Thread[Thread 1.1,5,grupo uno]

Thread[Thread 1.2,5,grupo uno]

null

null

## 2.10 Programación de aplicaciones multihilo.

Programar aplicaciones en multihilo tiene algunas ventajas y desventajas con respecto a aplicaciones que utilizan un solo hilo:

- Pueden utilizar la capacidad completa de procesadores modernos (con múltiples *kernels*)

- Permiten crear aplicaciones que funcionen más rápido, pues la funcionalidad requerida puede ser ejecutada en paralelo

- Permite crear aplicaciones en donde el orden de ejecución es controlado estrictamente

- Como mayor desventaja se puede decir que programación multihilo es más complicada y requiere un diseño cuidadoso de la aplicación y programadores experimentados

La programación con hilos, también conocida como multiproceso, tiene la ventaja de poder trabajar de manera asíncrona. Esto permite que aquellos procesos que pueden requerir un tiempo más o menos largo en llevarse a cabo se pongan a trabajar 'paralelamente' al proceso principal, de manera que la aplicación pueda retomar el control y así el usuario seguir trabajando con ella.

La programación multihilo no es siempre la solución correcta para todas las aplicaciones e incluso en algunos casos puede relentizar la aplicación aunque parezca que no es así o cosas aún peores como pérdida de datos, etc.

**Conclusiones**

• La programación de hilos requiere un cambio en la forma de pensar al programar, ya que la ejecución del código ya no se realiza de forma secuencial sino paralelamente.

• Tiene suma importancia saber cuándo hay que hacer uso de threads y cuándo evitarlos. Las principales razones de usarlos son al gestionar varias tareas que al entremezclarse hagan un uso más eficiente del ordenador, o porque al usuario le interese por una determinada circunstancia.

• Un único hilo es similar a un programa secuencial; es decir, tiene un comienzo, una secuencia y un final, además en cualquier momento durante la ejecución existe un sólo punto de ejecución. Sin embargo, un hilo no es un programa; no puede correr por sí mismo, corre dentro de un programa. Un hilo por si mismo no nos ofrece nada nuevo, es la habilidad de ejecutar varios hilos dentro de un programa lo que ofrece algo nuevo y útil, ya que cada uno de estos hilos puede ejecutar tareas distintas.

• Una ventaja de usar hilos es que las conmutaciones de contexto de procesos ligeras, sustituyen a las conmutaciones de contexto de procesos pesadas. Debido a que todos los hilos de un determinado proceso comparten el mismo espacio de memoria, una conmutación del proceso ligera sólo cambia la ejecución del programa y las variables locales, mientras que una conmutación de contexto pesada debe intercambiar todo el espacio en memoria.

• La mayor dificultad de trabajar con hilos es que dado un recurso, este podría estar siendo compartido por más de un hilo. Por ello hay que asegurarse de que varios hilos no intenten leer y cambiar un recurso simultáneamente. Esto requiere de un uso cuidadoso de *synchronized*, ya que aunque es una herramienta muy útil, puede llevar a situaciones de interbloqueo sin darnos cuenta.

• Los programas paralelos son usados en ambientes críticos y muy sensibles tales como operaciones y manipulación en bases de datos, recolección de información, etc. Con un creciente énfasis en el Web y herramientas independientes de la plataforma, es por esto que Java ha tenido un gran impacto en la creación de nuevos sistemas paralelo. El problema es que Java soporta hilos, pero no ofrece una protección adecuada para los recursos compartidos por los hilos y no tiene un mecanismo de detección de interbloqueos.

• En Java parece que existe un límite en la cantidad de hilos a crear, ya que en algunas ocasiones un número de hilos muy elevado da muestras de colapso, este punto crítico puede que se alcance con unos pocos cientos. Normalmente sólo se crean unos pocos hilos para solucionar un problema y por tanto este límite no se suele alcanzar, aunque puede parecer una limitación en grandes diseños.