Guía práctica de estudio 12:
Hilos

Elaborado por:
M.C. M. Angélica Nakayama C.
Ing. Jorge A. Solano Gálvez

Autorizado por:
M.C. Alejandro Velázquez Mena

Guía práctica de estudio 12: Hilos

Objetivo:

Implementar el concepto de multitarea utilizando **hilos** en un lenguaje orientado a objetos.

Objetivo:

- Implementar hilos utilizando la clase Thread.
- Implementar hilos utilizando la interfaz Runnable.

Introducción

Imaginemos una aplicación departamental que deba realizar varias operaciones complejas. Sus funciones son descargar el catálogo de precios de los productos nuevos, realizar la contabilidad del día anterior y aplicar descuentos a productos existentes.

En un flujo normal las tareas se realizarían de forma secuencial, es decir, las tareas se ejecutarán una después de la otra. Sin embargo, si la descarga de productos nuevos tarda demasiado, los descuentos no se podrían aplicar hasta que este proceso termine y, si se requiere un producto con descuento, éste no se podrá aplicar.

Lo ideal sería tener **varios flujos de ejecución** para poder realizar una tarea sin necesidad de esperar a las otras. Esto se puede lograr utilizando **hilos**, donde cada hilo representaría una tarea.

NOTA: En esta guía se tomará como caso de estudio el lenguaje de programación JAVA. Sin embargo, queda a criterio del profesor el uso de éste u otro lenguaje orientado a objetos.

Hilos (threads)

Un **hilo** es un único flujo de ejecución dentro de un proceso. Un proceso es un programa en ejecución dentro de su propio espacio de direcciones.

Por lo tanto, un **hilo** es una secuencia de código en ejecución dentro del contexto de un proceso, esto debido a que los **hilos** no pueden ejecutarse solos, requieren la supervisión de un proceso.

Hilos en Java

La Máquina Virtual Java (**JVM**) es capaz de manejar **multihilos**, es decir, puede crear varios flujos de ejecución de manera simultánea, administrando los detalles como asignación de tiempos de ejecución o prioridades de los **hilos**, de forma similar a como lo administra un Sistema Operativo múltiples procesos.

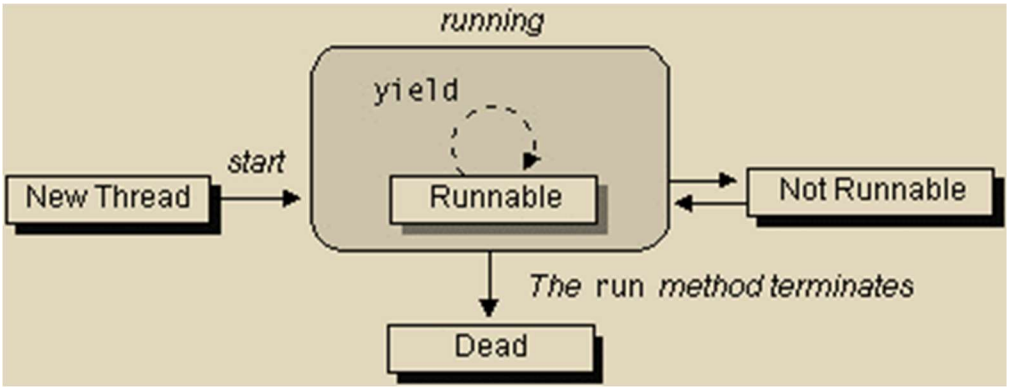
Java proporciona soporte para **hilos** a través de una interfaz y un conjunto de clases. La interfaz de Java y las clases que proporcionan algunas funcionalidades sobre hilos son:

- *Thread*
- *Runnable (interfaz)*
- *ThreadDeath*
- *ThreadGroup*
- *Object*

Tanto las clases como la interfaz son parte del paquete básico de java (*java.lang*).

Ciclo de vida de un hilo

El campo de acción de un **hilo** lo compone la etapa **runnable**, es decir, cuando se está **ejecutando** (corriendo) el proceso ligero.



Estado **new**

Un **hilo** está en el estado **new** (nuevo) 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.

Estado **runnable**

Cuando se llama al método **start** de un **hilo** nuevo, el método **run** es invocado, en ese momento el **hilo** entra en el estado **runnable** y por tanto, el hilo se encuentra en ejecución.

Estado **not running**

Cuando un **hilo** está **detenido** se dice que está en estado **not running**. Los **hilos** pueden pasar al estado **not running** por los métodos **suspend**, **sleep** y **wait** o por algún bloqueo de I/O.

Dependiendo de la manera en que el **hilo** pasó al estado **not running** es como se puede regresar al estado **runnable**:

- Cuando un hilo está **suspendido**, se invoca al método **resume**.
- Cuando un hilo está **durmiendo**, se mantendrá así el número de milisegundos especificado.
- Cuando un hilo está en **espera**, se activará cuando se haga una llamada a los métodos **notify** o **notifyAll**.
- Cuando un hilo está **bloqueado** por I/O, regresará al estado **runnable** cuando la operación I/O sea completada.

