1 HILOS (THREADS) EN JAVA

1.1QUÉ ES UN THREAD

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

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

Desde el punto de vista de las aplicaciones los hilos 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. Por ejemplo un hilo puede encargarse de la comunicación con el usuario, mientras que otros actúan en segundo plano, realizando la transmisión de un fichero, accediendo a recursos del sistema (cargar sonidos, leer ficheros ...), etc. De hecho, todos los programas con interface gráfico (**`AWT`** o **`Swing`**) son multihilo porque los eventos y las rutinas de dibujado de las ventanas corren en un hilo distinto al principal.

1.2 LA CLASE THREAD

La forma más directa para hacer un programa multihilo 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`**). El hilo se inicia con la llamada al método **`run()`** y termina cuando termina éste. El ejemplo ilustra estas ideas:

```
public class ThreadEjemplo extends Thread {
    public ThreadEjemplo(String str) {
        super(str);
    }
    public void run() {
        for (int i = 0; i < 10 ; i++)
            System.out.println(i + " " + getName());
        System.out.println("Termina thread " + getName());
    }
    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
2 Juan
6 Pepe
3 Juan
7 Pepe
4 Juan
8 Pepe
5 Juan
9 Pepe
6 Juan
Termina thread Pepe
7 Juan
8 Juan
9 Juan
Termina thread Juan
```

Ejecutando varias veces el programa, se podrá observar que no siempre se ejecuta igual.

Notas sobre el programa:

- 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 del **`Thread`**.
- El método **`run()`** contiene el bloque de ejecución del **`Thread`**. Dentro de él, el método **`getName()`** devuelve el nombre del **`Thread`** (el que se ha pasado como argumento al constructor).
- El método **`main`** crea dos objetos de clase **`ThreadEjemplo`** y los inicia con la llamada al método **`start()`**(el cual inicia el nuevo hilo y llama al método **`run()`**).
- Obsérvese en la salida el primer mensaje de finalización del **`thread main`**. La ejecución de los hilos es asíncrona. Realizada la llamada al método **`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 máquina virtual asigna tiempos a cada hilo.

1.3. LA INTERFACE `RUNNABLE`

La interface **`Runnable`** proporciona un método 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, que se desea ejecutar en un hilo independiente deba extender alguna otra clase. Dado que no existe herencia múltiple, la citada clase no puede extender a la vez la clase **`Thread`** y otra más. En este caso, la clase debe implantar la interface **`Runnable`**, variando ligeramente la forma en que se crean e inician los nuevos hilos.

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

```
public class ThreadEjemplo implements Runnable {
    public void run() {
        for (int i = 0; i < 5 ; 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");
    }
}
```

Obsérvese en este caso:

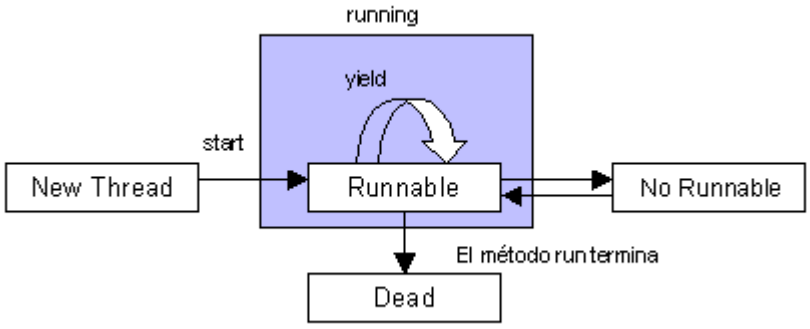
- Se implementa la interface **`Runnable`** en lugar de extender la clase **`Thread`**.
- El constructor que había antes no es necesario.
- En el **`main`** obsérvese la forma en que se crea el thread. Esa expresión es equivalente a:

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

  - Primero se crea la instancia de nuestra clase.
  - Después se crea una instancia de la clase **`Thread`**, pasando como parámetros la referencia de nuestro objeto y el nombre del nuevo thread.
  - Por último se llama al método **`start`** de la clase **`thread`**. Este método iniciará el nuevo thread y llamará al método **`run()`** de nuestra clase.
- Por último, obsérvese la llamada al método **`getName()`** desde **`run()`**. **`getName`** es un método de la clase **`Thread`**, por lo que nuestra clase debe obtener una referencia al thread propio. Es lo que hace el método estático **`currentThread()`** de la clase **`Thread`**.

