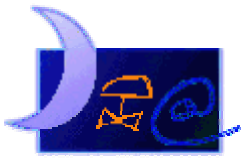
**Programación Orientada a Objetos**
**Mayo, 2002**

# Java Threads
# (Hilos en Java)

**Inmaculada González Pérez**

**Antonio Juan Sánchez Martín**

**David Vicente Hernández**



Departamento de Informática y Automática
Universidad de Salamanca

CRÉDITOS:

Revisado por:

Dr. Francisco José García Peñalvo
Área de Ciencia de la Computación e Inteligencia Artificial
Departamento de Informática y Automática
Facultad de Ciencias – Universidad de Salamanca

Información de los autores:

Inmaculada González Pérez
inmab612@hotmail.com

Antonio Juan Sánchez Martín
ant_solo@hotmail.com

David Vicente Hernández
dvdvicente@hotmail.com

# Resumen

Este documento recoge una descripción del concepto y uso de los hilos o *threads*, como flujo de control en un programa, en el lenguaje de programación orientado a objetos que es Java.

Se tratan los aspectos más relevantes desde una perspectiva teórica, incluyendo, a modo de aclaración, breves ejemplificaciones de uso

# Tabla de contenidos

# 1. Conceptos Básicos sobre Hilos

El multihilo soportado en Java gira alrededor del concepto de hilo. La cuestión es, ¿qué es un hilo? De forma sencilla, un hilo es un único flujo de ejecución dentro de un proceso. Pero será mejor comenzar desde el principio y explicar qué es un proceso.

Un proceso es un programa ejecutándose dentro de su propio espacio de direcciones. Java es un sistema multiproceso, esto significa que soporta varios procesos corriendo a la vez dentro de sus propios espacios de direcciones. Estamos más familiarizados con el término multitarea, el cual describe un escenario muy similar al multiproceso. Por ejemplo, consideremos la cantidad de aplicaciones que corren a la vez dentro de un mismo entorno gráfico. Mientras escribo esto, está corriendo Microsoft Word además de Internet Explorer, Windows Explorer, CD Player y el Volumen Control. Estas aplicaciones son todas procesos ejecutados dentro de Windows 95. De esta forma, se puede pensar que los procesos son análogos a las aplicaciones o a programas aislados, pero cada proceso tiene asignado espacio propio de ejecución dentro del sistema.

Un hilo es una secuencia de código en ejecución dentro del contexto de un proceso. Los hilos no pueden ejecutarse ellos solos; requieren la supervisión de un proceso padre para correr. Dentro de cada proceso hay varios hilos ejecutándose. Por ejemplo, Word puede tener un hilo en *background* chequeando automáticamente la gramática de lo que estoy escribiendo, mientras otro hilo puede estar salvando automáticamente los cambios del documento en el que estoy trabajando. Como Word, cada aplicación (proceso) puede correr varios hilos los cuales están realizando diferentes tareas. Esto significa que los hilos están siempre asociados con un proceso en particular.

Los hilos a menudo son conocidos o llamados procesos ligeros. Un hilo, en efecto, es muy similar a un proceso pero con la diferencia de que un hilo siempre corre dentro del contexto de otro programa. Por el contrario, los procesos mantienen su propio espacio de direcciones y entorno de operaciones. Los hilos dependen de un programa padre en lo que se refiere a recursos de ejecución. La siguiente figura muestra le relación entre hilos y procesos.

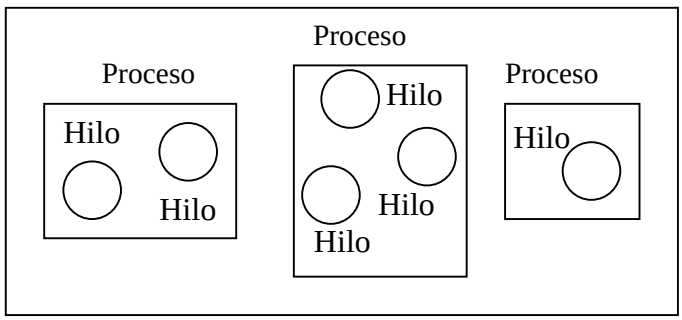



Figura 1.1 Relación entre hilos y procesos

Java es un lenguaje de programación que incorpora hilos en el corazón del mismo lenguaje. Comúnmente, los hilos son implementados a nivel de sistema, requiriendo una interfaz de programación específica separada del núcleo del lenguaje de programación. Esto es lo que ocurre con C/C++ programando en Windows, porque se necesita usar la interfaz de programación Win32 para desarrollar aplicaciones Windows multihilo.

Java se presenta como ambos, como lenguaje y como sistema de tiempo de ejecución (*runtime*), siendo posible integrar hilos dentro de ambos. El resultado final es que se pueden usar hilos Java como standard, en cualquier plataforma.

# 2. Clases Relacionadas con los Hilos

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`.

## `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. Para una documentación más extensa de todas las funciones miembro, se sugiere consultar http://java.sun.com/products/jdk/1.2/docs/api/java/lang/Thread.html

## `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*.

Aunque es un único método el que se define en esta interfaz, en http://java.sun.com/products/jdk/1.2/docs/api/java/lang/Runnable.html se puede encontrar amplia documentación sobre la misma.

## `ThreadDeath`

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.

En http://java.sun.com/products/jdk/1.2/docs/api/java/lang/ThreadDeath.html se puede encontrar documentación completa sobre las funciones miembro de esta clase.

## `ThreadGroup`

La clase `ThreadGroup` se utiliza para manejar un grupo de hilos de modo conjunto. Esto nos proporciona un medio para controlar de modo eficiente la ejecución de una serie de hilos. Por ejemplo la clase `ThreadGroup` nos proporciona métodos `stop`, `suspend` y `resume` para controlar la ejecución de todos los hilos pertenecientes al grupo. Los grupos de hilos también pueden contener otros grupos de hilos permitiendo una jerarquía anidada de hilos. Los hilos individuales tienen acceso al grupo pero no al padre del grupo.

La relación de métodos de esta clase, con amplia documentación sobre ello, se puede encontrar en http://java.sun.com/products/jdk/1.2/docs/api/java/lang/ThreadGroup.html

## `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).

Normalmente estos métodos se utilizan cuando hay ejecución multihilo, es decir, cuando un método espera a que otro método termine de hacer algo antes de poder continuar. El primer hilo espera hasta que otro hilo le notifique que puede continuar. La clase objeto está en la parte superior de la jerarquía de Java, lo cual significa que es el padre de todas las clases. En otras palabras, cada clase Java hereda la funcionalidad proporcionada por la clase objeto, incluyendo los métodos `wait`, `notify` y `notifyAll`.

La lista de funciones miembro de esta clase y documentación sobre las mismas puede encontrarse en http://java.sun.com/products/jdk/1.2/docs/api/java/lang/Object.html

# 3. Creación de Hilos

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: implementando la interfaz `Runnable`, o extendiendo la clase `Thread`, esto es, creando una subclase de esta clase.

