Anexo VII - Concurrencia - Hilos (Threads) en Java

Objetivos: Conocer y comprender el concepto de hilos, cómo se utilizan en java y cómo se aplica a ellos el concepto de concurrencia. Realizar aplicaciones sencillas en Java manejando hilos y los conceptos introducidos.

HILOS (Threads)

La Máquina Virtual de Java (JVM) es un sistema multihilo. Es decir, es capaz de ejecutar varios hilos de ejecución simultáneamente. Gestiona la asignación de tiempos de ejecución, prioridades, etc., de forma similar a como gestiona un Sistema Operativo (SO) múltiples procesos. La diferencia básica entre un proceso de SO y un **Thread** Java es que los hilos corren **dentro** de la JVM, que es **un proceso** del SO, y por tanto comparten todos los recursos, incluida la memoria y las variables y objetos allí definidos. A este tipo de procesos donde se comparten los recursos se les llama *procesos ligeros* (lightweight process).

Estos procesos ligeros o hilos (**threads**) son útiles porque permiten que el flujo del programa sea divido en dos o más partes, cada una ocupándose de alguna tarea de forma independiente. La forma más directa para crear un hilo es extender la clase **Thread**, y redefinir el método **run()**. Este método es invocado cuando se inicia el hilo, mediante una llamada al método **start()** de la clase **Thread**. Cuando el hilo se inicia, se ejecuta el método **run()**. El hilo termina cuando éste termina.

Eemplo:

```
public class ThreadEjemplo extends Thread {
	public ThreadEjemplo(String nombre) {
		super(nombre);		// el constructor de Thread le asigna este nombre al hilo
	}
	public void run() {		// redefine el método run de Thread
		for (int i = 0; i < 5 ; i++){
			System.out.println(i + " " + this.getName());
		}
		System.out.println("Termina thread " + Thread.currentThread().getName());
	}
}

public class PruebaEjemplo{
	public static void main (String [] args) {
		new ThreadEjemplo("Pepe").start();
		new ThreadEjemplo("Juan").start();
		System.out.println("Termina thread main");
	}
}
```

Si se compila y ejecuta el programa, podrá obtenerse una salida como la siguiente:

```
Termina thread main
0 Pepe
1 Pepe
2 Pepe
3 Pepe
0 Juan
4 Pepe
1 Juan
5 Pepe
Termina thread Pepe
2 Juan
3 Juan
4 Juan
5 Juan
Termina thread Juan
```

Ejecutando varias veces, se podrá observar que no siempre se produce el mismo resultado.

Notas:

- La clase **Thread** está en el paquete **java.lang**. Por tanto, no es necesario el **import**.
- El constructor **public Thread(String str)** recibe un parámetro que es la identificación o nombre del **hilo**.
- El método **run()** contiene el bloque de ejecución del **hilo**. En este caso se muestra el nombre del hilo con el método **getName()**
- El método **main** crea dos objetos de clase **ThreadEjemplo** y los inicia con la llamada al método **start(),** el cual llama al método **run()**.
- Obsérvese en la salida el primer mensaje de finalización del **hilo main**. La ejecución de los hilos es asíncrona. Realizada la llamada a **start()**, éste le devuelve control y continua su ejecución, independiente de los otros hilos.
- En la salida los mensajes de un hilo y otro se van mezclando. La JVM asigna tiempos a cada hilo.

La Interface Runnable

Proporciona un modo alternativo a la utilización de la clase **Thread**, para los casos en los que no es posible hacer que la clase definida extienda la clase **Thread**. Esto ocurre cuando dicha clase debe extender alguna otra clase. Dado que no existe herencia múltiple, la clase no puede extender a la vez la clase **Thread** y otra más. En este caso, la clase debe implementar la interface **Runnable**, variando ligeramente la forma en que se crean e inician los nuevos hilos.

