***Creación y Control de Hilos***

Antes de entrar en más profundidades en los hilos de ejecución, se propone una referencia rápida de la clase Thread.

**La clase Thread**

Es la clase que encapsula todo el control necesario sobre los hilos de ejecución (*threads*). Hay que distinguir claramente un objeto *Thread* de un hilo de ejecución o *thread*. Esta distinción resulta complicada, aunque se puede simplificar si se considera al objeto *Thread* como el panel de control de un hilo de ejecución (*thread*). La clase **Thread** es la única forma de controlar el comportamiento de los hilos y para ello se sirve de los métodos que se exponen en las secciones siguientes.

***Métodos de Clase***

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

***currentThread()***

Este método devuelve el objeto **thread** que representa al hilo de ejecución que se está ejecutando 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 nos hilos de menor prioridad no sufran inanición.

***sleep( long )***

El método *sleep()* provoca que el intérprete ponga al hilo en curso a dormir durante el número de milisegundos que se indiquen en el parámetro de invocación. Una vez transcurridos esos milisegundos, dicho hilo volverá a estar disponible para su ejecución. Los relojes asociados a la mayor parte de los intérpretes de Java no serán capaces de obtener precisiones mayores de 10 milisegundos, por mucho que se permita indicar hasta nanosegundos en la llamada alternativa a este método.

***Métodos de Instancia***

Aquí no están recogidos todos los métodos de la clase **Thread**, sino solamente los más interesantes, porque los demás corresponden a áreas en donde el estándar de Java no está completo, y puede que se queden obsoletos en la próxima versión del JDK, por ello, si se desea completar la información que aquí se expone se ha de recurrir a la documentación del interfaz de programación de aplicación (API) del JDK.

***start()***

Este método indica al intérprete de Java que cree un contexto del hilo del sistema y comience a ejecutarlo. A continuación, el método *run()* de este hilo será invocado en el nuevo contexto del hilo. Hay que tener precaución de no llamar al método *start()* más de una vez sobre un hilo determinado.

***run()***

El método *run()* constituye el cuerpo de un hilo en ejecución. Este es el único método del 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()***

Este método 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. Una forma más elegante de detener un hilo es utilizar alguna variable que ocasione que el método *run()* termine de manera ordenada. En realidad, nunca se debería recurrir al uso de este método.

***suspend()***

El método *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()***

El método *resume()* se utiliza para revivir un hilo suspendido. No hay garantías de que el hilo comience a ejecutarse inmediatamente, ya que puede haber un hilo de mayor prioridad en ejecución actualmente, pero *resume()* ocasiona que el hilo vuelva a ser un candidato a ser ejecutado.

***setPriority( int )***

El método *setPriority()* asigna al hilo la prioridad indicada por el valor pasado como parámetro. Hay bastantes constantes predefinidas para la prioridad, definidas en la clase **Thread**, tales como MIN\_PRIORITY, NORM\_PRIORITY y MAX\_PRIORITY, que toman los valores 1, 5 y 10, respectivamente. Como guía aproximada de utilización, se puede establecer que la mayor parte de los procesos a nivel de usuario deberían tomar una prioridad en torno a NORM\_PRIORITY. Las tareas en segundo plano, como una entrada/salida a red o el nuevo dibujo de la pantalla, deberían tener una prioridad cercana a MIN\_PRIORITY. Con las tareas a las que se fije la máxima prioridad, en torno a MAX\_PRIORITY, hay que ser especialmente cuidadosos, porque si no se hacen llamadas a *sleep()* o *yield()*, se puede provocar que el intérprete Java quede totalmente fuera de control.

***getPriority()***

Este método devuelve la prioridad del hilo de ejecución en curso, que es un valor comprendido entre uno y diez.

***setName( String )***

Este método permite identificar al hilo con un nombre menmónico. De esta manera se facilita la depuración de programas multihilo. El nombre mnemónico aparecerá en todas las líneas de trazado que se muestran cada vez que el intérprete Java imprime excepciones no capturadas.

