Unidad 2. Programación multihilo
Parte 1. Gestión básica de hilos con Java

Sumario

Cuando hablamos de Java, debemos tener en cuenta que todo es un hilo, incluso la aplicación principal, y todo lo que se genera a partir de esta aplicación es un hilo, por lo que el elemento más importante de la programación concurrente en Java son los hilos. Como hemos visto en apartados anteriores, un hilo es una especie de subproceso o subtarea cuyo contexto se comparte parcialmente con el resto de hilos de la misma aplicación. Para ser más precisos, todos los hilos de una misma aplicación tienen el mismo espacio de memoria, por lo que todos comparten los mismos datos.

Podemos hacer más o menos el mismo tipo de operaciones con hilos y procesos: crearlos, sincronizarlos, destruirlos… Pero te darás cuenta en unos minutos de que el “mundo de los hilos” ofrece un amplio abanico de posibilidades que no puedes encontrar para los procesos. Esto se debe a que tu aplicación principal Java ya es un hilo, y por tanto Java está centrado en hilos.

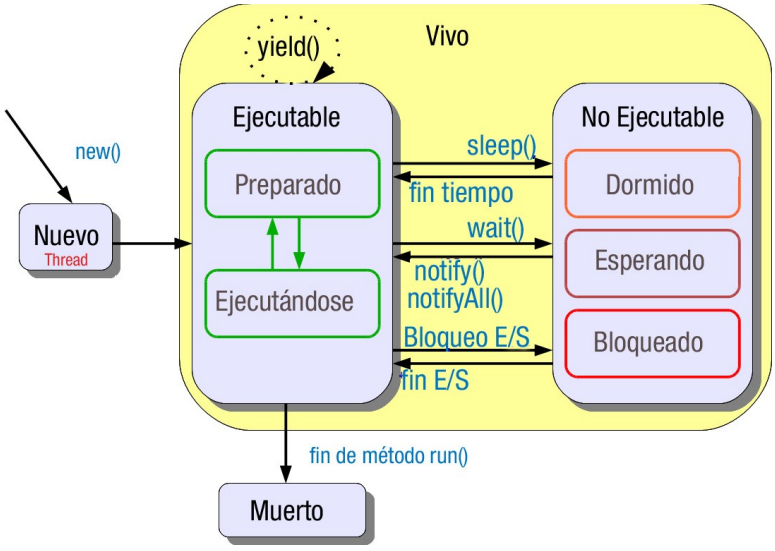
1. Estados de los hilos

Los procesos y los hilos pasan por casi los mismos estados a lo largo de su vida. Pero, además de esos estados, podemos añadir algunos más a esa lista en lo que respecta a los hilos de Java:

- **Dormido**: el hilo se ha quedado dormido debido a una llamada al método sleep que veremos más adelante. En cuanto se acabe el tiempo de sueño, volverá al estado Ready.

- **Esperando**: el hilo está esperando a que otro hilo lo reactive. Ocurre cuando los hilos están luchando por unos recursos limitados, y el que los consigue es el encargado de avisar a los demás cuando ha terminado. También veremos esta característica en otros documentos de esta unidad.

Un hilo puede llegar a estos estados desde un estado “Ejecutándose” (solo cuando está en ejecución se le puede pedir que duerma o espere), y cuando se despierta, pasa al estado “Preparado” hasta que se vuelva a ejecutar. Así que con estos nuevos estados, nuestro esquema para los estados de los hilos de Java quedaría así:



Hay otros estados que han quedado obsoletos, como “Suspendido” (en versiones anteriores de Java podíamos pausar y reanudar subprocesos desde cualquier lugar, y era potencialmente peligroso para la integridad de los datos y de la aplicación) o detenido (una forma de forzar la finalización de un subproceso, que también era potencialmente peligrosa).