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 un 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() ;
}
```

Comentados los aspectos más importantes de la interfaz Runnable, veamos ahora la definición de la clase Thread, de la cual podemos deducir lo que realmente se está haciendo:

```
public class Thread implements Runnable {
    ...
    public void run() {
```

```
            if( tarea != null )
                 tarea.run() ;

                }
        ...}
```

Se deduce, por tanto, que la propia clase `Thread` de Java también implementa la interfaz `Runnable`. Observamos que en el método `run` de `Thread` se comprueba si la clase con que se está trabajando en ese momento (tarea), que es la clase que se pretende ejecutar como hilo, es o no nula. En caso de no ser nula, se invoca al método `run` propio de dicha clase.

# 4. Estado y Control de Hilos

## 4.1. Estados de un Hilo

El comportamiento de un hilo depende del estado en que se encuentre, este estado define su modo de operación actual, por ejemplo, si esta 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 esta 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 esta 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 esta 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.

# 4.2. Control de un hilo

## *Arranque de un hilo*

En el contexto de las aplicaciones, sabemos que es `main` la primera función que se invoca tras arrancar, y por tanto, lógicamente, es el lugar más apropiado para crear y arrancar otros hilos.

La línea de código:

```
t1 = new TestTh( "Thread 1",(int)(Math.random()*2000) );
```

siendo `TestTh` una subclase de la clase `Thread` (o una clase que implemente la interfaz `Runnable`) crea un nuevo hilo. Los dos argumentos pasados, sin mayor relevancia, satisfarán el prototipo del constructor de la clase y se utilizarán para la inicialización del objeto.

Al tener control directo sobre los hilos, tenemos que arrancarlos explícitamente. Como ya se comentó anteriormente, es la función miembro `start` la que nos permite hacerlo. En nuestro ejemplo sería:

```
t1.start();
```

`start`, en realidad es un método oculto en el hilo que llama al método `run`.

## *Manipulación de un hilo*

Si todo fue bien en la creación del objeto `TestTh` (t1), éste debería contener un hilo, una traza de ejecución válida, que controlaremos en el método `run` del objeto.

El cuerpo de esta función miembro viene a ser el cuerpo de un programa como ya los conocemos. Digamos que es la rutina `main` a nivel de hilo. Todo lo que queremos que haga el hilo debe estar dentro del método `run`. Cuando finalice `run`, finalizará también el hilo que lo ejecutaba.

## *Suspensión de un Hilo*

La función miembro `suspend` de la clase `Thread` permite tener un control sobre el hilo de modo que podamos desactivarlo, detener su actividad durante un intervalo de tiempo indeterminado, a diferencia del uso de la llamada al sistema `sleep`, que simplemente lleva al hilo a un estado de “dormido”, y siempre durante un número de milisegundos concreto.

Este método puede resultar útil si, construyendo un *applet* con un hilo de animación, queremos permitir al usuario detener (que no finalizar) la animación, hasta que éste decida reanudarla.

Este método no detiene la ejecución permanentemente. El hilo es suspendido indefinidamente y para volver a activarlo de nuevo necesitamos realizar una invocación a la función miembro `resume`.

### *Parada de un Hilo*

Ya conocemos los métodos de control de hilos que nos permiten arrancarlos, suspenderlos y reanudarlos. El último elemento de control que se necesita sobre hilos es el método `stop`, utilizado para terminar la ejecución de un hilo de forma permanente:

```
t1.stop();
```

Señalar que esta llamada no destruye el hilo, sino que detiene su ejecución, y ésta no puede reanudarse con el método `start`. Cuando se desasignen las variables que se usan en el hilo, el objeto hilo (creado con `new`) quedará marcado para eliminarlo y el `garbage collector` (recolector de basura de Java) se encargará de liberar la memoria que utilizaba.

Tiene sentido su utilidad, por ejemplo, en aplicaciones complejas que necesiten un control sobre cada uno de los hilos que se lancen.

Por último, un método de control de hilos que nos permite comprobar si una instancia está viva (el hilo se ha arrancado y aún no se ha detenido) o no (bien no se arrancó; bien ya finalizó). Estamos hablando de la función miembro `isAlive`.

```
t1.isAlive();
```

Devolverá `true` en caso de que el hilo `t1` esté vivo, es decir, ya se haya llamado a su método `run` y no haya sido parado con un `stop` ni haya terminado el método `run` en su ejecución. En otro caso, lógicamente, devolverá `false`.

# 5. Agrupamiento de Hilos

Todo hilo de Java es un miembro de un *grupo de hilos*. Los grupos de hilos proporcionan un mecanismo de reunión de múltiples hilos dentro de un único objeto y de manipulación de dichos hilos en conjunto, en lugar de una forma individual. Por ejemplo, se pueden arrancar o suspender todos los hilos que están dentro de un grupo con una única llamada al método. Los grupos de hilos de Java están implementados por la clase `ThreadGroup` en el paquete `Java.lang`.

El *runtime system* (sistema de tiempo de ejecución) de Java pone un hilo dentro de un grupo de hilos en el momento de la construcción del mismo. Cuando creas un hilo, se puede dejar que el sistema de tiempo de ejecución ponga el nuevo hilo en algún grupo razonable por defecto, o se puede establecer explícitamente el grupo del nuevo hilo. El hilo es un miembro permanente de aquel que sea el grupo de hilos al cual se unió en el momento de su creación. No puede moverse un hilo a un nuevo grupo una vez que ha sido creado.

## 5.1. El grupo de hilos por defecto

Si se crea un nuevo hilo sin especificar su grupo en el constructor, el sistema de tiempo de ejecución colocará el nuevo hilo automáticamente en el mismo grupo que el hilo que lo ha creado (conocido como grupo de *hilos actual* e *hilo actual*, respectivamente). Así que, ¿cuál es el grupo de hilos del hilo principal de una aplicación?

Cuando se arranca una aplicación Java, el sistema de tiempo de ejecución de Java crea una instancia de la clase `ThreadGroup` llamada `main`. A menos que especifiques lo contrario, todos los nuevos hilos que crees se convertirán en miembros del grupo de hilos `main`.

## 5.2. Creación de un hilo en un grupo de forma explícita

Como hemos mencionado anteriormente, un hilo es un miembro permanente de aquel grupo de hilos al cual se unió en el momento de su creación (no tenemos la posibilidad de cambiarlo posteriormente). De este modo, si quieres poner tu nuevo hilo en un grupo de hilos distinto del grupo por defecto, debes especificarlo explícitamente cuando lo creas. La clase `Thread` tiene tres constructores que te permiten establecer un nuevo grupo de hilos.

- `public Thread( ThreadGroup` *`group`*`, Runnable` *`runnable`* `)`
- `public Thread( ThreadGroup` *`group`*`, String` *`name`* `)`
- `public Thread( ThreadGroup group, Runnable` *`runnable`*`, String` *`name`* `)`

Cada uno de estos constructores crea un nuevo hilo, lo inicializa en base a los parámetros `Runnable` y `String`, y hace al nuevo hilo miembro del grupo especificado. Por ejemplo, la siguiente muestra de código crea un grupo de hilos (`myThreadGroup`) y entonces crea un hilo (`myThread`) en dicho grupo