***getName()***

Este método devuelve el valor actual, de tipo cadena, asignado como nombre al hilo en ejecución mediante *setName()*.

**Creación de un Thread**

Hay dos modos de conseguir hilos de ejecución (*threads*) en Java. Una es implementando el interfaz**Runnable**, la otra es extender la clase **Thread**.

La implementación del interfaz **Runnable** es la forma habitual de crear hilos. Los interfaces proporcionan al programador una forma de agrupar el trabajo de infraestructura de una clase. Se utilizan para diseñar los requerimientos comunes al conjunto de clases a implementar. El interfaz define el trabajo y la clase, o clases, que implementan el interfaz para realizar ese trabajo. Los diferentes grupos de clases que implementen el interfaz tendrán que seguir las mismas reglas de funcionamiento.

Hay una cuantas diferencias entre interfaz y clase, que ya son conocidas y aquí solamente se resumen. Primero, un interfaz solamente puede contener métodos abstractos y/o variables estáticas y finales (constantes). Las clases, por otro lado, pueden implementar métodos y contener variables que no sean constantes. Segundo, un interfaz no puede implementar cualquier método. Una clase que implemente un interfaz debe implementar todos los métodos definidos en ese interfaz. Un interfaz tiene la posibilidad de poder extenderse de otros interfaces y, al contrario que las clases, puede extenderse de múltiples interfaces. Además, un interfaz no puede ser instanciado con el operador *new*; por ejemplo, la siguiente sentencia no está permitida:

Runnable a = new Runnable(); // No se permite

El primer método de crear un hilo de ejecución es simplemente extender la clase **Thread**:

class MiThread extends Thread {

public void run() {

. . .

}

El ejemplo anterior crea una nueva clase **MiThread** que extiende la clase **Thread** y sobreescribe el método *Thread.run()* por su propia implementación. El método *run()* es donde se realizará todo el trabajo de la clase. Extendiendo la clase **Thread**, se pueden heredar los métodos y variables de la clase padre. En este caso, solamente se puede extender o derivar una vez de la clase padre. Esta limitación de Java puede ser superada a través de la implementación de *Runnable*:

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()* está definido en el interfaz **Runnable** y tiene que ser implementado. La única diferencia entre los dos métodos es que este último es mucho más flexible. En el ejemplo anterior, todavía está la oportunidad de extender la clase **MiThread**, si fuese necesario. La mayoría de las clases creadas que necesiten ejecutarse como un hilo, implementarán el interfaz **Runnable**, ya que probablemente extenderán alguna de su funcionalidad a otras clases.

No pensar que el interfaz **Runnable** está haciendo alguna cosa cuando la tarea se está ejecutando. Solamente contiene métodos abstractos, con lo cual es una clase para dar idea sobre el diseño de la clase **Thread**. De hecho, si se observan los fuentes de Java, se puede comprobar que solamente contiene un método abstracto:

package java.lang;

public interface Runnable {

public abstract void run() ;

}

Y esto es todo lo que hay sobre el interfaz **Runnable**. Como se ve, un interfaz sólo proporciona un diseño para las clases que vayan a ser implementadas. En el caso de **Runnable**, fuerza a la definición del método *run()*, por lo tanto, la mayor parte del trabajo se hace en la clase **Thread**. Un vistazo un poco más profundo a la definición de la clase **Thread** da idea de lo que realmente está pasando:

public class Thread implements Runnable {

...

public void run() {

if( tarea != null )

tarea.run() ;

}

}

...

}

De este trocito de código se desprende que la clase **Thread** también implemente el interfaz **Runnable**. *tarea.run()* se asegura de que la clase con que trabaja (la clase que va a ejecutarse como un hilo) no sea nula y ejecuta el método *run()* de esa clase. Cuando esto suceda, el método *run()* de la clase hará que corra como un hilo.