2. Manejo básico de hilos: creación y lanzamiento

2.1. Definiendo de un hilo

Si queremos definir un hilo, tenemos dos formas básicas de hacerlo:

- **Heredando** de la clase **Thread**
- **Implementando** la interfaz **Runnable**

Heredando de la clase Thread

Existe una clase en Java llamada Thread, que se puede usar para crear hilos heredando de ella e implementando (sobreescribiendo) su método “run”.

```
public class MiHilo extends Thread {
    ... // atributos, constructores, métodos

    @Override
    public void run(){
        // Código a ejecutar por el hilo
    }
}
```

Implementando la interfaz Runnable

También podemos crear una clase que implemente la interfaz Runnable e incorpore el código a ejecutar en su método “run”.

```java
public class MiOtroHilo implements Runnable {
    ... // atributos, constructores, métodos
    @Override
    public void run() {
        // Código a ejecutar por el hilo
    }
}
```

En este último caso también podemos usar una clase anónima o una expresión lambda para definir el objeto Runnable.

```java
Runnable lambdaRun = () -> {
    // Código a ejecutar por el hilo
};
```

Como puedes observar, en todos los caso hay que definir (sobreescribir) un método “run” que heredamos de la clase Thread o de la interfaz Runnable. Este será el método “main” de nuestro hilo.

2.2. Creación y lanzamiento de un hilo

Para iniciar la ejecución de un hilo (recuerda, tu aplicación principal también es un hilo que se ejecuta en la JVM), no tenemos que llamar a su método “run” directamente: no habría ninguna multitarea, ya que el hilo actual (normalmente la aplicación principal) ejecutaría el método “run”, no el nuevo hilo en sí. En lugar de hacer esto, tenemos que llamar al método “start” que tiene cada hilo; luego, el sistema cargará el estado del hilo en la memoria y llamará al método “run” correctamente, de modo que podamos tener tantos hilos como necesitemos, ejecutándose todos juntos.

Si hemos definido el hilo heredando de la clase Thread, entonces podemos crear un objeto hilo y ejecutarlo con estas instrucciones (de acuerdo con el ejemplo anterior de la clase MiHilo):

```java
Thread t = new MiHilo();
t.start();
```

Si hemos definido el hilo implementado la interfaz Runnable, entonces podemos crear y ejecutar un hilo simplemente creando una nueva instancia de Thread con el objeto Runnable como parámetro. Veamos ambos ejemplos (clase y expresión lambda) creados anteriormente:

```java
// Clase normal que implementa Runnable
Thread t = new Thread(new MiOtroHilo());
t.start();

// Expresión lambda
Thread t = new Thread(lambdaRun);
t.start();
```

Haciendo esto, el objeto Thread que hemos creado sabe donde encontrar el método “run”, dentro del objeto Runnable que recibe como parámetro.

2.3. ¿Heredar de Thread o implementar Runnable?

Como te encontrarás en muchas otras situaciones a lo largo de tu carrera como programador, existen distintas formas de hacer lo mismo. En este caso podemos crear y lanzar un hilo de 2 formas: heredando de la clase Thread o implementando la interfaz Runnable. Al final, el comportamiento del hilo creado será el mismo, pero hay algunas diferencias o razones para elegir una forma y no la otra:

- Si heredas de la clase Thread, no podrás heredar de ninguna otra clase (recuerda que Java no soporta la herencia múltiple). Así que usa esta forma solo cuando tu clase para el hilo no necesite heredar de nada más. Esta opción es habitual en aplicaciones pequeñas y uy sencillas.
- En caso contrario, tienes la "opción B", es decir, implementar la interfaz Runnable (o usar clases anónimas o expresiones lambda). Recuerda, puedes implementar múltiples interfaces, pero solo puedes heredar de una clase. Por eso Java deja esta puerta abierta: en caso de que ya hayas heredado de otra clase, aún puedes tener tus hilos en ella. Esta opción es más habitual en aplicaciones complejas.

2.4. Ejemplo

Escribamos un ejemplo para ver cómo funciona un hilo. Para empezar con algo sencillo, vamos a crear un hilo que cuente del 1 al 10. Como no necesitamos heredar de ninguna otra clase, vamos a crear una clase del tipo Thread. En ejemplos posteriores utilizaremos la interfaz Runnable, para que veas cómo trabajar con ambas opciones.