```
ThreadGroup myThreadGroup = new
    ThreadGroup( “My Group of Threads” );
Thread myThread = new
    Thread( myThreadGroup, “a thread for my group” );
```

El `ThreadGroup` pasado al constructor `Thread` no tiene que ser necesariamente un grupo que hayas creado tú, puede tratarse de un grupo creado por el sistema de ejecución de Java, o un grupo creado por la aplicación en la cual se está ejecutando el *applet*.

## 5.3. 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.

El navegador *HotJava* es un ejemplo de aplicación que implementa su propio administrador de seguridad. *HotJava* necesita asegurar que los *applet*s se comportan de forma correcta y que no realizan ninguna acción "sucia" respecto a los demás *applet*s que se están ejecutando al mismo tiempo (como por ejemplo rebajar la prioridad de los hilos de otro *applet*). El administrador de seguridad de *HotJava* no permite a los hilos pertenecientes a grupos diferentes modificarse entre sí. Obsérvese que las restricciones de acceso basadas en los grupos de hilos pueden variar de un navegador a otros, de modo que los *applet*s tal vez se comporten de distinta manera en distintos navegadores. A continuación, una breve explicación sobre la seguridad en Java, concretamente en relación a los *applet*s.

# 6. Planificación y Prioridad de Hilos

## 6.1. Planificación (*Scheduling*)

Java tiene un Planificador (*Scheduler)*, una lista de procesos, que muestra por pantalla todos los hilos que se están ejecutando en todos los programas y decide cuáles deben ejecutarse y cuáles deben encontrarse preparados para su ejecución. Hay dos características de los hilos que el planificador tiene en cuenta en este proceso de decisión.

- La prioridad del hilo (la más importante).
- El indicador de demonio (que pasaremos a explicar en los siguientes apartados).

La regla básica del planificador es que si solamente hay hilos demonio ejecutándose, la *Máquina Virtual Java* (JVM) concluirá. Los nuevos hilos heredan la prioridad y el indicador de demonio de los hilos que los han creado. El planificador determina qué hilos deberán ejecutarse comprobando la prioridad de todos los hilos. Aquellos con prioridad más alta dispondrán del procesador antes de los que tienen prioridad más baja.

El planificador puede seguir dos patrones, ***preventivo*** y ***no preventivo***. Los planificadores preventivos proporcionan un segmento de tiempo a todos los hilos que están corriendo en el sistema. El planificador decide cuál será el siguiente hilo a ejecutarse y llama a `resume` para darle vida durante un período fijo de tiempo. Cuando finaliza ese período de tiempo, se llama a su método `suspend` y el siguiente hilo en la lista de procesos será relanzado mediante su método `resume`. Los planificadores no preventivos, en cambio, deciden qué hilo debe correr y lo ejecutan hasta que concluye. El hilo tiene control total sobre el sistema mientras esté en ejecución. El método `yield` es un mecanismo que permite a un hilo forzar al planificador para que comience la ejecución de otro hilo que esté esperando. Dependiendo del sistema en que esté corriendo Java, el planificador será preventivo o no preventivo.

El planificador de hilos no está especificado tan rigurosamente como el resto de clases en Java, y en este asunto hay, sin duda, diferencias de un sistema a otro. Por ejemplo, la versión del JDK 1.0.2 de Solaris es una versión no preventiva, a diferencia del planificador de Win32, que sí lo es. [Esto es exactamente lo opuesto a lo que nos esperábamos.] Un planificador no preventivo no interrumpirá un hilo en ejecución, de forma muy parecida al comportamiento de Windows 3.1. El planificador de hilos Java de Win32 sí interrumpirá los hilos en ejecución, dando lugar a una planificación más fiable. Por ejemplo, si se arrancan dos hilos con grandes bucles de ejecución bajo un sistema Solaris, el hilo que arrancó antes completará su tarea antes de que el otro consiga arrancar. Pero en Windows95 o NT, el segundo hilo sí consigue turno de ejecución.

## 6.2. Prioridad

Cada hilo tiene una prioridad, que no es más que un valor entero entre 1 y 10, de modo que cuanto mayor el valor, mayor es la prioridad.

El planificador determina el hilo que debe ejecutarse en función de la prioridad asignada a cada uno de ellos. Cuando se crea un hilo en Java, éste hereda la prioridad de su padre, el hilo que lo ha creado. A partir de aquí se le puede modificar su prioridad en cualquier momento utilizando el método `setPriority.` Las prioridades de un hilo varían en un rango de enteros comprendido entre `MIN_PRIORITY` y `MAX_PRIORITY` (anbas definidas en la clase `Thread`). El entero más alto designará la prioridad más alta y el más bajo, como es de esperar, la menor. Se ejecutará primero el hilo de prioridad superior, el llamado "Ejecutables", y sólo cuando éste para, abandona o se convierte en "No Ejecutable", comienza la ejecución de en hilo de prioridad inferior. Si dos hilos tienen la misma prioridad, el programador elige uno de ellos en alguna forma de competición. El hilo seleccionado se ejecutará hasta que:

- Un hilo comprioridad mayor pase a ser "Ejecutable".
- En sistemas que soportan tiempo-compartido, termina su tiempo.
- Abandone, o termine su método run.

Luego, un segundo hilo puede ejecutarse, y así continuamente hasta que el intérprete abandone.

El algoritmo del sistema de ejecución de hilos que sigue Java es de tipo preventivo. Si en un momento dado un hilo que tiene una prioridad mayor a cualquier otro hilo que se está ejecutando pasa a ser "Ejecutable", entonces el sistema elige a este nuevo hilo.

### *Ejemplo:* ***carrera de hilos***

En el siguiente ejemplo, mostramos la ejecución de dos hilos con diferentes prioridades. Un hilo se ejecuta a prioridad más baja que el otro. Los hilos incrementarán sus contadores hasta que el hilo que tiene prioridad más alta alcance al contador que corresponde a la tarea con ejecución más lenta.