A continuación se presenta un ejemplo, [java1001.java](http://www.itapizaco.edu.mx/paginas/JavaTut/froufe/fuentes/java1001.java), que implementa el interfaz **Runnable** para crear un programa multihilo.

class java1001 {

static public void main( String args[] ) {

// Se instancian dos nuevos objetos Thread

Thread hiloA = new Thread( new MiHilo(),"hiloA" );

Thread hiloB = new Thread( new MiHilo(),"hiloB" );

// Se arrancan los dos hilos, para que comiencen su ejecución

hiloA.start();

hiloB.start();

// Aquí se retrasa la ejecución un segundo y se captura la

// posible excepción que genera el método, aunque no se hace

// nada en el caso de que se produzca

try {

Thread.currentThread().sleep( 1000 );

}catch( InterruptedException e ){}

// Presenta información acerca del Thread o hilo principal

// del programa

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

// Se detiene la ejecución de los dos hilos

hiloA.stop();

hiloB.stop();

}

}

class NoHaceNada {

// Esta clase existe solamente para que sea heredada por la clase

// MiHilo, para evitar que esta clase sea capaz de heredar la clase

// Thread, y se pueda implementar el interfaz Runnable en su

// lugar

}

class MiHilo extends NoHaceNada implements Runnable {

public void run() {

// Presenta en pantalla información sobre este hilo en particular

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

}

}

Como se puede observar, el programa define una clase **MiHilo** que extiende a la clase **NoHaceNada**e implementa el interfaz **Runnable**. Se redefine el método *run()* en la clase **MiHilo** para presentar información sobre el hilo.

La única razón de extender la clase **NoHaceNada** es proporcionar un ejemplo de situación en que haya que extender alguna otra clase, además de implementar el interfaz.

En el ejemplo [java1002.java](http://www.itapizaco.edu.mx/paginas/JavaTut/froufe/fuentes/java1002.java) muestra el mismo programa básicamente, pero en este caso extendiendo la clase **Thread**, en lugar de implementar el interfaz **Runnable** para crear el programa multihilo.

class java1002 {

static public void main( String args[] ) {

// Se instancian dos nuevos objetos Thread

Thread hiloA = new Thread( new MiHilo(),"hiloA" );

Thread hiloB = new Thread( new MiHilo(),"hiloB" );

// Se arrancan los dos hilos, para que comiencen su ejecución

hiloA.start();

hiloB.start();

// Aquí se retrasa la ejecución un segundo y se captura la

// posible excepción que genera el método, aunque no se hace

// nada en el caso de que se produzca

try {

Thread.currentThread().sleep( 1000 );

}catch( InterruptedException e ){}

// Presenta información acerca del Thread o hilo principal

// del programa

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

// Se detiene la ejecución de los dos hilos

hiloA.stop();

hiloB.stop();

}

}

class MiHilo extends Thread {

public void run() {

// Presenta en pantalla información sobre este hilo en particular

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

}

}

En ese caso, la nueva clase **MiHilo** extiende la clase **Thread** y no implementa el interfaz **Runnable**directamente (la clase **Thread** implementa el interfaz **Runnable**, por lo que indirectamente **MiHilo**también está implementando ese interfaz). El resto del programa es similar al anterior.

Y todavía se puede presentar un ejemplo más simple, utilizando un constructor de la clase **Thread**que no necesita parámetros, tal como se presenta en el ejemplo [java1003.java](http://www.itapizaco.edu.mx/paginas/JavaTut/froufe/fuentes/java1003.java). En los ejemplos anteriores, el constructor utilizado para **Thread** necesitaba dos parámetros, el primero un objeto de cualquier clase que implemente el interfaz **Runnable** y el segundo una cadena que indica el nombre del hilo (este nombre es independiente del nombre de la variable que referencia al objeto Thread).

class java1003 {

static public void main( String args[] ) {

// Se instancian dos nuevos objetos Thread

Thread hiloA = new MiHilo();

Thread hiloB = new MiHilo();

// Se arrancan los dos hilos, para que comiencen su ejecución

hiloA.start();

hiloB.start();

// Aquí se retrasa la ejecución un segundo y se captura la

// posible excepción que genera el método, aunque no se hace

// nada en el caso de que se produzca

try {

Thread.currentThread().sleep( 1000 );

}catch( InterruptedException e ){}

// Presenta información acerca del Thread o hilo principal

// del programa

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

// Se detiene la ejecución de los dos hilos

hiloA.stop();

hiloB.stop();

}

}

class MiHilo extends Thread {

public void run() {

// Presenta en pantalla información sobre este hilo en particular

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

}

}

Las sentencias en este ejemplo para instanciar objetos Thread, son mucho menos complejas, siendo el programa, en esencia, el mismo de los ejemplos anteriores.