Nuestro hilo básico sería así:

```
public class HiloContador extends Thread {
    @Override
    public void run(){
        for (int i = 1; i <= 10; i++)
            System.out.println("Counting " + i);
    }
}
```

Y nuestro programa principal que crea y lanza este hilo se ve así:

```
public class MainContador {
    public static void main(String[] args){
        HiloContador t = new HiloContador();
        t.start();
    }
}
```

Intenta copiar estas clases en un proyecto y ejecuta el programa principal para ver que funciona correctamente. Ahora, agreguemos algunos cambios al programa principal para ver cómo cambia su comportamiento inicial. Si ponemos esta línea al final del método main:

```
public class MainContador {
   public static void main(String[] args) {
      HiloContador t = new HiloContador();
      t.start();
      System.exit(0);
   }
}
```

¿Qué sucede cuando ejecutamos el programa nuevamente? Si lo ejecutas varias veces descubrirás que a veces cuenta hasta 10, a veces no cuenta nada... y a veces cuenta entre 1 y 10. Si se produce muy rápido y no lo ves, prueba a aumentar la cuenta a un valor mucho más alto.

Esto se debe a que el programa principal finaliza inesperadamente con el método"exit" y, luego, también se eliminan todos sus hilos. Si el hilo comenzó a ejecutarse antes de que se eliminara su hilo padre (el asociado al main), podrá contar algunos números.

Ahora cambia esa instrucción por esta:

```
public class MainContador {
    public static void main(String[] args){
        HiloContador t = new HiloContador();
        t.start();
        System.out.println("¡Hola!");
    }
}
```

Qué sucede ahora? Tu hilo cuenta hasta 10 (o el valor que sea) y en algún momento entre medio de esta cuenta aparece un mensaje "¡Hola!".

Quizás se muestre antes del número 1 o después del número 7… Depende del momento en que el programa principal llegue al procesador para mostrar su mensaje.

Finalmente, intenta llamar al método "start" nuevamente después de la primera llamada:

```
public class MyMainCounter {
    public static void main(String[] args){
        MyCounterThread t = new MyCounterThread();
        t.start();
        t.start();
    }
}
```

Ahora verás que se genera una excepción del tipo `IllegalThreadStateException`. No podemos llamar al método "start" más de una vez, en todo caso tendríamos que crear un nuevo objeto Thread.

2.5. Conclusiones

De este ejemplo podemos sacar algunas conclusiones importantes:

- Cuando lanzamos un hilo desde nuestra aplicación principal, este comienza su ejecución de forma independiente y paralela.

- Cuando nuestra aplicación principal finaliza correctamente, nuestro hilo continúa ejecutando su tarea hasta que finaliza.
- Cuando nuestra aplicación principal se ve forzada a finalizar, nuestro hilo finaliza también inesperadamente. Para ser más precisos, si algún hilo de nuestra aplicación llama al método System.exit, todos los hilos finalizarán su ejecución.
- No hay forma de saber el orden exacto en el que la aplicación principal y sus hilos producirán sus resultados. Depende del planificador de tareas del sistema operativo. De todos modos, aprenderemos a dar más tiempo de CPU a algunos hilos a expensas de los demás en breve, y también a sincronizar o coordinar hilos para producir resultados en un orden determinado.
- Después de lanzar un hilo, no podremos volver a llamar a su método de inicio. Pero sí podremos llamar a otros métodos para obtener su estado y algunas otras características, como veremos más adelante.

Actividades a realizar

a) Modifica el ejemplo del contador para que funcione con Runnable en vez de heredando de Thread.