El siguiente ejemplo es equivalente al anterior, pero utilizando la interface **Runnable**:

```
public class ThreadEjemplo implements Runnable {
  public void run() {
    for (int i = 0; i < 10 ; i++){
        System.out.println(i + " " + Thread.currentThread().getName());
    }
    System.out.println("Termina thread " + Thread.currentThread().getName());
  }
  public static void main (String [] args) {
       new Thread (new ThreadEjemplo(), "Pepe").start();
       new Thread (new ThreadEjemplo(), "Juan").start();
       System.out.println("Termina thread main");
  }
}
```

Nota: observe en este ejemplo **otra forma de declarar el método main**, dentro de la propia clase

Obsérvese en este caso:

• Se implementa la interface **Runnable** en lugar de extender la clase **Thread**.
• El constructor no es necesario, porque se usa luego el constructor por defecto.
• En el método main obsérvese la forma en que se crea el **hilo**. Esa expresión es equivalente a:

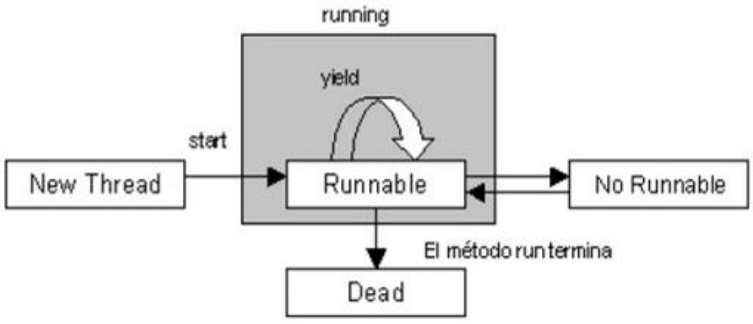
```
ThreadEjemplo ejemplo = new ThreadEjemplo();
Thread unThread = new Thread (ejemplo, "Pepe");
unTthread.start();
```

- ✓ se crea la instancia de la clase (`ThreadEjemplo`).
- ✓ se crea una instancia de la clase **Thread**, pasando como parámetros la referencia al objeto y nombre del **hilo** a crear
- ✓ se llama al método **start** de la clase **Thread**. Este método iniciará el hilo *`unThread`* y llamará a su método **run()** .

- Por último, observe la llamada al método **getName()** desde **run().** Es un método de la clase **Thread**, por lo que la clase debe obtener una referencia al thread propio (**currentThread())**. Nótese que es un método estático de la clase **Thread**, por lo que se lo invoca anteponiendo el nombre de la clase.

El Ciclo de vida de un Thread

El gráfico resume el ciclo de vida de un thread:



Cuando se instancia la clase **Thread** (o una subclase) se crea un nuevo **hilo** que está en su estado inicial ('**New Thread**' en el gráfico). En este estado es simplemente un objeto más. No existe todavía el hilo en ejecución. El único método que puede invocarse sobre él es el método **start().**

Cuando se invoca el método **start()** sobre el hilo el sistema crea los recursos necesarios, lo planifica (le asigna prioridad) y llama al método **run()**. En este momento el hilo está corriendo, se encuentra en el estado **'Runable'**.

Si el método **run()** invoca internamente el método **sleep()** o **wait()** o el hilo tiene que esperar por una operación de entrada/salida, entonces el hilo pasa al estado '**No Runnable**' (no ejecutable) hasta que la condición de espera finalice. Durante este tiempo el sistema puede ceder control a otros hilos activos.

Por último cuando el método **run()** finaliza, el hilo termina y pasa a la situación '**Dead**' (Muerto).

THREADS Y PRIORIDADES

Aunque un programa utilice varios hilos y aparentemente estos se ejecuten simultáneamente, el sistema ejecuta una única instrucción cada vez (esto es particularmente cierto en sistemas con una sola CPU), aunque las instrucciones se ejecutan concurrentemente (entremezclándose éstas). El mecanismo por el cual un sistema controla la ejecución concurrente de procesos se llama planificación (scheduling). Java soporta un mecanismo simple denominado planificación por prioridad fija (fixed priority scheduling). Esto significa que la planificación de los hilos se realiza en base a la prioridad relativa de un hilo frente a las prioridades de otros.