Estado **dead**

Un **hilo** entra en estado **dead** cuando ya no es un objeto necesario. Los **hilos** en estado **dead** no pueden ser resucitados, es decir, ejecutados de nuevo. Un **hilo** puede entrar en estado **dead** por dos causas:

- El método **run** termina su ejecución.
- Se realiza una llamada al método **stop**.

Planificador

Java tiene un **Planificador** (*Scheduler*) que controla todos los **hilos** que se están ejecutando en todos sus programas y decide cuáles deben ejecutarse (dependiendo de su prioridad) y cuáles deben encontrarse preparados para su ejecución.

Prioridad

Un **hilo** tiene una **prioridad** (un valor entero entre 1 y 10) de modo que cuanto mayor es el valor, mayor es la prioridad.
El **planificador** determina qué **hilo** 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. Una vez creado el **hilo** es posible modificar su **prioridad** en cualquier momento utilizando el método **setPriority**.

El planificador ejecuta primero los **hilos** de **prioridad superior** y sólo cuando éstos terminan, puede ejecutar **hilos** de **prioridad inferior**. Si varios hilos tienen la misma

prioridad, el planificador elige entre ellos de manera alternativa (forma de competición). Cuando pasa a ser "Ejecutable", entonces el sistema elige a este nuevo hilo.

Clase Thread

Es la clase en Java responsable de producir **hilos** funcionales para otras clases. Para añadir la funcionalidad de **hilo** a una clase solo se hereda de ésta.

La clase **Thread** posee el método *run*, el cual define la acción de un hilo y, por lo tanto, se conoce como el cuerpo del hilo. La clase *Thread* también define los métodos *start* y *stop*, los cuales permiten iniciar y detener la ejecución del hilo.

Por lo tanto, para añadir la funcionalidad deseada a cada hilo creado es necesario redefinir el método *run*. Este método es invocado cuando se inicia el hilo. Un hilo se inicia mediante una llamada al método *start* de la clase *Thread*. El hilo se inicia con la llamada al método *run* y finaliza cuando termina este método llega a su fin.

Ejemplo:

Clase *Hilo* que hereda de *Thread*

```
public class Hilo extends Thread {

    public Hilo (String nombre) {
        super(nombre);
    }

    public void run() {
        for(int i = 0 ; i < 5 ; i++) {
            System.out.println("IteraciOn " + (i+1) + " de " + getName());
        }
        System.out.println("Termina el " + getName());
    }

    public static void main(String[] args) {
        new Hilo ("Primer hilo").start();
        new Hilo ("Segundo Hilo").start();
        System.out.println("Termina el hilo principal");
    }

}
```

Interfaz Runnable

La interfaz **Runnable** permite producir hilos funcionales para otras clases. Para añadir la funcionalidad de hilo a una clase por medio de *Runnable*, solo es necesario implementar la interfaz.

La interfaz *Runnable* proporciona un método alternativo al uso de la clase *Thread*, para los casos en los que no es posible hacer que la clase definida herede de la clase *Thread*.

Las clases que implementan la interfaz *Runnable* proporcionan un método *run* que es ejecutado por un objeto *Thread* creado. Ésta es una herramienta muy útil y, a menudo, es la única salida que se tiene para incorporar hilos dentro de las clases.

Ejemplo:

Clase *Hilo* que implementa *Runnable*

```
public class Hilo implements Runnable {

    public void run() {
        for(int i = 0 ; i < 5 ; i++) {
            System.out.println("IteraciOn " + (i+1) + " de " +
                    Thread.currentThread().getName());
        }
        System.out.println("Termina el " +
                Thread.currentThread().getName());
    }

    public static void main(String[] args) {
        new Thread(new Hilo (), "Primer hilo").start();
        new Thread(new Hilo (), "Segundo Hilo").start();
        System.out.println("Termina el hilo principal");
    }
}
```

Clase ThreadDeath

ThreadDeath 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 hilo como un error. Así, cuando se detiene al hilo de esta forma, se detiene de forma asíncrona. El hilo morirá cuando reciba realmente la excepción *ThreadDeath*.

Sólo se debe recoger el objeto *ThreadDeath* si se necesita para realiza una limpieza específica para la ejecución asíncrona, lo cual es una situación bastante inusual. Si se recoge el objeto, debe ser relanzado para que el hilo en realidad muera.

Clase ThreadGroup

ThreadGroup se utiliza para manejar un grupo de hilos de manera simplificada (en conjunto). Esta clase proporciona una manera de controlar de modo eficiente la ejecución de una serie de hilos.

ThreadGroup proporciona métodos como *stop*, *suspend* y *resume* para controlar la ejecución del grupo (todos los hilos del grupo).

Los hilos de un grupo pueden, a su vez, contener otros grupos de hilos permitiendo una jerarquía anidada de hilos. Los hilos individuales tienen acceso al grupo, pero no al padre del grupo.