2) Ahora vas a realizar un proyecto Java, llamado HilosMultiplicadores, para resolver el siguiente problema. Define una clase que herede de Thread y que tenga un número como atributo. Asigna un valor a este número a través del constructor de la clase. En el método run, el hilo tiene que mostrar la tabla de multiplicar de su atributo. Luego, desde la aplicación principal (es decir en main), crea 10 hilos (cada uno con un número diferente pasado por parámetro al atributo) y ejecútalos todos al mismo tiempo. Debes observar cómo los mensajes de un hilo se mezclan con los mensajes de otros hilos de forma desordenada. Por ejemplo…

```
1 x 0 = 0
1 x 1 = 1
3 x 0 = 0
4 x 1 = 4
```

Tras eso, debes modificar el programa para hacer que los hilos muestren sus resultados de forma ordenada, no mezclada. Para ello, puedes optar por usar `Thread.sleep()` o por usar el método `.join()` (más adelante, y en Aules, tienes ejemplos para usar .join)

3. Información básica de los hilos

Existen algunos métodos y propiedades útiles en la clase Thread para obtener y modificar información sobre un hilo. Por ahora, nos centraremos en tres de ellos:

- Cómo configurar y obtener el nombre del hilo
- Cómo obtener el estado del hilo
- Cómo obtener el identificador de un hilo

3.1. Configuración y obtención del nombre del hilo

Si deseas darle un nombre a los hilos, simplemente puedes agregar un atributo name a la clase (ya sea extendiendo Thread o implementando Runnable). Pero hay algunos métodos en la clase Thread que nos permiten establecer y obtener este nombre sin agregar ninguna información adicional: el método setName establece el nombre de nuestro hilo, el método getName obtendrá este nombre.

```
Thread t = new MyCounterThread();
t.setName("MyThread A");
t.start();
System.out.println("Thread " + t.getName() + " has been launched.");
```

En este ejemplo, hemos creado un hilo, le hemos asignado un nombre y luego lo hemos mostrado. Si queremos obtener o asignar el nombre del hilo dentro del hilo mismo (por ejemplo, desde el método “run” del hilo), podemos llamar al método currentThread para obtener un objeto Thread que apunte al hilo actual y luego obtener o asignar su nombre.

```
@Override
public void run() {
    Thread.currentThread().setName("AAA");
    ...
    System.out.println(Thread.currentThread().getName());
}
```

Si ejecutas un hilo desde una instancia Runnable, puedes establecer el nombre directamente al crear el hilo como segundo parámetro en el constructor.

```
Runnable counterRun = () -> {
    System.out.println(Thread.currentThread().getName() + " running");

   for (int i = 1; i <= 10; i++)
        System.out.println("Counting " + i);
};

Thread t = new Thread(counterRun, "CounterThread");
t.start();
```

3.2. Obtención del estado del hilo

También podemos obtener el estado actual del hilo en cualquier momento. Para gestionar estos estados, existe una enumeración interna denominada Thread.State y un método getState en la clase Thread . El siguiente ejemplo lanza un hilo y, unas líneas más abajo, comprueba su estado actual:

```
Thread t = new MyCounterThread();
t.start();
...
Thread.State st = t.getState();
```

Lo que devuelve el método getState puede ser uno de los siguientes estados, que se representan mediante constantes en la enumeración Thread.State: NEW, RUNNABLE, BLOCKED, WAITING, TIMED_WAITING o TERMINATED . Por ejemplo, si queremos comprobar si el hilo ha finalizado su tarea, podemos hacerlo de la siguiente manera:

```
if (st == Thread.State.TERMINATED)
    System.out.println("Thread is terminated.");
```

También podemos comprobar si un hilo ha terminado su tarea con el método isAlive (de la clase Thread):

```
if (!t.isAlive())
    System.out.println("Thread is terminated.");
```

3.3. Obtención del identificador del hilo

La máquina virtual Java asigna un identificador único a cada hilo que se crea. Si queremos obtenerlo, solo tenemos que llamar al método getId de la clase Thread:

```
@Override
public void run() {
    ...
    System.out.println("Thread #" + Thread.currentThread().getId());
}
```

4. Los métodos sleep y yield

En esta sección vamos a aprender a poner a dormir hilos, o pedirles que dejen libre el procesador.

4.1. El método sleep

Cuando llamamos al método sleep, el hilo que lo está llamando se duerme automáticamente (es decir, pausa su ejecución), hasta que transcurra el número de milisegundos indicado en el parámetro. Esto es útil para dejar libre el procesador para otros hilos si nuestro hilo actual no tiene nada que hacer por ahora, o si queremos ayudar a mejorar la concurrencia entre nuestros hilos.

