Unión Europea
Fondo Social Europeo
El FSE invierte en tu futuro

UD 2 Sincronizar HILOS (PSP)

CFGS Desarrollo de Aplicaciones Multiplataforma

Pepe

GENERALITAT VALENCIANA
Conselleria d'Educació, Investigació, Cultura i Esport

GOBIERNO DE ESPAÑA
MINISTERIO DE EDUCACIÓN, CULTURA Y DEPORTE

Continguts

1 EL MÉTODO JOIN

1.1 Objetivo del método

El método *join()* provoca que el hilo que hace la llamada espere la finalización de otros hilos. Por ejemplo, si en el hilo actual escribo "fil1.join()", el hilo se queda en espera hasta que muera el hilo sobre el que se realiza el join(), en este caso, el fil1.

```
public class Hilo extends Thread {

    public void run() {
        for(int i=0; i<=5; i++)
        System.out.println("Hola soy un hilo "+Thread.currentThread().
            getName()+" "+i);
    }

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Hilo fil1=new Hilo();

        fil1.start();

        try {
            fil1.join();
        } catch (InterruptedException e) {
        // TODO Auto-generated catch block
            e.printStackTrace();
        }


        //imprime el nombre del hilo en la Salida (función getName())
        System.out.println("El hilo se llama " + Thread.currentThread() +
            "\n");

    }

}
```

```
Hola soy un hilo Thread-0 0
Hola soy un hilo Thread-0 1
Hola soy un hilo Thread-0 2
Hola soy un hilo Thread-0 3
Hola soy un hilo Thread-0 4
Hola soy un hilo Thread-0 5
El hilo se llama Thread[#1,main,5,main]
```

(a) con join

```
Hola soy un hilo Thread-0 0
Hola soy un hilo Thread-0 1
El hilo se llama Thread[#1,main,5,main]

Hola soy un hilo Thread-0 2
Hola soy un hilo Thread-0 3
Hola soy un hilo Thread-0 4
Hola soy un hilo Thread-0 5
```

(b) sin join

Gracias al método join() le decimos al programa Principal que tiene que esperarse a que finalice el hilo

fil1 para poder mostrar el mensaje de “El hilo se llama”, fíjate en la diferencia entre utilizar join y no utilizarlo.

Otra opción seria utilizar el método `run`, el hilo principal quedaria bloqueado hasta que terminen los hilos

```
public static void main(String[] args) {
        // TODO Auto-generated method stub
        Thread t1 = new Thread(() -> System.out.println("Hola 1"));
        Thread t2 = new Thread(() -> System.out.println("Hola 2"));
        Thread t3 = new Thread(() -> System.out.println("Hola 3"));

        System.out.println("Inicio");
        t1.run();
        t2.run();
        t3.run();
        System.out.println("Fin");


    }
```

```
Inicio
Hola 1
Hola 2
Hola 3
Fin
```

(c) con run

2 EL MÉTODO INTERRUPT

Interrumpimos, pramos la ejecución de un hilo.

```
public class Hilo extends Thread {

    public void run() {
        for(int i=0; i<=5; i++) {
            System.out.println("Hola soy un hilo "+Thread.currentThread().
               getName()+" "+i);
            try {
                Thread.sleep(2000);
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
```

```
                //e.printStackTrace();
                System.out.println("I'm resumed");
                return;
            }
        }
    }

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Hilo fil1=new Hilo();

        fil1.start();

        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }//pausar el hilo principal

      fil1.interrupt(); // si no ha finalizado fil1 lo paro yo.

    }

}
```