Si dos o más hilos están listos para ejecutarse y tienen la misma prioridad, la máquina virtual va cediendo control de forma cíclica (round-robin).

El hecho de que un hilo con una prioridad más alta interrumpa a otro se denomina 'planificación preventiva o apropiativa' (preemptive scheduling).

Cuando un hilo entra en ejecución y no cede voluntariamente el control para que puedan ejecutarse otros hilos, se dice que es un hilo egoísta (selfish thread). Algunos SO, como Windows, combaten estas actitudes con una estrategia de planificación por división de tiempos (time-slicing), que opera con hilos de igual prioridad que compiten por la CPU. En estas condiciones el SO asigna tiempos a cada hilo y va cediendo el control consecutivamente a todos los que compiten por el control de la CPU, impidiendo que uno de ellos se apropie del sistema durante un intervalo de tiempo prolongado. Este mecanismo lo proporciona el sistema operativo, no Java.

SINCRONIZACIÓN DE THREADS

La palabra reservada **synchronized** se usa para indicar que ciertas partes del código, (habitualmente un método) están sincronizadas, es decir, que un solo hilo puede acceder a dicho método a la vez. Es un mecanismo que permite establecer 'reglas de juego' para acceder a recursos (objetos) compartidos.

Un ejemplo típico en que dos procesos necesitan sincronizarse es el caso en que un hilo produzca algún tipo de información que es procesada por otro hilo. Al primer hilo denominaremos *productor* y al segundo, *consumidor*. El productor podría tener el siguiente aspecto general:

```
public class Productor extends Thread {
	private Contenedor contenedor;

	public Productor (Contenedor c) {
		this.setContenedor(c);
	}
	private void setContenedor(Contenedor c){this.contenedor = c;}
	public Contenedor getContenedor(){return this.contenedor;}

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

La clase Productor tiene un atributo *contenedor* que es una referencia a un objeto de tipo *Contenedor*, que sirve para almacenar los datos que va produciendo. El método **run** genera el dato y lo coloca en el contenedor con el método **put**. Luego espera una cantidad de tiempo aleatoria (hasta 100 milisegundos) con el método **sleep**. El productor no se preocupa de si el dato ya ha sido consumido o no. Solo lo coloca en el contenedor.

El consumidor, por su parte podría tener el siguiente aspecto general:

```
public class Consumidor extends Thread {
	private Contenedor contenedor;

	public Consumidor (Contenedor c) {
		this.setContenedor(c);
	}
	private void setContenedor(Contenedor c){this.contenedor = c;}
	public Contenedor getContenedor(){return this.contenedor;}

	public void run() {
		for (int i = 0; i < 10; i++) {
			System.out.println("Consumidor. get: " + this.getContenedor().get());
		}
	}
}
```

El constructor es equivalente al del Productor. El método run recupera el dato del contenedor con el método get y lo muestra en la consola. Tampoco el consumidor se preocupa de si el dato está disponible en el contenedor o no.

Productor y Consumidor se usarían desde un método main de la siguiente forma:

```
public class ProducTorConsumidorTest {
	public static void main(String[] args) {
		Contenedor c = new Contenedor ();
		Productor produce = new Productor (c);
		Consumidor consume = new Consumidor (c);
		produce.start();
		consume.start();
	}
}
```

Se crean los objetos de las clases Contendor, Productor y Consumidor y se inician los hilos de estas dos últimas.

La sincronización que permite a productor y consumidor operar correctamente, es decir, la que hace que consumidor espere hasta que haya un dato disponible, y que productor no genere uno nuevo hasta que haya sido consumido, está en la clase Contenedor. Se implementa de la siguiente forma:

```
public class Contenedor {
    private int dato;
    private boolean hayDato;

    Contenedor(){
        this.setDato(0);
        this.setHayDato(false);
    }

    private void setDato(int p_dato){this.dato = p_dato;}
    private void setHayDato(boolean p_hayDato){this.hayDato = p_hayDato;}

    public synchronized int get() {
        while (hayDato == false) {
            try {
                wait();   // espera a que el productor coloque un valor
            } catch (InterruptedException e) { }
        }
        this.setHayDato(false);
        notifyAll();  // notifica que el valor ha sido consumido
        return dato;
    }
    public synchronized void put(int valor) {
        while (hayDato == true) {
            try {
                wait();   // espera a que se consuma el dato
            } catch (InterruptedException e) { }
        }
        this.setDato(valor);
        this.setHayDato(true);
        notifyAll();  // notifica que ya hay dato.
    }
}
```

El atributo **dato** es el que contiene el valor que se almacena con ***put*** y se devuelve con ***get***. El atributo **hayDato** es una bandera (flag) que indica si el objeto contiene un valor o no.

En el método ***put***, antes de almacenar el valor en **dato** hay que asegurarse de que el valor anterior ha sido consumido. Si todavía hay valor (**hayDato** es true) se suspende la ejecución del hilo mediante el método **wait**. Invocando **wait** (que es un método de la clase **Object**) se suspende el hilo indefinidamente hasta que alguien le envíe una 'señal' con el método **notify** o **notifyAll (**métodos de la clase **Object**). Cuando esto se produce (en este caso, la señalización mediante **notify** lo produce el método **get**) el método continua, asume que el dato ya se ha consumido, almacena el valor en dato y envía a su vez un **notifyAll** para notificar a su vez que hay un dato disponible.

Por su parte, el método **get** chequea si hay dato disponible (no lo hay si **hayDato** es false) y si no hay espera hasta que le avisen (método **wait**). Una vez notificado (desde el método **put**) cambia el **flag** y devuelve el dato, pero antes notifica a **put** de que el dato ya ha sido consumido, y por tanto se puede almacenar otro.

La sincronización se lleva a cabo usando los métodos **wait** y **notifiAll**.

Existe además otro componente básico en el ejemplo. Los objetos productor y consumidor utilizan un recurso compartido que es el objeto contenedor. Si mientras el productor llama al método **put** y este se encuentra cambiando las variables miembro dato y **hayDato**, el consumidor llamara al método **get** y este a su vez empezara a cambiar estos valores podrían producirse resultados inesperados (este ejemplo es sencillo pero fácilmente pueden imaginarse otras situaciones más complejas).

Interesa, por tanto que mientras se esté ejecutando el método **put** nadie más acceda a las variables miembro del objeto. Esto se consigue con la palabra **synchronized** en la declaración del método. Cuando la JVM inicia la ejecución de un método con este modificador adquiere un bloqueo en el objeto sobre el que se ejecuta el método, que impide que nadie más inicie la ejecución en ese objeto de otro método que también esté declarado como **syncrhonized**. En el ejemplo, cuando comienza el método **put,** se bloquea el objeto de tal forma que si alguien intenta invocar el método **get** o **put** (ambos son **synchronized**) quedará en espera hasta que el bloqueo se libere (cuando termine la ejecución del método). Este mecanismo garantiza que los objetos compartidos mantienen la consistencia.

Este modo de gestionar los bloqueos implica que:

- Es responsabilidad del programador pensar y gestionar los bloqueos.
- Los métodos **synchronized** son más costosos en el sentido de que adquirir y liberar los bloqueos consume tiempo (este es el motivo por el que no están sincronizados por defecto todos los métodos).

