

1.- COMUNICACIÓN Y SINCRONIZACIÓN DE HILOS

Analizar el siguiente ejercicio donde dos hilos comparten un objeto y a través de él pueden comunicarse.

Clase contador

```
package Sincroniza1;

class Contador {
    private int c=0;
    Contador(int c) {
        this.c=c;
    }
    public void incrementa() { c=c+1; }
    public void decrementa() { c=c-1; }
    public int getValor() {return c;}
}
```

Clase HiloA

```
package Sincroniza1;

class HiloA extends Thread {
    private Contador contador;
    public HiloA(String n, Contador c) {
        setName(n);
        contador = c;
    }
    public void run() {
        for (int j=0; j<300; j++) {
            contador.incrementa();
            try { sleep(100); }
            catch (InterruptedException e) {
                System.out.println("Se ha producido un error en el hilo A");
            }
        }
        System.out.println(getName()+"Hilo A. Contador vale :"+ contador.getValor());
    }
}
```

Clase HiloB

```
package Sincroniza1;

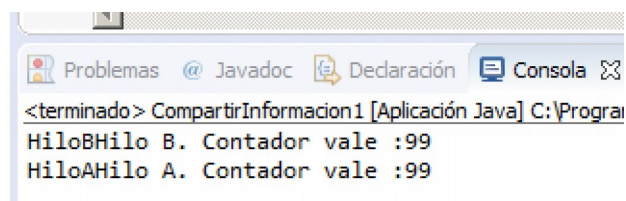
class HiloB extends Thread {
    private Contador contador;
    public HiloB (String n, Contador c) {
        setName(n);
        contador = c;
    }
    public void run() {
        for (int j=0; j<300; j++) {
            contador.decrementa();
            try { sleep(100); }
            catch (InterruptedException e) {
                System.out.println("Se ha producido un error en el hilo A");
            }
        }
        System.out.println(getName()+"Hilo B. Contador vale :"+ contador.getValor());
    }
}
```

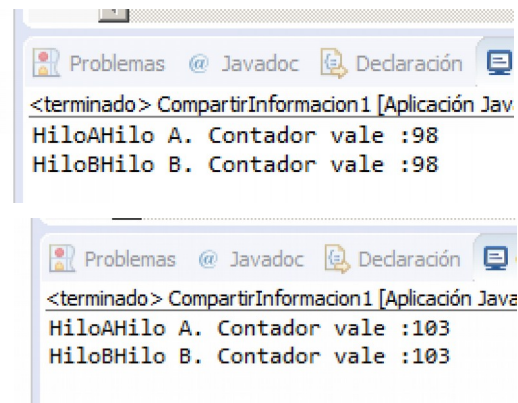
Clase CompartirInformacion

```
package Sincroniza1;

public class CompartirInformacion1 {
    public static void main(String[] args) {
        Contador cont = new Contador(100);
        HiloA a = new HiloA("HiloA",cont);
        HiloB b = new HiloB("HiloB",cont);
        a.start();
        b.start();
    }
}
```

Ejecutamos varias veces el código





Los valores varían de una ejecución a otra ya que no controlamos que la ejecución sea atómica (transaccional). Es decir, hay que asegurarse que mientras hacemos la suma nadie hace la resta, y viceversa.

Para resolver este problema utilizamos “**synchronized**” en la parte del código que queremos que se ejecute de forma atómica (*ningún otro proceso puede tomar conocimiento de los cambios realizados hasta que se complete todas las instrucciones del conjunto; es decir, el conjunto de instrucciones se ve como una sola instrucción*). Es decir Java marcará esa parte de código como una región crítica.

Para resolver este problema utilizamos “**synchronized**” en la parte del código que queremos que se ejecute de forma atómica. Es decir Java marcará esa parte de código como una región crítica.

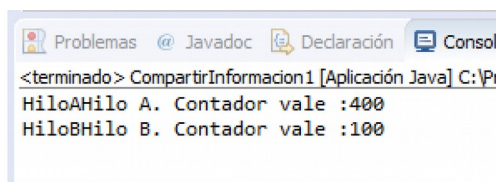
Modificar el código de la clase HiloA e HiloB

```
package Sincroniza1;

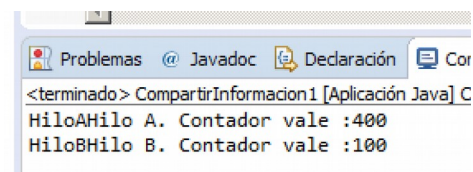
class HiloA extends Thread {
    private Contador contador;
    public HiloA (String n, Contador c) {
        setName(n);
        contador = c;
    }
    public void run() {
        synchronized (contador) {
            for (int j=0; j<300; j++) {
                contador.incrementa();
                try { sleep(10); }
                catch (InterruptedException e) {
                    System.out.println("Se ha producido un error en el hilo A");
                }
            }
            System.out.println(getName()+"Hilo A. Contador vale :"+ contador.getValor());
        }
    }
}
```

```
package Sincroniza1;

class HiloB extends Thread {
    private Contador contador;
    public HiloB (String n, Contador c) {
        setName(n);
        contador = c;
    }
    public void run() {
        synchronized (contador) {
            for (int j=0; j<300; j++) {
                contador.decrementa();
                try { sleep(10); }
                catch (InterruptedException e) {
                    System.out.println("Se ha producido un error en el hilo A");
                }
            }
            System.out.println(getName()+"Hilo B. Contador vale :"+ contador.getValor());
        }
    }
}
```



```
<terminado> CompartirInformacion1 [Aplicación Java] C:\P
HiloAHilo A. Contador vale :400
HiloBHilo B. Contador vale :100
```



```
<terminado> CompartirInformacion1 [Aplicación Java] C
HiloAHilo A. Contador vale :400
HiloBHilo B. Contador vale :100
```