1.4. 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 **`Thread`** que está en su estado inicial (`'New Thread'` en el gráfico). En este estado es simplemente un objeto más. No existe todavía el thread 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).

2 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 sus 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.

La prioridad de un hilo es un valor entero (cuanto mayor es el número, mayor es la prioridad), que puede asignarse con el método **`setPriority`**. Por defecto la prioridad de un hilo es igual a la del hilo que lo creó. Cuando hay varios hilos en condiciones de ser ejecutados (estado `runnable`), la máquina virtual elige el hilo que tiene una prioridad más alta, que se ejecutará hasta que:

- Un hilo con una prioridad más alta esté en condiciones de ser ejecutado (`runnable`), o
- El hilo termina (termina su método **`run`**), o
- Se detiene voluntariamente, o
- Alguna condición hace que el hilo no sea ejecutable (`runnable`), como una operación de entrada/salida o, si el sistema operativo tiene planificación por división de tiempos (*time slicing*), cuando expira el tiempo asignado.

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 se denomina 'planificación 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 Sistemas Operativos, 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 Sistema Operativo 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.

3 SINCRONIZACIÓN DE THREADS

Los ejemplos anteriores muestran como un programa ejecuta varios hilos de forma asíncrona. Es decir, una vez que es iniciado, cada hilo vive de forma independiente de los otros, no existe ninguna relación entre ellos, ni tampoco ningún conflicto, dado que no comparten nada. Sin embargo, hay ocasiones que distintos hilos en un programa sí necesitan establecer alguna relación entre sí, o compartir objetos. Se necesita entonces algún mecanismo que permita sincronizar hilos, así como, establecer unas 'reglas del 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 le denominaremos productor y al segundo, consumidor. El productor podría tener el siguiente aspecto:

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

    public Productor (Contenedor c) {
        contenedor = c;
    }

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

Productor tiene una variables miembro: contenedor es una referencia a un objeto **Contenedor**, que sirve para almacenar los datos que va produciendo. El método **run** genera aleatoriamente el dato y lo coloca en el contenedor con el método **put**. Después 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. Simplemente lo coloca en el contenedor.

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

```
public class Consumidor extends Thread {
    private Contenedor contenedor;
    public Consumidor (Contenedor c) {
```

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

El constructor es equivalente al del **Productor**. El método **run**, simplemente 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 ya 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();
    }
}
```

Simplemente se crean los objetos, **Contendor**, **Productor** y **Consumidor** y se inician los hilos de estos dos últimos.

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 esta en la clase Contenedor, se consigue de la siguiente forma:

```
public class CubbyHole {
    private int dato;
    private boolean hayDato = false;

    public synchronized int get() {
        while (hayDato == false) {
            try {
                // espera a que el productor coloque un valor
                wait();
            } catch (InterruptedException e) { }
        }
        hayDato = false;
        // notificar que el valor ha sido consumido
        notifyAll();
```

```
        return dato;
    }
    public synchronized void put(int valor) {
        while (hayDato == true) {
            try {
                // espera a que se consuma el dato
                wait();
            } catch (InterruptedException e) { }
        }
        dato = valor;
        hayDato = true;
        // notificar que ya hay dato.
        notifyAll();
    }
}
```

La variable miembro dato es la que contiene el valor que se almacena con **put** y se devuelve con **get**. La variable miembro **hayDato** es un flag interno que indica si el objeto contiene dato 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**. 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 lo hay espera hasta que le avisen (método **wait**). Una vez ha sido 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 pues 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 máquina virtual 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 nuestro 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 método 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).