```
import Java.awt.*;
import Java.applet.Applet;

// En este applet se crean dos hilos que incrementan un
contador, se
// proporcionan distintas prioridades a cada uno y se para
cuando los
// dos coinciden

public class SchThread extends Applet {
    Contar alto,bajo;

    public void init() {
        // Creamos un thread en 200, ya adelantado
```

```
        bajo = new Contar( 200 );
        // El otro comienza desde cero
        alto = new Contar( 0 );
        // Al que comienza en 200 le asignamos prioridad mínima
        bajo.setPriority( Thread.MIN_PRIORITY );
        // Y al otro máxima
        alto.setPriority( Thread.MAX_PRIORITY );
        System.out.println( "Prioridad alta es
"+alto.getPriority() );
        System.out.println( "Prioridad baja es
"+bajo.getPriority() );
        }


    // Arrancamos los dos threads, y vamos repintando hasta que
    //el thread que tiene prioridad más alta alcanza o supera al
    //que tiene prioridad más baja, pero empezó a contar más
//alto

    public void start() {
        bajo.start();
        alto.start();
        while( alto.getContar() < bajo.getContar() )
            repaint();
        repaint();
        bajo.stop();
        alto.stop();
        }


    // Vamos pintando los incrementos que realizan ambos threads
    public void paint( Graphics g ) {
        g.drawString( "bajo = "+bajo.getContar()+
            " alto = "+alto.getContar(),10,10 );
        System.out.println( "bajo = "+bajo.getContar()+
            " alto = "+alto.getContar() );
        }


    // Para parar la ejecución de los threads
    public void stop() {
        bajo.stop();
        alto.stop();
        }
    }
```

## 6.3. Tiempo compartido

La estrategia del tiempo compartido determina cuál de una serie de hilos con igual prioridad se pasa a ejecutar. Por ejemplo, supongamos un programa Java que crea dos hilos “egoístas” con la misma prioridad, que tienen el siguiente método `run`:

```
public void run() {
    while (tick < 400000) {
        tick++;
        if ((tick % 50000) == 0) {
            System.out.println("Thread #" + num + ", tick
= " + tick);
        }
    }
}
```

Este método sólo cuenta desde 1 hasta 400.000. La variable `tick` es pública y se utiliza para determinar cuánto se ha progresado. Cada 50.00 ticks imprime el identificador del hilo y su contador `tick`. Cuando este programa se ejecuta en un sistema que utiliza el tiempo compartido, la salida será del tipo :

Thread #1, tick = 50000
Thread #0, tick = 50000
Thread #0, tick = 100000
Thread #1, tick = 100000
Thread #1, tick = 150000
Thread #1, tick = 200000
Thread #0, tick = 150000
Thread #0, tick = 200000
Thread #1, tick = 250000
Thread #0, tick = 250000
Thread #0, tick = 300000
Thread #1, tick = 300000
Thread #1, tick = 350000
Thread #0, tick = 350000
Thread #0, tick = 400000
Thread #1, tick = 400000

El sistema de tiempo compartido divide el tiempo de proceso de la CPU en espacios de tiempo y le asigna tiempo de proceso a un hilo dependiendo de su prioridad. En este caso, como las prioridades del los hilos eran iguales, los tiempos de proceso también lo son. Observa que el tiempo compartido nos ofrece garantías sobre la frecuencia y el orden en el que se van a ejecutar los hilos.

Para hacer una comparativa, veamos cómo sería la salida en una máquina sin tiempo compartido, donde se ejecuta un único hilo continuadamente hasta que aparece otro con prioridad superior.

Thread #0, tick = 50000
Thread #0, tick = 100000
Thread #0, tick = 150000

Thread #0, tick = 200000
Thread #0, tick = 250000
Thread #0, tick = 300000
Thread #0, tick = 350000
Thread #0, tick = 400000
Thread #1, tick = 50000
Thread #1, tick = 100000
Thread #1, tick = 150000
Thread #1, tick = 200000
Thread #1, tick = 250000
Thread #1, tick = 300000
Thread #1, tick = 350000
Thread #1, tick = 400000

**Nota :** No todos los sistemas aceptan el tiempo compartido, por lo tanto los programas en Java no deberían ser relativos al tiempo compartido ya que podrían producirse resultados distintos para sistemas diferentes.

# 7. Sincronización

El problema de la sincronización de hilos tiene lugar cuando varios hilos intentan acceder al mismo recurso o dato. A la hora de acceder a datos comunes, los hilos necesitan establecer cierto orden, por ejemplo en el caso del productor consumidor. Para asegurarse de que hilos concurrentes no se estorban y operan correctamente con datos (o recursos) compartidos, un sistema estable previene la inanición y el punto muerto o interbloqueo. La inanición tiene lugar cuando uno o más hilos están bloqueados al intentar conseguir acceso a un recurso compartido de ocurrencias limitadas. El interbloqueo es la última fase de la inanición; ocurre cuando uno o más hilos están esperando una condición que no puede ser satisfecha. Esto ocurre muy frecuentemente cuando dos o más hilos están esperando a que el otro u otros se desbloquee, respectivamente.

A continuación se presenta un ejemplo, el problema del Productor/Consumidor, con la intención de explicar de una forma más práctica el concepto y las situciones de sincronización de hilos.

## 7.1. Ejemplo: problema del productor-consumidor

El productor genera un entero entre 0 y 9 (inclusive), lo almacena en un objeto "*CubbyHole*", e imprime el número generado. Para hacer más interesante el problema de la sincronización, el productor duerme durante un tiempo aleatorio entre 0 y 100 milisegundos antes de repetir el ciclo de generación de números.

```
class Productor extends Thread {
private CubbyHole cubbyhole;
private int numero;
     public Productor(CubbyHole c, int numero) {
       cubbyhole = c;
       this.numero = numero;
```

```
      }
    public void run() {
      for (int i = 0; i < 10; i++) {
       cubbyhole.put(i);
       System.out.println("Productor  #"+this.numero+"  pone:
" +i);
              try {
                  sleep((int)(Math.random() * 100));
              } catch (InterruptedException e) {
                }
      }
    }
  }
```

El consumidor, por su parte, está “hambriento”, consume todos los enteros de `CubbyHole` (exactamente el mismo objeto en que el productor puso los enteros en primer lugar) tan pronto como estén disponibles.

```
class Consumidor extends Thread {
 private CubbyHole cubbyhole;
 private int numero;

 public Consumidor(CubbyHole c, int numero) {

  cubbyhole = c;
  this.numero = numero;
 }
 public void run() {
  int value = 0;
  for (int i = 0; i < 10; i++) {
   value = cubbyhole.get();
   System.out.println("Consumidor  #"+this.numero+"  obtiene:
   "+value);
   }
  }
}
```

En este ejemplo, el Productor y el Consumidor comparten datos a través de un objeto `CubbyHole` común. Observará que ninguno de los dos hace ningún tipo de esfuerzo para asegurarse de que el consumidor obtiene cada valor producido una y sólo una vez. La sincronización entre estos dos hilos realmente ocurre a un nivel inferior, dentro de los métodos `get()` y `put()` del objeto `CubbyHole`. Sin embargo, asumamos por un momento que estos dos hilos no están sincronizados y veamos los problemas potenciales que podría provocar esta situación.

Un problema sería el que se daría cuando el `Productor` fuera más rápido que el `Consumidor` y generara dos números antes de que el `Consumidor` tuviera una posibilidad de consumir el primer número. Así el `Consumidor` se saltaría un número. Parte de la salida se podría parecer a esto.

```
       . . .
```

```
Consumidor #1 obtiene: 3
Productor #1 pone: 4
Productor #1 pone: 5
Consumidor #1 obtiene: 5
    . . .
```

Otro problema podría aparecer si el consumidor fuera más rápido que el productor y consumiera el mismo valor dos o más veces. En esta situación el `Consumidor` imprimirá el mismo valor dos veces y podría producir una salida como esta.