El método sleep es un método estático de la clase Thread, por lo que para llamarlo sólo tenemos que añadir esta instrucción en la posición donde queremos que duerma el hilo, con el tiempo de sueño deseado en milisegundos:

```
  Thread.sleep(2000);
```

Este ejemplo pone al hilo que ejecuta la instrucción a dormir durante 2 segundos (2000 milisegundos). De hecho, necesitamos capturar una posible excepción que puede lanzarse al usar este método:

```
try {
    Thread.sleep(2000);
} catch (InterruptedException e) {
    ...
}
```

Observa que, incluso si usamos un objeto Thread para llamar a este método…

```
public static void main(String[] args) {
    Thread t = new MyThread();
    t.start();
    t.sleep(2000);
}
```

El hilo representado por el objeto t no dormirá, pero nuestra aplicación principal sí. **Recuerda**: el hilo desde el que se llama al método es el que duerme.

En cuanto a los milisegundos, también podemos usar la clase TimeUnit (del paquete java.util.concurrent) y sus propiedades para especificar otra unidad de tiempo, que se convertirá automáticamente a milisegundos. Por ejemplo, si queremos que nuestro hilo duerma 5 segundos, también podemos hacerlo así:

```
import java.util.concurrent.TimeUnit;
...
try {
    TimeUnit.SECONDS.sleep(5);
} catch (InterruptedException e) { ... }
```

Puedes echar un vistazo a la API de Java para ver más constantes que puedes usar desde la clase TimeUnit, como MINUTOS, HORAS, etc. La llamada a TimeUnit.sleep genera una llamada a Thread.sleep de hecho, con la conversión adecuada a milisegundos.

4.2. El método yield

El método yield es similar al método sleep, pero no necesita un número de milisegundos como parámetro. Simplemente deja el procesador libre para que el planificador de tareas lo asigne a otro hilo. Si ningún hilo está esperando al procesador, entonces el hilo que cedió lo recupera.

Este método es estático, y también se aplica al hilo que lo llama. No lanza ninguna excepción cuando se le llama, por lo que podemos usarlo simplemente así:

```
Thread.yield();
```

Hay un problema potencial al usar el método yield: el planificador de tareas de la JVM puede ignorar esta instrucción, por lo que no podemos estar seguros de que un hilo cederá cuando se lo pidamos.

4.3. Ejemplo

En este ejemplo, vamos a definir un hilo (implementando la interfaz Runnable a través de una expresión lambda) que cuenta de A a Z, durmiendo 100 ms después de mostrar cada letra. El programa principal esperará a que este hilo termine verificando su estado después de cada iteración.

```
public static void main(String[] args){
    Thread t = new Thread(() -> {
        for (char c = 'A'; c <= 'Z'; c++) {
            System.out.println(c);
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                System.err.println("Error: Thread interrupted");
            }
        }
    });

    t.start();

    do {
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) { }
    } while (t.isAlive());
    System.out.println("Thread has finished, and so do I");
}
```

Observe que el programa principal solo duerme unos pocos milisegundos (pueden ser 50, 100, 200... no importa) en cada iteración. Solo queda esperar a que termine el hilo, no tiene nada que hacer, así que mejor deja libre el procesador durmiendo o cediendo.

Ejercicio

Resuelve el siguiente problema. Crea un proyecto llamado CarreraHilos. Define una clase con propiedades para ejecutarse en un hilo y, en el programa principal, crea 3 objetos de esta clase, cada uno con un nombre que será “A”, “B” ó “C” y que deberán contar del 1 al 1000. El programa principal tendrá que esperar a que finalicen todos sus hilos, deberá dormir 100 ms después de cada iteración y escribirá la cuenta actual para cada hilo. Veamos un ejemplo de salida:

```
Hilo A: 27  Hilo B: 72   Hilo C: 57
Hilo A: 111 Hilo B: 114  Hilo C: 107
...
```

5b) Modifica el proyecto anterior para que, en cuanto el hilo A llegue al valor 700 en su conteo, finalice. Para ello debes emplear una variable bandera de tipo booleano que controle su ejecución. Cuando lo hayas conseguido de esta forma, modifica el proyecto nuevamente para que, ahora, en vez de controlar la finalización del hilo con una variable booleana, llames al método "interrupt()". También tendrás que usar el método "`is`Interrupted()".