Veamos en un ejemplo práctico donde un hilo titene que ejcutar varias tareas *(los mensajes)* en un periodo de tiempo y si sobre pasa un tiempo estipulado, *deadline*, abortamos.

```
public class Interrupciones {

    public static void main(String[] args) throws InterruptedException {
        // TODO Auto-generated method stub
        List<String> messages =Arrays.asList("Mensaje1","Mensaje2","
           Mensaje3","Mensaje4");
        long starTime= System.currentTimeMillis();
        long deadLine =8000;

        Thread fil2 = new Thread(()->{
            Thread.currentThread().setName("fil2");
            System.out.println(Thread.currentThread().getName()+"
               Iniciando Trabajo");
            try {
                for(String msg:messages) {
                    Thread.sleep(4000);
                    String nom= Thread.currentThread().getName();
                    System.out.println( nom+" "+msg);
                    //print(msg);
                }
                System.out.println(Thread.currentThread().getName()+" Fin
                   Trabajo");
```

```
            }catch(InterruptedException e) {
                //throw new RuntimeException(e);
                System.out.println(" Hilo Parado");
            }

        });

        fil2.start();

        while(fil2.isAlive() ) { //comprobamos que el hilo esta vivo...
            podria finalizaar antes
            System.out.println(Thread.currentThread().getName()+"
                Esperando");
            fil2.sleep(1000);
            long endTime= System.currentTimeMillis();
            long lapso= endTime-starTime;

            if(lapso>deadLine && fil2.isAlive()) { //comprobamos los
                tiempos  y si está vivo... en caso afirmativo abortamos.
                System.out.println(Thread.currentThread().getName()+"
                    Sacabo");
                fil2.interrupt();
            }

        }

        System.out.println(Thread.currentThread().getName()+" fin");

    }

    static void print(String message) {
        String nom= Thread.currentThread().getName();
        System.out.format("%s: %s%n", nom, message);
    }

}
```

3 SINCRONIZACIÓN ENTRE HILOS

3.1 Métodos sincronizados

Los métodos sincronizados se encargan de definir la *sección crítica*. La llamamos así por ser la sección de *código del objeto compartido* a la cual intentarán acceder varios hilos al mismo tiempo, pero que hemos de sincronizar para que en esta zona solo haya 1 hilo ejecutándose.

Las *secciones críticas* son aquellas secciones de código que no pueden ejecutarse concurrentemente, pues en ellas se encuentran los recursos o información que comparten diferentes hilos y que por tanto

pueden ser problemáticas. Veamos un ejemplo. En él, se pone de manifiesto el problema conocido como la **condición de carrera**, que se produce cuando varios hilos acceden a la vez a un mismo recurso, por ejemplo a una variable, cambiando su valor y obteniendo de esta forma un valor no esperado de la misma.

tiempo	`thread1 { c = c + 1; }`	`thread2 { c = c - 1; }`
t0	c = 100	
t1	Lee 100	
t2	Incrementa a 101	
t3		Lee 100
t4	Escribe 101	
t5		Decrementa a 99
t6		Escribe 99
final	c = 99	

Cada vez que un hilo intenta acceder a un método sincronizado, pregunta si este ya está en posesión de algún hilo. Si es así, el hilo se espera y, cuando el primer hilo termina su ejecución o se suspende, pasa a ejecutarse el nuevo hilo. Se consiguen métodos sincronizados añadiendo la palabra **synchronized** a la cabecera del método del objeto que se comparte, quedando del siguiente modo:

```
public class AumentarContador{
    private int c = 0

    public synchronized void incrementa(){
        c = c +1;
    }
}
```

- Añadir synchronized a un método significará que:
  - Hemos creado un monitor,sección cr´titca, asociado al objeto.
  - Sólo un hilo puede ejecutar el método synchronized de ese objeto a la vez.
  - Los hilos que necesitan acceder a ese método synchronized permanecerán bloqueados y en espera.
  - Cuando el hilo finaliza la ejecución del método synchronized, los hilos en espera de poder ejecutarlo se desbloquearán. El planificador Java seleccionará a uno de ellos.

```
class ServidorWeb {
private int cuenta;

    public ServidorWeb() {
        cuenta = 0;
    }
    public synchronized void incrementaCuenta() { //método sincronizado
        System.out.println("hilo " + Thread.currentThread().getName()
        + "----- Entra en Servidor");
        //cuenta cada acceso al servidor y muestra el número de accesos
        cuenta++;
        System.out.println(cuenta + " accesos");
    }
}

class Hilo_Terminal extends Thread {

    private ServidorWeb servidor;
    public Hilo_Terminal(ServidorWeb s) {
        this.servidor = s;
    }
    @Override
    public void run() {
        //método que incrementa la cuenta de accesos
        for (int i = 1; i <= 10; i++) { //se simulan 10 accesos
            servidor.incrementaCuenta();
```

```
        }
    }
}
public class Principal {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        ServidorWeb servidor = new ServidorWeb();
        Hilo_Terminal hterminal1 = new Hilo_Terminal(servidor);
        Hilo_Terminal hterminal2 = new Hilo_Terminal(servidor);
        Hilo_Terminal hterminal3 = new Hilo_Terminal(servidor);
        Hilo_Terminal hterminal4 = new Hilo_Terminal(servidor);

    hterminal1.start();
    hterminal2.start();
    hterminal3.start();
    hterminal4.start();
    }

}
```