```
    . . .
Productor #1 pone: 4
Consumidor #1 obtiene: 4
Consumidor #1 obtiene: 4
Productor #1 pone: 5
    . . .
```

De cualquier forma, el resultado es erróneo. Se quiere que el consumidor obtenga cada entero producido por el productor y sólo una vez. Los problemas como los descritos anteriormente, se llaman “condiciones de carrera”. Se alcanzan cuando varios hilos ejecutados asíncronamente intentan acceder a un mismo objeto al mismo tiempo y obtienen resultados erróneos.

Para prevenir estas condiciones en nuestro ejemplo Productor/Consumidor, el almacenamiento de un nuevo entero en `CubbyHole` por el Productor debe estar sincronizado con la recuperación del entero por parte del Consumidor. El Consumidor debe consumir cada entero exactamente una vez. El programa productor-consumidor utiliza dos mecanismos diferentes para sincronizar los hilos `Productor` y `Consumidor`; los monitores, y los métodos `notify()` y `wait`.

## 7.2. Monitores

A los objetos como `CubbyHole`, a los que acceden varios hilos, son llamados “condiciones variables”. Una de las formas de controlar el acceso a estas condiciones variables y de, por tanto, sincronizar los hilos, son los monitores. Las secciones críticas son los segmentos del código donde los hilos concurrentes acceden a las condiciones variables. Estas secciones, en Java, se marcan normalmente con la palabra reservada `synchronized`:

```
Synchronized int MiMetodo();
```

Generalmente, las secciones críticas en los programas de Java son los métodos. Sin embargo el uso indiscriminado de s`ynchronized` viola los fundamentos de la programación objetual, por lo que es mejor utilizar `synchronized` sólo a nivel de métodos**.** Java asocia un solo monitor a cada objeto que tiene un método sincronizado. En el ejemplo anterior del productor-consumidor tiene dos métodos de sincronización: `put()`, que cambia el valor de *`CubbyHole`*, y `get()`, para recuperar el valor actual. Este sería el código fuente del objeto *`CubbyHole`* utilizando las técnicas de sincronización nuevas :

```
class CubbyHole {
    private int contents;
    private boolean available = false;

    public synchronized int get() {
        while (available == false) {
            try {
                wait();
            } catch (InterruptedException e) {
            }
        }
        available = false;
        notify();
        return contents;
    }

    public synchronized void put(int value) {
        while (available == true) {
            try {
                wait();
            } catch (InterruptedException e) {
            }
        }
        contents = value;
        available = true;
        notify();
    }
}
```

La variable `contents` tiene el valor actual de `CubbyHole` y `available` indica si se puede recuperar o no el valor. Cuando `available` es verdadero, el productor aún no ha acabado de producir.

`CubbyHole` tiene dos métodos de sincronización, y Java proporciona un solo monitor para cada ejemplar de `CubbyHole` (incluyendo el compartido por el `Productor` y el `Consumidor`). Siempre que el control entra en un método sincronizado, el hilo que ha llamado al método adquiere el monitor del objeto al cual pertenece el método. Otros hilos no pueden llamar a un método sincronizado del mismo objeto mientras el monitor no sea liberado.

**Nota:** Java permite re-adquirir un monitor. En este caso se trata de los llamados *monitores re-entrantes*.

Cuando el `Productor` invoca el método `put()` de `CubbyHole`, adquiere el monitor del objeto *`CubbyHole`* y por lo tanto el `Consumidor` no podrá llamar a `get()` de *`CubbyHole`* y se quedará bloquedo (existe un método, `wait()`, que libera temporalmente el monitor). De igual forma sucede cuando el `Consumidor` invoca `get()`.

```
public synchronized void put(int value) {
        // El productor adquiere el monitor
    while (available == true) {
```