```
Productor. put: 0
Consumidor. get: 0
Productor. put: 1
Consumidor. get: 1
Productor. put: 2
Consumidor. get: 2
Productor. put: 3
Consumidor. get: 3
Productor. put: 4
Consumidor. get: 4
Productor. put: 5
Consumidor. get: 5
Productor. put: 6
Consumidor. get: 6
Productor. put: 7
Consumidor. get: 7
Productor. put: 8
Consumidor. get: 8
Consumidor. get: 9
Productor. put: 9
```

Java tiene incorporado soporte para prevenir colisiones sobre un tipo de recurso: la memoria en un objeto. Dado que típicamente los datos en una clase son privados y se accede a la memoria solo a través de sus métodos, se pueden prevenir colisiones haciendo al método synchronized. Solo un hilo a la vez puede llamar un método synchronized para un objeto en particular (a pesar de que ese hilo puede llamar a más de uno de los métodos sincronizados de un objeto).
Por ejemplo, si se tienen dos métodos sincronizados simples:

```
synchronized void m1() { /* ... */ }
synchronized void m2(){ /* ... */ }
```

Cada objeto contiene un bloqueo simple (también llamado *monitor*). Esto es automáticamente parte de un objeto (no se debe escribir ningún código especial). Cuando se llama un método sychronized, este objeto es bloqueado y ningún otro método synchronized de este objeto puede ser llamado hasta que el primero termine y libere el bloqueo. En el ejemplo anterior, si m1() es llamado para un objeto, m2() no puede ser llamado para el mismo objeto hasta que m1() es completado y libere el

bloqueo. De esta forma, hay un bloqueo simple que es compartido por un método synchronized de un objeto en particular, y este bloqueo previene que memoria en común sea escrita por más de un método a la vez (o más de un hilo a la vez).
Es importante tener en cuenta que cada método que accede a un recurso crítico compartido debe ser synchronized, o no trabajará correctamente. Si se sincroniza solo uno de los métodos, entonces el otro es libre de ignorar el bloqueo y puede ser llamado con impunidad.

SINCRONIZACIÓN EXPLÍCITA DE LISTAS (ArrayLists) EN JAVA

Los **ArrayLists** no son sincronizados. Es decir que a pesar de que el método que utiliza un **ArrayList** esté sincronizado, si otro hilo por medio de otro método puede acceder a ella, podrá insertar o eliminar un elemento de la misma, aunque otro hilo ya la esté utilizando. Por esta razón, no deberían usarse **ArrayLists** en un entorno **multi-hilo** sin sincronización explícita.

Existen dos maneras de sincronización explícita de Listas en Java:

1. Utilizando el método **Collections.synchronizedList()**
2. **CopyOnWriteArrayList**

El mecanismo **CopyOnWriteArrayList** no se abordará en este tutorial.

Collections.synchronizedList()

Este método recibe como parámetro una lista **no sincronizada** (List), y retorna la misma lista pero **sincronizada**.

Ejemplo 1: crear una lista, agregar elementos, sincronizarla y mostrar en pantalla

```
import java.util.*;

public class EjemploArraySincro {
   public static void main(String[] args) {

       List<String> unaLista = new ArrayList<String>();

       unaLista.add("1");
       unaLista.add("2");
       unaLista.add("3");
       unaLista.add("4");

       // creando lista sincronizada
       List<String> listaSincro = Collections.synchronizedList(unaLista);

       // usando lista sincronizada
       System.out.println("Lista sincronizada :" + listaSincro);
   }
}
```

Alternativamente, la lista puede ser sincronizada desde el momento de su creación.

Ejemplo 2: crear una lista sincronizada, agregar elementos y mostrar en pantalla

```
import java.util.*;

public class EjemploListaSincro {
   public static void main(String[] args) {

        List<String> unaLista = Collections.synchronizedList(new ArrayList<String>());

       unaLista.add("1");
       unaLista.add("2");
       unaLista.add("3");
       unaLista.add("4");

       // usando lista sincronizada
       System.out.println("Lista sincronizada :" + unaLista);
   }
}
```