3.2 Bloques sincronizados

Una versión algo más simple que los métodos sincronizados son los bloques sincronizados. En esta ocasión, no es necesario definir un método sino que en el mismo código, haciendo uso de la palabra reservada “synchronized” seguida del objeto que se está compartiendo entre paréntesis, llevamos a cabo la misma operación.

```
synchronized (object){
    //sentencias críticas
}
```

Ese object será compartido entre varios hilos de ejecución, pero gracias al uso de synchronized garantizamos que solo 1 hilo ejecute a la vez la sección crítica. Es una solución más ágil que codificar un método, aunque impide la reutilización de código.

Java soporta comunicación entre hilos mediante los siguientes métodos:

- **wait()**: Detiene el hilo (pasa a “no ejecutable”), el cual no se reanudará hasta que otro hilo notifique que ha ocurrido lo esperado.
- **wait(long tiempo)**: Como el caso anterior, solo que ahora el hilo también puede reanudarse (pasar a “ejecutable·) si ha concluido el tiempo pasado como parámetro.
- **notify()**: Notifica a uno de los hilos puestos en espera para el mismo objeto que ya puede continuar.

- **notifyAll()**: Notifica a todos los hilos puestos en espera para el mismo objeto que ya pueden continuar. La llamada a estos métodos se realiza dentro de bloques `synchronized`. Un problema clásico que permiten ilustrar la necesidad de sincronizar y comunicar hilos es:

El problema del Productor-Consumidor. Permite modelar situaciones en las que se divide el trabajo entre los hilos. Modela el acceso simultáneo de varios hilos a una estructura de datos u otro recurso, de manera que unos hilos producen y almacenan los datos en el recurso y otros hilos (consumidores) se encargan de eliminar y procesar esos datos.

- Veamos un ejemplo: crearemos 2 tipos de hilos:
  - uno Productor que pondrá algunos datos (por ejemplo, un número entero) en un objeto dado (lo llamaremos DatoCompartido), y otro
  - Consumidor que obtendrá estos datos.

La clase *DatoCompartido* será:

```
public class DatoCompartido {
int dato;
    public int obtener() {
        return dato;
    }
    public void poner(int nuevoDato) {
        dato = nuevoDato;
    }
}
```

Y nuestras clases para los hilos *Productor* y *Consumidor* son:

```
public class Productor extends Thread {
DatoCompartido dato;
    public Productor( DatoCompartido dato){
        this.dato = dato;
    }
    @Override
    public void run({
        for (int i = 0; i < 50; i++)
        {
            dato.poner(i);
            System.out.println("Número producido " + i);
            try
            {
                Thread.sleep(10);
            } catch (Exception e) {}
        }
    }
}

public class Consumidor extends Thread
```

```
{
DatoCompartido dato;
public Consumidor(DatoCompartido dato){
    this.dato = dato;
}
@Override
public void run(){
    for (int i = 0; i < 50; i++){
    int n = dato.obtener();
    System.out.println("Número consumido " + n);
    try
    {
        Thread.sleep(10);
    } catch (Exception e) {}
    }
}
}
```

La aplicación principal creará un objeto de tipo DatoCompartido y un hilo de cada tipo, e iniciará ambos.

```
public static void main(String[] args) {
    DatoCompartido sd = new DatoCompartido();
    Productor p = new Productor(sd);
    Consumidor c = new Consumidor(sd);
    p.start();
    c.start();
}
```

Copia y arranca el ejemplo, verás la salida así:

```
Número consumido 0
Número producido 0
Número consumido 0
Número producido 1
Número consumido 1
Número producido 2
Número producido 3
Número consumido 3
Número producido 4
Número consumido 4
Número consumido 4
...
```

Fíjate, a veces, el productor pone los números demasiado rápido y, a veces, el consumidor también obtiene los números demasiado rápido, de modo que no están coordinados (el consumidor puede leer dos veces el mismo número, o el productor puede poner dos números consecutivos, lo cual no tiene ningún sentido).

