



# 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

<b>1</b>	<b>EL MÉTODO JOIN</b>	<b>3</b>
1.1	Objetivo del método . . . . .	3
<b>2</b>	<b>EL MÉTODO INTERRUPT</b>	<b>4</b>
<b>3</b>	<b>SINCRONIZACIÓN ENTRE HILOS</b>	<b>6</b>
3.1	Métodos sincronizados . . . . .	6
3.2	Bloques sincronizados . . . . .	9
<b>4</b>	<b>Ejercicio</b>	<b>9</b>
4.1	Lambda . . . . .	9
4.2	Sincronizar . . . . .	10
<b>5</b>	<b>El jefe</b>	<b>11</b>

## 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

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

Otra opción sería 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 tiene que ejecutar 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 startTime= 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-startTime;

        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 crítica, 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.

## 4 Ejercicio

### 4.1 Lambda

Vamos a simular 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:XXXXXXXXXXXXXXXXXXXXX 100%
Virus 2:XXXXXXXXXXXXXXXXXXXXX 100%
Virus 3:XXXXXXXXXXXXXXXXXXXXX 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.

## 5 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

““java 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)