Programación asíncrona

Conceptualmente, la programación Asíncrona es independiente del lenguaje de programación, siempre y cuando la admita. Cada uno de ellos suministrará una serie de herramientas para poder crear y gestionar hilo. Por tanto:

- **Programación asíncrona**: o explícita es aquella en la que en un programa pueden suceder varias cosas al mismo tiempo y bajo tu control. El programa continúa ejecutándose incluso después de iniciar una tarea adicional, avisa cuando tiene un resultado e incluso prosigue realizando otras tareas mientras tanto.

- **Programación sincrónica**: o implícita (o secuencial), es donde las cosas suceden una tras otra. El programa se detendrá mientras inicia una nueva tarea, y solo prosigue su ejecución cuando obtiene el resultado.

Java utiliza los siguientes mecanismos para crear, ejecutar o sincronizar estos hilos.

Interfaz Runnable

Una clase que implemente el interfaz *Runnable* está concebida para ejecutarse en un hulo (*Thread*). Sólo tiene un método, el **run ()**, y no tiene argumentos ni retorno. El código escrito en el run () se ejecuta de manera asíncrona, por lo tanto, el código del hilo principal no se detiene.

Ya hemos visto anteriormente cómo crear un hilo mediante la interfaz *Runnable*.

Clase Thread

Esta clase representa un hilo de ejecución. Cuando una clase hereda de *Thread* puedes implementar el método run () y ejecutarlo de forma asíncrona.

Ya hemos visto anteriormente cómo crear un hilo mediante la clase *Thread*.

Adicionalmente, thread ofrece una serie de métodos para gestionar hilos. Los más importantes son:

start	Inicia el bloque asíncrono (run ())
join	Detiene el hilo hasta que finalice otro hilo
Sleep	Detiene un hilo temporalmente
isAlive	Comprueba si el hilo está vivo
setPriority	Cambia la prioridad de un hilo. El caso que le haga depende del sistema operativo.

Interrupciones

En programación, una interrupción es la suspensión temporal de la ejecución de un programa o de un hilo. Por norma general, las interrupciones pertenecen al Sistema Operativo, y son generadas por los diferentes dispositivos: disco duro, memoria, etc.

En Java, una *interrupción* significa que el Hilo debe detenerse para hacer otra cosa. Es cosa del programador decidir qué hay que hacer cuando sucede una interrupción, siendo lo habitual detener el Hilo. Dado que es Java, las interrupciones se capturan mediante un bloque try-catch. La excepción que se captura es la **InterruptedException**.

Por ejemplo, vamos a realizar un programa que cuenta hasta cinco segundos dentro de un bucle infinito. Generamos la interrupción y la capturamos.

```
public class Interrupcion extends Thread {

    @Override
    public void run() {
        int contador = 0;
        while (true) {
            contador++;
            try {
                System.out.println(contador);
                if (contador == 5)
                    this.interrupt();
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                System.out.println("- FIN -");
                return; // Finaliza el hilo
            }
        }
    }

    public static void main(String[] args) {
        new Interrupcion().start();
    }
}
```

Ejercicio 3

Crea el programa **HiloDurmiente**. Este programa genera 5 hilos en una sola clase que hereda de Thread. Cada hilo debe de tener un nombre (setName ()). En sus métodos run (), habrá un bucle infinito que diga “Soy el bucle X y estoy trabajando”. Se detendrá aleatoriamente la ejecución del hilo de 1 a 10 segundos.

Problemas y Soluciones de la Programación concurrente

Si bien en Java es razonablemente sencillo crear tareas separadas, aún sufre de los dos problemas característicos de la programación concurrente. Por tanto, tenemos que entender los siguientes conceptos:

- **Recursos Compartidos**: dos hilos pueden tener que acceder a la vez a la misma variable estática, a un fichero, o a un elemento del sistema. Esto puede provocar multitud de problemas. El más habitual es cuando un hilo pretende leer una variable mientras otro quiere escribir en ella. Hemos visto ya un ejemplo de ello anteriormente en el programa **RatónUnicoBucle**.

- **Dependencias**: No todas las tareas se pueden ejecutar en un entorno multihilo. Para poder hacerlo, hay que estar seguros de que los bloques de código se van a poder ejecutar en paralelo son independientes y el orden de su ejecución es irrelevante. La existencia de dependencias (de datos, de flujos o de datos) impiden la programación concurrente salvo que existan mecanismos de sincronización.

- **Condiciones de Bernstein**: El cumplimiento de las Condiciones de Bernstein determinan si dos segmentos de código pueden ser ejecutados en paralelo. Para ello, ambos tienen que ser independientes entre sí. Dos códigos son independientes si:

    - Las entradas de C2 son distintas de las salidas de C1. Si no, tenemos una **dependencia de flujo**.

    - Las entradas de C1 son distintas de las salidas de C2. Si no, tenemos una **antidependencia**.

    - Las salidas de C1 son distintas de las salidas de C2. Si no, tenemos una **dependencia de salida.**

- **Acción y acceso atómicos**: Una sección atómica es aquella que se ejecuta sin interrupciones, de una sola vez. En Java, las acciones atómicas son por ejemplo leer o escribir un tipo de dato primitivo (salvo long y double).

- **Sección Crítica**: La sección crítica de un programa multihilo es el bloque de código que accede a recursos compartidos, por lo que solamente un hilo puede acceder a dicho recurso en cada instante de tiempo. Determinar esta sección crítica permite sincronizar correctamente un programa. Conseguir que sólo un hilo acceda a la sección crítica se le llama **exclusión mutua**.

- **Seguridad en hilos (Thread Safety)**: Son las clases o elementos de software que están diseñadas para ser ejecutadas de forma segura en Hilos.

Problemas de la Programación Concurrente

Problemas que surgen como consecuencia de la programación multihilo:

- **Interbloqueo o deadlock**: tenemos dos o más Hilos que se bloquean entre sí. Es un error habitual cuando tienes a un hilo esperando a que otro termine, pero éste a su vez está esperando a que el otro lo haga antes.

- **Muerte por inanición**: sucede cuando alteramos las prioridades de los hilos y uno de ellos jamás llega a ejecutarse porque no llega a tener acceso a la CPU.

- **Condiciones de carrera**: sucede cuando dos bloques tienen dependencia entre ellos pero no se ejecutan en el orden correcto.

- **Inconsistencia de memoria**: sucede cuando dos o más hilos tienen valores diferentes para la misma variable.

- **Condiciones deslizadas**: sucede cuando se está evaluando si un bloque tiene que ejecutarse o no, y entonces otro hilo cambia la condición. Esto puede provocar que se ejecuten bloques de instrucciones no deseados.