Podríamos pensar que, si solo agregamos la palabra clave *synchronized* a los métodos poner y obtener,

resolveríamos el problema:

```
public class DatoCompartido {
int dato;
    public synchronized int obtener() {
        return dato;
    }
    public synchronized void poner(int nuevoDato) {
        dato = nuevoDato;
    }
}
```

Sin embargo, si volvemos a ejecutar el programa, podemos notar que sigue fallando.

De hecho, hay dos problemas que debemos resolver. Pero empecemos por lo más importante:

productor y consumidor tienen que trabajar coordinados, en cuanto el productor pone un número, el consumidor puede conseguirlo, y el productor no podrá producir más números hasta que el consumidor consiga los anteriores.

Para hacer esto, necesitamos agregar algunos cambios a nuestra *clase DatoCompartido*. En primer lugar, necesitamos una *variable lógica* que les diga a los productores y consumidores quién sigue. Dependerá de si hay nuevos datos para consumir (turno para el consumidor) o no (turno para el productor).

```
public class DatoCompartido {
int dato;

boolean disponible = false;

    public synchronized int obtener() {
        disponible = false;
        return dato;
    }
    public synchronized void poner(int nuevoDato) {
        dato = nuevoDato;
        disponible = true;
    }
}
```

Además, debemos asegurarnos de que los métodos poner y obtener se llamarán alternativamente. Para hacer esto, necesitamos usar la variable booleana y los **métodos wait: notify/notifyAll.**

Por lo que, finalmente, la clase DatoCompartido quedaría así:

```
public class DatoCompartido {
int dato;
boolean disponible = false;

public synchronized int obtener() {
    if (!disponible)
```

```
    try {
        wait();
    } catch (Exception e) {}
    disponible = false;
    notify();
    return dato;
}
public synchronized void poner(int nuevoDato) {
    if (disponible)
    try {
        wait();
    } catch (Exception e) {}
    dato = nuevoDato;
    disponible = true;
    notify();
    }
}
```

4 Ejercicios

4.1 Lambda

Vamos a siumlar en Java un programa para sabotear los servidores de la sede de defensa del gobierno de Estados Unidos con solo unas líneas de código malicioso.

Para ello, el programa solicitará al usuario el número de virus a inyectar. Por cada uno de ellos, el programa funcionará del siguiente modo:

- Lanzará un hilo de ejecución (codificado en forma de **expresión lambda**) informando el número de virus (Virus 1, Virus 2, etc…).
- En la misma línea, se mostrará una barra de progreso en la que, en cada segundo, acumularemos de forma aleatoria un 5 o un 10, para de esa forma dotar de más realismo, si cabe.
- Cuando sumemos un 5 al total, imprimiremos una “X” en la barra de progreso y cuando sumemos 10, imprimiremos “XX”, esperando medio segundo entre impresión e impresión para ir viendo en tiempo real como se va completando.
- Cuando lleguemos al 100%, imprimiremos el valor “100%” al lado de la barra de progreso y pasaremos a darle el control al siguiente virus, que se comportará exactamente igual.
- Cuando se hayan cargado todos los virus, el programa imprimirá el mensaje **“HAS SIDO INFECTADO”**. Un ejemplo de ejecución sería el siguiente: si queremos cargar 4 virus en el Pentágono, primero se carga el virus 1, luego el virus 2…

```
Introduzca numero de virus a cargar...
3
```

```
Virus 1:XXXXXXXXXXXXXXXXXXX 100%
Virus 2:XXXXXXXXXXXXXXXXXXX 100%
Virus 3:XXXXXXXXXXXXXXXXXXX 100%

HAS SIDO INFECTADO!
```

4.2 Sincronizar

Sigamos con otro ejemplo sencillo para ilustrar la necesidad de sincronización. Vamos a ver una aplicación que modifica, a través de 2 hilos distintos, el valor de un contador. En primer lugar, creamos un objeto de clase Contador que será compartido entre hilos.

```
public class Contador{
    int valor;
    public Contador(int valor){
        this.valor = valor;
    }
    public void incrementar(){
        valor++;
    }
    public void decrementar(){
        valor--;
    }
    public int getValor(){
        return valor;
    }
}
```

Y el programa principal, el cual creará 2 tipos de hilos: uno que incrementará el contador en un bucle y otro que lo decrementará.