Ejemplo:
Grupo de hilos

```
public class EjThreadGroup extends Thread {

    public EjThreadGroup(ThreadGroup g, String n){
        super(g,n);
    }

    public void run(){
        for (int i = 0 ; i < 10 ; i++){
            System.out.print(getName());
        }
    }

    public static void listarHilos(ThreadGroup grupoActual) {
        int numHilos;
        Thread [] listaDeHilos;

        numHilos = grupoActual.activeCount();
        listaDeHilos = new Thread[numHilos];
        grupoActual.enumerate(listaDeHilos);
        System.out.println("\nNumero de hilos activos = " + numHilos + "\n");
        for (int i = 0 ; i < numHilos ; i++) {
            System.out.println("\nHilo activo " + (i+1) + " = "
                    + listaDeHilos[i].getName());
        }
    }

    public static void main(String[] args) {
        ThreadGroup grupoHilos = new ThreadGroup("Grupo con prioridad normal");
        Thread hilo1 = new EjThreadGroup(grupoHilos, "Hilo 1 con prioridad mAxima");
        Thread hilo2 = new EjThreadGroup(grupoHilos, "Hilo 2 con prioridad normal");
        Thread hilo3 = new EjThreadGroup(grupoHilos, "Hilo 3 con prioridad normal");
        Thread hilo4 = new EjThreadGroup(grupoHilos, "Hilo 4 con prioridad normal");
        Thread hilo5 = new EjThreadGroup(grupoHilos, "Hilo 5 con prioridad normal");
        hilo1.setPriority(Thread.MAX_PRIORITY);
        grupoHilos.setMaxPriority(Thread.NORM_PRIORITY);
        System.out.println("Prioridad del grupo = " +
                grupoHilos.getMaxPriority());

        System.out.println("Prioridad del Thread = " + hilo1.getPriority());
        System.out.println("Prioridad del Thread = " + hilo2.getPriority());
        System.out.println("Prioridad del Thread = " + hilo3.getPriority());
        System.out.println("Prioridad del Thread = " + hilo4.getPriority());
        System.out.println("Prioridad del Thread = " + hilo5.getPriority());

        hilo1.start();
        hilo2.start();
        hilo3.start();
        hilo4.start();
        hilo5.start();

        listarHilos(grupoHilos);
    }
}
```

Métodos o bloques sincronizados

Los métodos **sincronizados** (*synchronizable*) son aquellos a los que es imposible acceder si un objeto está haciendo uso de ellos, es decir, dos objetos no pueden acceder a un método **sincronizado** al mismo tiempo.

Los métodos **sincronizados** son muy utilizados en hilos (*threads*) para evitar la pérdida de información o ambigüedades en la misma. Los hilos se ejecutan como **procesos paralelos** (independientes). Cuando se ejecutan varios hilos de manera simultánea, éstos pueden intentar acceder al mismo tiempo a un método, para, por ejemplo, obtener o modificar el valor de un atributo. Como los hilos se ejecutan en paralelo, el acceso al recurso no es controlado y, por lo tanto, puede haber pérdida de información.

Estas complicaciones se pueden evitar **bloqueando** el acceso al método mientras algún proceso lo esté ejecutando. Lo anterior se puede lograr haciendo que el método en cuestión sea **sincronizado** (*synchronizable*). Así, al estar sincronizados los métodos de una clase, si un recurso está ocupando un método, éste es inaccesible hasta que sea liberado.

Ejemplo:

```
public class Cuenta extends Thread {

    private static long saldo = 0;

    public Cuenta (String nombre){
        super(nombre);
    }

    public void run() {
        if (getName().equals("Deposito 1") ||
                getName().equals("Deposito 2")) {
            this.depositarDinero(100);
        } else {
            this.extraerDinero(50);
        }
        System.out.println("Termina el " + getName());
    }

    public synchronized void depositarDinero(int cantidad) {
        saldo += cantidad;
        System.out.println("Se depositaron " + cantidad + " pesos");
                notifyAll();
    }

    public synchronized void extraerDinero(int cantidad) {
        try {
            if (saldo <= 0){
                System.out.println(getName() + " espera depOsito"
                        + "\nSaldo = " + saldo);
                        sleep(5000);
            }
        } catch (InterruptedException e) {
            System.out.println(e);
        }
        saldo -= cantidad;
        System.out.println(getName() + " extrajo " + cantidad +
                " pesos.\nSaldo restante = "+ saldo);
                notifyAll();
    }

    public static void main(String[] args) {
        new Cuenta("Acceso 1").start();
        new Cuenta("Acceso 2").start();
        new Cuenta("Deposito 1").start();
        new Cuenta("Deposito 2").start();
        System.out.println("Termina el hilo principal");
    }
}
```

Bibliografía

Sierra Katy, Bates Bert
SCJP Sun Certified Programmer for Java 6 Study Guide
Mc Graw Hill

Deitel Paul, Deitel Harvey.
Como programar en Java
Septima Edición.
México
Pearson Educación, 2008

Martín, Antonio
Programador Certificado Java 2.
Segunda Edición.
México
Alfaomega Grupo Editor, 2008

Dean John, Dean Raymond.
Introducción a la programación con Java
Primera Edición.
México
Mc Graw Hill, 2009