5. Finalización e interrupción de hilos

Existen dos formas de forzar a un hilo a finalizar su tarea: utilizando indicadores booleanos para indicarle al hilo que debe detenerse cuando verifique dichos indicadores, o utilizando interrupciones para hacer que se detenga.

5.1. Finalización de hilos con indicadores booleanos

Los hilos finalizan su tarea cuando ejecutan todas las instrucciones de su método run. No hay forma de que podamos detener un hilo en un momento determinado (había un método stop y un método destroy en versiones anteriores de Java, pero ahora están obsoletos). Incluso si ponemos su variable a null, los recursos del hilo se mantendrán bloqueados.

Pero no te preocupes. Aún tenemos un método para pedirle a un hilo que finalice, aunque no termine en este preciso momento. Este método se aplica a hilos que tienen algún tipo de bucle en su método run. Si implementamos este bucle correctamente, podemos utilizar un indicador booleano para indicarle al hilo si puede continuar o si debe finalizar.

Veamos este método con un ejemplo. Si definimos una subclase de hilo como esta:

```
public class KillableThread extends Thread {
    boolean finish = false;

    public void setFinish(boolean finish) {
        this.finish = finish;
    }

    @Override
    public void run() {
        while (!finish) {
            ... // Thread task
        }
    }
}
```

Luego podemos crear y lanzar un hilo desde nuestra aplicación principal, y pedirle al hilo que finalice con su método setFinish:

```
public static void main(String[] args) {
    KillableThread kt = new KillableThread();
    kt.start();
    ...
```

```
    if (someCondition)
        kt.setFinish(true);
}
```

Tan pronto como el subproceso llegue al principio del bucle y verifique que la variable finish es verdadera, finalizará su método run.

Ejercicio

Crea un proyecto llamado ThreadRaceKilled basado en el proyecto creado anteriormente. Modifica la aplicación principal para que, en cuanto el hilo A llegue a 700, se le pida que finalice (con una variable booleana). Siéntete libre de añadir todo el código que necesites a cada clase del proyecto.

5.2. Finalización de hilos con interrupciones

Existe una segunda forma de finalizar un hilo. Consiste en llamar al método de interrupción de dicho hilo. Veámoslo en el siguiente ejemplo:

```
public static void main(String[] args) {
    Thread t = new Thread(() -> {
        try {
            while (!Thread.currentThread().isInterrupted()) {
                System.out.println("Running");
                Thread.sleep(100);
            }
        } catch (InterruptedException e) { }
        System.out.println("Finished by an interruption");
    });

    t.start();
    try {
        // Wait for a while...
        Thread.sleep(1000);
    } catch (InterruptedException e) { }

    t.interrupt();
}
```

En este ejemplo creamos un hilo que comprueba en cada bucle si ha sido interrumpido (con el método isInterrupted, de la clase Thread). Si no, sigue ejecutándose (es decir, muestra un mensaje y duerme 100 ms). Desde el hilo principal esperamos algunos milisegundos y luego interrumpimos el hilo creado anteriormente con el método interrupt. Este método provoca una InterruptedException que hace que el hilo vaya a la sección catch y finalice el método run.

La InterruptedException solo se lanza debido a la llamada sleep en Runnable. Si no llamamos a sleep, wait, join o cualquier otro método que pueda lanzar esta excepción, la estructura try...catch no sería necesaria y este hilo finalizaría llamando a su método isInterrupted. Ten en cuenta que un hilo decide si responde a la interrupción o no, utilizando su método isInterrupted y/o capturando las posibles excepciones que se puedan lanzar.