**Arranque de un Thread**

Las aplicaciones ejecutan *main()* tras arrancar. Esta es la razón de que *main()* sea el lugar natural para crear y arrancar otros hilos. La línea de código:

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

crea un nuevo hilo de ejecución. Los dos argumentos pasados representan el nombre del hilo y el tiempo que se desea que espere antes de imprimir el mensaje.

Al tener control directo sobre los hilos, hay que arrancarlos explícitamente. En el ejemplo con:

t1.start();

*start()*, en realidad es un método oculto en el hilo de ejecución que llama a *run()*.

**Manipulación de un Thread**

Si todo fue bien en la creación del hilo, t1 debería contener un thread válido, que controlaremos en el método *run()*.

Una vez dentro de *run()*, se pueden comenzar las sentencias de ejecución como en otros programas. *run()* sirve como rutina *main()* para los hilos; cuando *run()* termina, también lo hace el hilo. Todo lo que se quiera que haga el hilo de ejecución ha de estar dentro de *run()*, por eso cuando se dice que un método es **Runnable**, es obligatorio escribir un método *run()*.

En este ejemplo, se intenta inmediatamente esperar durante una cantidad de tiempo aleatoria (pasada a través del constructor):

sleep( retardo );

El método *sleep()* simplemente le dice al hilo de ejecución que duerma durante los milisegundos especificados. Se debería utilizar *sleep()* cuando se pretenda retrasar la ejecución del hilo. *sleep()* no consume recursos del sistema mientras el hilo duerme. De esta forma otros hilos pueden seguir funcionando. Una vez hecho el retardo, se imprime el mensaje "Hola Mundo!" con el nombre del hilo y el retardo.

**Suspensión de un Thread**

Puede resultar útil suspender la ejecución de un hilo sin marcar un límite de tiempo. Si, por ejemplo, está construyendo un applet con un hilo de animación, seguramente se querrá permitir al usuario la opción de detener la animación hasta que quiera continuar. No se trata de terminar la animación, sino desactivarla. Para este tipo de control de los hilos de ejecución se puede utilizar el método*suspend()*.

t1.suspend();

Este método no detiene la ejecución permanentemente. El hilo es suspendido indefinidamente y para volver a activarlo de nuevo se necesita realizar una invocación al método *resume()*:

t1.resume();

**Parada de un Thread**

El último elemento de control que se necesita sobre los hilos de ejecución es el método *stop()*. Se utiliza para terminar la ejecución de un hilo:

t1.stop();

Esta llamada no destruye el hilo, sino que detiene su ejecución. La ejecución no se puede reanudar ya con *t1.start()*. Cuando se desasignen las variables que se usan en el hilo, el objeto *Thread* (creado con new) quedará marcado para eliminarlo y el *garbage collector* se encargará de liberar la memoria que utilizaba.

En el ejemplo, no se necesita detener explícitamente el hilo de ejecución. Simplemente se le deja terminar. Los programas más complejos necesitarán un control sobre cada uno de los hilos que lancen, el método *stop()* puede utilizarse en esas situaciones.

Si se necesita, se puede comprobar si un hilo está vivo o no; considerando vivo un hilo que ha comenzado y no ha sido detenido.

t1.isAlive();

Este método 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 el ejemplo no hay problemas de realizar una parada incondicional, al estar todos los hilos vivos. Pero si a un hilo de ejecución, que puede no estar vivo, se le invoca su método *stop()*, se generará una excepción. En este caso, en los que el estado del hilo no puede conocerse de antemano es donde se requiere el uso del método *isAlive()*.