Los hilos para **incrementar** y **decrementar** los podéis desarrollar de dos maneras bien creando sendas clase que hereden de *Thread* o bien con funciones anónimas.

Usando la sentencia `synchronized`, consigue la sincronización de forma que, cada vez que ejecutes el programa éste siempre de 100 como resultado.

4.3 El jefe

Vamos a crear un programa que los Empleados digan buenos dias cuando entre el jefe… Para ello tendremos tres clases:

- La clase Principal
- La clase Empleado

```
String nombre;
Saludo saludo;
boolean esJefe;
```

- La clase saludo extends Thread El método run() comprobará si es el hilo jefe o empleado para saludar…

Con los métodos:

- SaludoEmpleado(String nombre)
- SaludoJefe(String nom)

4.4 Consumidor - Productor

Crearemos 2 tipos de hilos:

- uno **Productor** que pondrá algunos datos (por ejemplo, un número entero) en un objeto dado (lo llamaremos **DatoCompartido**),
- y otro **Consumidor** que obtendrá estos datos.

La clase **DatoCompartido** será:

```
public class DatoCompartido {
    int dato;
    public int obtener() {
        return dato;
    }
    public void poner(int nuevoDato) {
        dato = nuevoDato;
    }
}
```

La ejecución debe de ser algo así:

```
Poner(1)
Obtener(1)
Poner(2)
Obtener(2)
...
```

- Modificar el código para que el productor introduzca los elementos en un ArrayList `datos = new ArrayList<Integer>(limite);` y el consumidor los vaya borrando `datos.remove(...`

La salida debe ser algo parecido a esto:

```
Productor - 0
Consumidor - 0
Productor - 1
Productor - 2
Productor - 3
Productor - 4
Productor - 5
Productor - 6
Consumidor - 2
Consumidor - 6
Consumidor - 5
Consumidor - 4
Productor - 7
Productor - 8
Productor - 9
Consumidor - 3
Consumidor - 9
Consumidor - 8
Consumidor - 7
Consumidor - 1
Fin
```

(d) Proveedor-Consumidor

4.5 LavaPlatos

Crea un proyecto llamado Lavavajillas. Vamos a simular el proceso de lavado de platos en casa, cuando alguien lava los platos y otro más los seca.

Crea las siguientes clases:

- Una clase Plato con solo un atributo entero: el número de plato (para identificar los diferentes

platos).

- Una clase PilaPlatos que almacenará hasta 5 platos. Tendrá un método lavar que pondrá un plato en la pila (si hay espacio disponible) y un método secar que cogerá un plato de la pila (si hay alguno). Tal vez necesites un parámetro Plato en el método lavar para añadir un plato a la pila.
- Un hilo Friega que recibirá un número N como parámetro y un objeto de tipo PilaPlatos. En su método run pondrá (lavará) N platos en la pila, con una pausa de 50ms entre cada plato.
- Un hilo Seca que recibirá un número N como parámetro y un objeto de tipo PilaPlatos. En su método run, sacará (secará) N platos de la pila, con una pausa de 100 ms entre cada plato.
- La clase principal creará el objeto PilaPlatos y un hilo de cada tipo (Friega y Seca). Tendrán que lavar/secar 20 platos coordinadamente, por lo que el resultado podría ser algo así:

<terminated> LavavajillasStack [Java Application] /Library/Java/JavaV

```
Plato lavado #0, total en pila: 1
Plato secado #0, total en pila: 0
Plato lavado #1, total en pila: 1
Plato secado #1, total en pila: 0
Plato lavado #2, total en pila: 1
Plato lavado #3, total en pila: 2
Plato secado #3, total en pila: 1
Plato lavado #4, total en pila: 2
Plato lavado #5, total en pila: 3
Plato secado #5, total en pila: 2
Plato lavado #6, total en pila: 3
Plato lavado #7, total en pila: 4
Plato secado #7, total en pila: 3
Plato secado #6, total en pila: 2
Plato secado #4, total en pila: 1
Plato secado #2, total en pila: 0
```

(e) Tiempo de ejecución