```
            try {
                wait();
            } catch (InterruptedException e) {
            }
        }
        contents = value;
        available = true;
        notify();// lo notifica al Productor
            // El productor libera el monitor
    }
    public synchronized int get() {
            // El consumidor adquier el monitor
        while (available == false) {
            try {
              wait(); // espera que el Productor invoque a
    notify ()
            } catch (InterruptedException e) {
            }
        }
        available = false;
        notify();    return contents;
            // el Consumidor libera el monitor
    }
```

# 8. Hilos Demonio (*Daemon*)

Un proceso demonio es un proceso que debe ejecutarse continuamente en modo *background* (en segundo plano), y generalmente se diseña para responder a peticiones de otros procesos a través de la red. La palabra "*daemon*" (proveniente de la palabra griega "*ghost*") es propia de UNIX, pero no se utiliza de este mismo modo en Windows. En Windows NT, los demonios se denominan "servicios". Cuando los servicios atienden peticiones, se conocen como la parte "Servidor" de una arquitectura Cliente/Servidor.

Los hilos demonio también se llaman servicios, porque se ejecutan, normalmente, con prioridad baja y proporcionan un servicio básico a un programa o programas cuando la actividad de la máquina es reducida. Un ejemplo de hilo demonio que está ejecutándose continuamente es el recolector de basura (*garbage collector*). Este hilo, proporcionado por la Máquina Virtual Java, comprueba las variables de los programas a las que no se accede nunca y libera estos recursos, devolviéndolos al sistema.

Un hilo puede fijar su indicador de demonio pasando un valor `true` al método `setDaemon()`. Si se pasa `false` a este método, el hilo será devuelto por el sistema como un hilo de usuario. No obstante, esto último debe realizarse antes de que se arranque el hilo con el método `start()`.

# 9. Conclusiones

Se pueden usar hilos Java como standard, sin tener en cuenta la plataforma en la que vayan a ejecurtarse.

La clase `Thread` es la clase responsable de producir hilos funcionales para otras clases. La interfaz `Runnable` proporciona la capacidad de añadir la funcionalidad de un hilo a una clase en lugar de derivándola de la clase `Thread`.

Para arrancar un hilo se llama a su método `start` el cual invoca al método `run` del propio hilo. Todo la tarea del hilo debe estar dentro del método `run.`

Para terminar la ejecución de un hilo de forma permanente se utiliza su método `stop.`

La clase ThreadGroup es la implementación del concepto de grupo de hilos en Java.

Java tiene un Planificador (*Scheduler)*, el cual decide que hilos deben ejecuatarse y cuales encontrarse preparados para su ejecución.

Cada hilo tiene una prioridad, que no es más que un valor entero entre 1 y 10, de modo que cuanto mayor el valor, mayor es la prioridad.

# *Apéndice A: Funciones Miembro de la Clase* `Thread`

## Sumario de constructores

`Thread()`
Realiza la reserva de memoria necesaria para la creación de un nuevo objeto `Thread`.

`Thread(Runnable` *`target`*`)`
Realiza la reserva de memoria necesaria para la creación de un nuevo objeto `Thread`.

`Thread(Runnable target, String` *`name`*`)`
Realiza la reserva de memoria necesaria para la creación de un nuevo objeto `Thread`.

`Thread(String` *`name`*`)`
Realiza la reserva de memoria necesaria para la creación de un nuevo objeto `Thread`.

`Thread(ThreadGroup` *`group`*`, Runnable` *`target`*`)`
Realiza la reserva de memoria necesaria para la creación de un nuevo objeto `Thread`.

`Thread(ThreadGroup` *`group`*`, Runnable` *`target`*`, String` *`name`*`)`
Crea un nuevo objeto `Thread` con un objeto de ejecución concreto y un nombre concreto, y se une al grupo de hilos especificado.

`Thread(ThreadGroup` *`group`*`, String` *`name`*`)`
Crea un nuevo objeto `Thread` como miembro de un grupo de hilos concreto.

## Sumario de métodos

No se citan todos los métodos. Para amplicación del sumario, se sugiere consultar la página http://java.sun.com/products/jdk/1.2/docs/api/java/lang/Thread.html

`static int activeCount()`
Devuelve el número actual de hilos activos en el grupo de hilos de este hilo.

`void checkAccess()`
Determina si el hilo actualmente en ejecución tiene permiso para modificar este hilo.

`static Thread currentThread()`
Devuelve una referencia al objeto hilo que se está ejecutando actualmente

`void destroy()`
Destruye este hilo, sin realizar ningún tipo de limpieza

`static void dumpStack()`
Imprime una traza de pila del hilo actual

`static int enumerate(Thread[] tarray)`
Copia dentro del array especificado todos los hilos activos del grupo y subgrupos de hilos del hilo en cuestión

`ClassLoader getContextClassLoader()`
Devuelve el contexto `ClassLoader` de este `Thread`

`String getName()`
Devuelve el nombre del hilo

`void setName(String name)`
Cambia el nombre de este hilo, asignándole el especificado como argumento

`int getPriority()`
Devuelve la prioridad del hilo

`ThreadGroup getThreadGroup()`
Devuelve el grupo de hilos al cual cual pertenece el hilo

`void interrupt()`
Interrumpe la ejecución del hilo

`static boolean interrupted()`
Comprueba si el hilo actual ha sido interrumpido

`boolean isAlive()`
Comprueba si el hilo está vivo

`boolean isDaemon()`
Comprueba si el hilo es un hilo daemon

`void setDaemon(boolean on)`
Establece este hilo como hilo daemon, o como hilo de usuario

`void join()`
Espera a que este hilo muera

`void join(long millis)`
Espera, como mucha `millis` milisegundos a que este hilo muera

`void run()`
Si este hilo se construyó utilizando un objeto `Runnable` de ejecución independiente, entonces el método `run` de ese objeto es invocado; en otro caso, este método no hace nada y vuelve.

`static void sleep(long millis)`
Hace que el hilo actualmente en ejecución pase a dormir temporalmente durante el número de milisegundos especificado.

`void start()`
Hace que este hilo comience la ejecución; la Máquina Virtual de Java llama al método `run` de este hilo.

`String toString()`
Devuelve una representación en formato cadena de este hilo, incluyendo el nombre del hilo, la prioridad, y el grupo de hilos.
`static void yield()`
Hace que el hilo actual de ejecución, pare temporalmente y permita que otros hilos se ejecuten (útil en sistemas con planificación de hilos no preventiva).

Se aconseja consultar las referencia citada anteriormente para consultar la descripción de métodos cuyo uso directo es inherentemente inseguro, como son `stop`, `suspend` y `resume`.

## Métodos heredados de la clase java.lang.Object

`clone`, `equals`, `finalize`, `getClass`, `hashCode`, `notify`, `notifyAll`, `wait`

# *Apéndice B: Hilos y Métodos Nativos*

Un método nativo no es más que un programa o función escrita en un lenguaje de programación distinto a Java.

Partiendo de las dos máximas de Java (portabilidad y reutilización), Java permite la integración de métodos nativos en sus aplicaciones, aunque esto suponga una limitación a la portabilidad del nuevo software.

La *Interfaz de Java Nativo* (JNI) se define como el vínculo entre la aplicación de Java y el código nativo, permitiendo el acceso a métodos nativos a través de bibliotecas compartidas de vínculos dinámicos (en Windows, las conocidas DLL). Podremos, de este modo, invocar desde una aplicación Java un método nativo del mismo modo que se hace si se está en una aplicación nativa. Así, le pasaremos los argumentos apropiados y recogeremos el valor de retorno si así procede. La JNI proporciona la posibilidad, también de incluir la JVM en aplicaciones que no sean de Java.

Por otro lado, la plataforma Java es un sistema multihilo. Debido a esto, los métodos nativos deben ser programas seguros respecto a la ejecución en multihilo. A menos que tengas conocimiento de lo contrario, como, por ejemplo, saber que el método nativo está sincronizado, debes asumir o tener en cuenta que en cualquier instante de tiempo puede haber múltiples hilos de control ejecutando el método nativo.

Los métodos nativos, por tanto, no deben modificar variables globales sensibles de forma desprotegida. Esto es, deben compartir y coordinar su acceso a variables en determinadas secciones críticas de código.

Conviene estar familiarizado con los conceptos básicos sobre hilos, en general, y más concretamente en Java, así como con la programación multihilo, para poder entrar en la programación multihilo y con métodos nativos.

# Hilos y la JNI

A continuación se presenta una serie de precauciones que deben tenerse en cuenta a la hora de programar en multihilo con métodos nativos:

- El puntero a la interfaz JNI (JNIEnv *) sólo es válido en el hilo actual. No debes pasar este puntero de un hilo a otro, o almacenarlo y utilizarlo en varios hilos. La Máquina Virtual de Java proporcionará el "mismo" puntero de interfaz cuando se invoque de forma consecutiva un método nativo desde un mismo hilo. De todos modos, diferentes hilos pasan "diferentes" punteros de interfaz a los métodos nativos.
- No se deben pasar referencias locales de un hilo a otro. En concreto, una referencia local debería convertirse en referencia inválida antes de que cualquier otro hilo pueda tener una oportunidad de utilizarla. Siempre que se dé la situación en que diferentes hilos tal vez tengan que usar la misma referencia a un objeto Java, se deberían convertir las referencias locales en referencias globales.
- Comprobar cuidadosamente el uso de variables globales. Varios hilos pueden estar accediendo a estas variables globales en el mismo instante de tiempo. Hay que asegurarse de poner los "*locks*" (cierres o controles de paso) apropiados para garantizar la seguridad.

# Sincronización de hilos en métodos nativos

La JNI proporciona dos funciones de sincronización que te permiten implementar bloques de sincronización. En Java propiamente dicho puedes implementar bloques de sincronización utilizando la sentencia `synchronized`. Por ejemplo:

```
synchronized (obj) {
    ...                    /* bloque de sincronización
*/
    ...
}
```

La Máquina Virtual de Java garantiza que un hilo debe haberse hecho con el monitor asociado con el objeto Java `obj` antes de que pueda ejecutar las sentencias interiores del bloque. De este modo, en cualquier instante de tiempo dado, sólo puede haber como mucho un hilo ejecutándose dentro del bloque de sincronización.

El código nativo puede desarrollar una sincronización equivalente sobre los objetos utilizando las funciones que la JNI le proporciona, `MonitorEnter` y `MonitorExit`. Por ejemplo:

```
...
(*env)->MonitorEnter(env, obj);
...                        /* bloque de sincronización
*/
(*env)->MonitorExit(env, obj);
...
```

Un hilo debe hacerse con el monitor asociado a `obj` antes de poder continuar su ejecución. Un hilo tiene permitida la entrada al monitor en múltiples ocasiones. El monitor contiene un contador que señala cuántas veces ha sido accedido por un hilo en concreto. `MonitorEnter` incrementa el contador cuando el hilo que entra en él ya ha entrado previamente. `MonitorExit` decrementa el contador. Otros hilos pueden entrar en el monitor cuando el contador alcanza el valor cero (0).

# *Apéndice C: Hilos en applets. Ejemplos*

Recordar que un *applet* se define como un programa Java que podemos incluir en una página HTML como si de una imagen se tratara. Un navegador que soporta la tecnología Java presentará una página que contiene un *applet*, transfiriendo previamente el código del *applet* al sistema local y ejecutándolo con su Máquina Virtual Java (JVM) (la del navegador).

Este apartado aborda tres ejemplos de uso de los hilos en *applet*s. Dado que no se hace ningún inciso en el código básico de los hilos, se recomienda consultar los apartados anteriores para entender el uso de los mismos.

El primer *applet*, AnimatorApplet, muestra cómo usar un hilo para desarrollar una tarea repetitiva. El segundo ejemplo razona el uso de hilos para llevar a cabo inicializaciones de relativo coste. Por último, se presenta un ejemplo un poco más extenso de uso de hilos con concurrencia.

## Uso de un hilo para desarrollar una tarea repetitiva

Por lo general, un *applet* que desarrolla la misma tarea una y otra vez, debería tener un hilo con un bucle `while` (o `do ... while`) que desarrolle dicha tarea. Un ejemplo típico es un *applet* que realiza una animación temporizada, como un reproductor de películas o un juego. Los *applet*s de animación necesitan un hilo que solicite un "repintado" a intervalos de tiempo regulares. Otro ejemplo sería un *applet* que lee datos proporcionados por una aplicación servidora.

Por lo general, los *applet* crean hilos para tareas repetitivas en el método `create` del *applet*. La creación de un hilo ahí supone que sea más fácil para el *applet* detener el hilo cuando el usuario abandone la página. Todo lo que necesitas hacer es implementar el método `stop` de modo que detenga el hilo del *applet*. Cuando el usuario vuelva a la página del *applet*, entonces el método `start` es invocado de nuevo, y el *applet* puede crear de nuevo un hilo para desarrollar la tarea repetitiva.

Abajo se muestra la implementación de los métodos `start` y `stop` del *applet* AnimatorApplet.

```
public void start() {
    if (frozen) {
```

```
            //No se hace nada. El usuario ha
        solicitado que paremos cambiando la
        imagen.
            } else {
                //Comienza la animación!
                if (animatorThread == null) {
                    animatorThread = new
        Thread(this);
                }
                animatorThread.start();
            }
        }

        public void stop() {
            animatorThread = null; }
```

El hecho de especificar `this` en la sentencia `new Thread(this)` indica que el *applet* proporciona el cuerpo del hilo. Esto se consigue implementado la interfaz `Java.lang Runnable`, la cual requiere que el *applet* proporcione un método `run` que conforme el cuerpo del hilo. Trataremos sobre el método `run` del *applet* `AnimatorApplet` un poco más tarde.

Obsérvese como en ningún momento, en la clase `AnimatorApplet` se invoca al método `stop` de `Thread`. Esto es porque llamar al método `Thread stop` sería como darle porrazos en la cabeza al hilo. Es una forma drástica de conseguir que el hilo detenga lo que está haciendo. En lugar de esto, puedes escribir el método `run` del hilo de tal modo que el hilo finalizará de forma gratificante cuando le des un golpecito en el hombre. Dicho golpecito en el hombro viene a ser la puesta a null de una instancia de una variable de tipo `Thread`.

En `AnimatorApplet`, esta instancia de variable se denomina `animatorThread`. El método `start` la establece como referencia al nuevo objeto `Thread` creado. Cuando el *applet* necesita matar el hilo, asigna a `animatorThread` el valor null. Esto finaliza el hilo, no convirtiéndolo en basura que pueda recogerse (no puede ser recogido como basura mientras sea ejecutable), sino porque en la sentencia del bucle, el hilo comprueba el valor de la variable `animatorThread`, continuando o finalizando dependiendo de su valor. Este es el código relevante:

```
        public void run() {
            . . .
            while    (Thread.currentThread()    ==
        animatorThread) {
       ...//Presenta un marco de animación y luego duerme.
            }
        }
```

Si `animatorThread` apunta al mismo hilo que el hilo que está actualmente en ejecución, el hilo continua ejecutándose. Si, por otro lado, `animatorThread` es null, el hilo finaliza. Si `animatorThread` hace referencia a otro hilo, entonces estaremos ante una extraña situación (algo extraño habrá ocurrido): `start` ha sido invocado tan pronto después de

`stop` (o este hilo ha empleado tanto tiempo en su bucle), que `start` ha creado otro hilo antes de que este hilo alcanzara la sentencia de comprobación de su bucle `while`. Sea cual sea la causa de esta extraña condición, este hilo debería finalizar.

## Uso de un hilo para desarrollar una inicialización de una sola vez

Si un *applet* necesita llevar a cabo alguna tarea de inicialización que suponga un cierto tiempo, se debería considerar alguna forma de realizar la inicialización en un hilo. Por ejemplo, cualquier cosa que requiera hacer una conexión de red debería ser llevada a cabo, por lo general, en un hilo en un segundo plano. Afortunadamente, la descarga de imágenes GIF y JPEG se realiza automáticamente en segundo plano utilizando hilos de los cuales no nos tenemos que preocupar.

La descarga de sonido, por desgracia, no está garantizado que se realice en segundo plano. En implementaciones actuales, los métodos del *`Applet`* `getAudioClip` no retornan hasta que hayan cargado todos los datos de audio. Como consecuencia, si se quiere descargar datos de audio, se debería crear uno o más hilos para llevarlo a cabo.

Utilizar un hilo para desarrollar una tarea de inicialización de un golpe para un *applet* es una variante del escenario clásico productor/consumidor. El hilo que desarrolla la tarea es el productor, y el *applet* es el consumidor.

# Utilización de hilos en *applets*. Otro ejemplo

Este es un ejemplo de un applet que crea un `hilo` de animación que nos presenta el globo terráqueo en rotación. Aquí podemos ver que estamos creando un `hilo` de sí mismo, lo que se conoce como **concurrencia**. Además, `animación.start()` llama al método `start()` del hilo, no del *applet*, el cual, automáticamente, llamará a la función miembro `run()` del hilo.

A continuación, el código fuente:

```
Import java.awt.*;
Import java.applet.Applet;

Public class Animacion extends Applet implments Runnable {
  Image imagenes [ ];
  MediaTracker tracker;
  int indice = 0;
  Thread animacion;

  Int maxAncho, maxAlto;
  Image offScrImage; //Componente off-screen para buffering
doble
  Graphics offScrGC;

  //Nos indicará si ya se puede pitnar
  boolean cargado = false;
```

```
    //Inicializamos el applet, establecemos su tamaño y
   //cargamos las imágenes
    public void init(){
           tracker = new MediaTracker( this );
           //Fijamos el tamaño del applet
           maxAncho = 100;
           maxAlto = 100;
           imagenes = new Image[36];

           //Establecemos el doble buffer, y dimensionamos el
//applet
           try{
               offScrImage = createImage ( maxAncho, maxAlto );
               offScrGC = offScrImage.getGraphics();
               offScrGC.setColor( Color.ligthGray );
               offScrGC.fillRect( 0, 0, maxAncho, maxAlto );
               resize( maxAncho, maxAlto );
           } catch( Exceptcion e ){
               e.printStackTrace();
               }

           //Cargamos las imágenes en un array
           for(int i=0; i<36; i++)
               {
               String fichero = new
Sring("Tierra"+String.valueOf(i+1)+".gif" );
               Imagenes[ i ] = getImage( getDocumentBase(),
fichero );
               //Registramos las imágenes con el tracker
               tracker.addImage( imagens[ i ], i);
               }

           try {
                                 //Utilizamos el tracker para
           //comprobar que todas las imágenes
               //están cargadas
               trakcer.waitForAll();
           } catch( InterruptedException e ){
               ;
               }
           cargado = true;
    }

    //Pintamos el fotograma que corresponda
    public void paint( Graphics g ) {
           if( cargado )
               g.drawImage( offScrImage, 0, 0, this );
    }
```

```
    //Arrancamos y establecemos la primera imagen
    public void start() {
           if( tracker.checkID(indice) )
               offScrGC.drawImage( imagenes[ indice ], 0, 0,
this );
           animacion = new Thread( this );
           animacion.start();
    }

    //Aquí hacemos el trabajo de animación
    //Muestra una imagen; para; muestra la siguiente imagen...
    public void run() {
           //Obtiene el identificador del thread
           Thread thActual = Thread.currentThread();

           //Nos aseguramos de que se ejecuta cuando estamos en
           //un thread y además es el actual
           while( animación != null && animación == thActual )
               {
               if( tracker.checkID( indice ) )
                   {
               //Obtenemos la siguiente imagen
                   offScrGC.drawImage( imagenes[ indice ], 0,
0, this );
                   indice++;
           //Volvemos al principio y seguimos, para el bucle
                   if( indice >= imágenes.length )
                       indice = 0;
                   }

           //Ralentizamos la animación para que parezca normal
               try {
                     animacion.sep( 200 );
                    } catch( InterruptedException e ) {
                       ;
                   }

               //Pintamos el siguiente fotograma
               repaint();
               }
           }
    }
```

En el ejemplo podemos observar más cosas. La variable `thActual` es propia de cada hilo que se lance, y la variable `animación` la estarán viendo todos los hilos. No hay duplicidad de procesos, sino que todos comparten las mismas variables; cada hilo, sin embargo, tiene su

pila local de variables, que no comparte con nadie y que son las que están declaradas dentro de las llaves del método `run()`.

La excepción `InterruptedException` salta en el caso en que se haya tenido al hilo parado más tiempo del debido. Es imprescindible recoger esta excepción cuando se están implementando hilos; tanto es así, que en el caso de no capturarla, el compilador generará un error.

# *Apéndice D: Hilos en servlets.*

Un servlets es un programa Java que se ejecuta en un servidor web. Los clientes pueden invocarlo mediante el protocolo HTTP. Del mismo modo que un applets es cargado y ejecutado en el navegador en la maquina cliente, un servlet es cargado y ejecutado por el servidor web. Así en su uso más habitual, un servlet acepta peticiones de cliente, procesa la información y devuelve los resultados, que podrán ser visualizados mediante applets, páginas HTML, etc.

Cuando un proceso se inicia en un servidor, necesita la asignación de varios recursos. El cambio entre procesos implica también mucha sobrecarga debido al cambio de contexto al tener que grabar toda la información de un proceso para volver más tarde a él. A un proceso se le puede llamar hilo pesado debido a que inicia un proceso completo, con una enorme cantidad de sobrecarga en términos de tiempo, memoria, etc...

Por el contrario, por cada petición a un servlet, se crea un hilo ligero para manejarla. Un hilo ligero es un proceso hijo, que es controlado por el proceso padre (en este caso, el servidor). En tal escenario, el cambio de contexto se hace muy fácil, y los hilos pueden pasar fácilmente de activos a inactivos, o a espera. Esto mejora sustancialmente el rendimiento de los servlets sobre los scripts CGI.

Todo servlet debe directa o indirectamente implementar su interfaz. Como cualquier otro interfaz de Java, este es también una colección de declaraciones vacías de métodos. Uno de los métodos que están declarados en el interfaz Servlet es:

```
public abstract void destroy ()
```

El método *destroy* se invoca para liberar todos los recursos solicitados. También se encarga de la sincronización de cualquier hilo pendiente. Este método se llama una sola vez, automáticamente, como el método init.

# Bibliografía

**[ 1 ] Cohn, Mike; Morgan, Bryan; T. Nygard Michael; Joshi, Dan; Trinko, Tom** "*Java, Developer's Reference*" Sams.net (1ª Editorial) 1996

**[ 2 ]** Tutorial de Java, Agustín Froufe, Universidad de Sevilla

## Referencias

**[ 3 ]** http://java.sun.com/docs/books/tutorial/essential/threads

**[ 4 ]** http://java.sun.com/docs/books/tutorial/native1.1/implementing/sync.html

**[ 5 ]** http://java.sun.com/applets/

**[ 6 ]** http://www.usenix.org/publications/java/usingjava3.html

**[ 7 ]** http://java.sun.com/products/jdk/1.2/docs/api/java/lang/Thread.html

**[ 8 ]** http://java.sun.com/products/jdk/1.2/docs/api/java/lang/Runnable.html

**[ 9 ]** http://java.sun.com/products/jdk/1.2/docs/api/java/lang/ThreadDeath.html

**[ 10 ]** http://java.sun.com/products/jdk/1.2/docs/api/java/lang/ThreadGroup.html

**[ 11 ]** http://java.sun.com/products/jdk/1.2/docs/api/java/lang/Object.html

**[ 12 ]** http://programacion.com/java

**[ 13 ]** http://www.salnet.com.ar/inv_java/Pagina11.htm