Estamos bloqueando el objeto “contador” de forma que el hilo que intenta acceder a un objeto bloqueado queda suspendido y queda a la espera a que se quede libre.

El objeto quedará libre cuando se termine la ejecución, se ejecute un return o bien se dispare una excepción.

1.1.-Sincronización de bloques y métodos

Se debe evitar la sincronización de bloques de código y sustituirlo por la sincronización de métodos. La sincronización disminuye el rendimiento de una aplicación, por lo que debemos utilizarlo lo justo.

```
synchronized public void metodo() {
    // instrucciones atómicas ...
}
```

MÉTODOS SINCRONIZADOS

MÉTODOS SINCRONIZADOS

Se debe de evitar la sincronización de bloques de códigos y sustituirlas siempre que sea posible por la sincronización de métodos, **exclusión mutua** de los procesos respecto a la variable compartida. Imaginemos la situación que dos personas comparten una cuenta y pueden sacar dinero de ella en cualquier momento; antes de retirar dinero se comprueba siempre si existe saldo. La cuenta tiene 50€, una de las personas quiere retirar 40€ y la otra 30€. La primera llega al cajero , revisa el saldo, comprueba que hay dinero y se prepara para retirar el dinero, pero antes de retirarlo llega la otra persona a otro cajero , comprueba el saldo que todavía muestra 50€ y también se dispone a retirar el dinero. Las dos personas retiran el dinero, pero entonces el saldo actual será ahora de -20€.

Para sincronizar un método , simplemente añadimos la palabra clave `synchronized` a su declaración. Por ejemplo, la clase Contador con métodos sincronizados sería así:

```
class ContadorSincronizado {  
    private int c = 0;  
    Contador(int c) {  
        this.c = c;  
    }  
  
    public synchronized void incrementa() {  
        c = c + 1;  
    }  
  
    public synchronized void decrementa() {  
        c = c - 1;  
    }  
  
    public synchronized int getValor() {  
        return c;  
    }  
}
```

El uso de métodos sincronizados implica que no es posible invocar dos métodos sincronizados del mismo objeto a la vez. Cuando un hilo está ejecutando un método sincronizado de un objeto, los demás hilos que invoquen a métodos sincronizados para el mismo objeto se bloquean hasta que el primer hilo termine con la ejecución del método.

Ejercicio CuentaBancaria

Realizar el ejercicio de la cuenta bancaria sin usar métodos sincronizados y posteriormente con métodos sincronizados.

Clase cuenta :

Definir un atributo “saldo” y tres métodos.

- **Método getSaldo:** devuelve el saldo de la cuenta.
- **Método restar:** resta al saldo una cantidad
- **Método RetirarDinero:** realiza las comprobaciones para hacer la retirada del dinero, es decir que el saldo actual sea \geq que la cantidad que se quiere retirar.

El constructor inicia el saldo actual. También se añade un **sleep()** intencionadamente para probar que un hilo se duerma y mientras tanto , el otro haga las operaciones.

Clase SacarDinero (extiende a Thread)

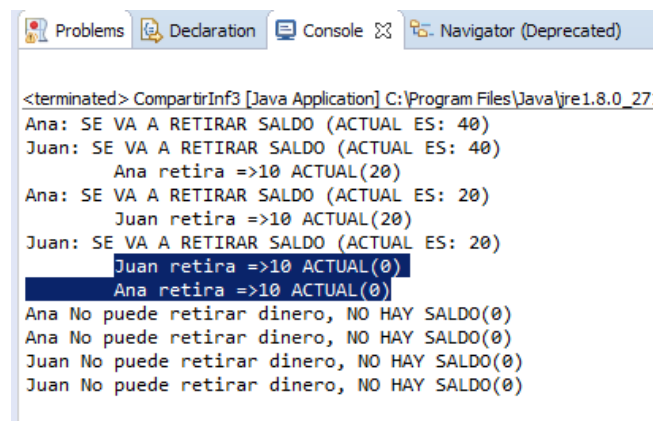
Esta clase usa la clase Cuenta para retirar dinero. El constructor recibe una cadena, para dar nombre al hilo; y la cuenta que será compartida por varias personas.

- **Método run()** se realiza un bucle donde se invoca al método RetirarDinero() de la clase Cuenta varias veces con la cantidad a retirar, en este caso siempre es 10, y el nombre del hilo

Clase CompartirCuenta

Esta clase contiene solo el método main(), donde primero se define un objeto de la clase Cuenta y se le asigna un saldo inicial de 40. A continuación se crean dos objetos de la clase SacarDinero, y se inician los hilos.

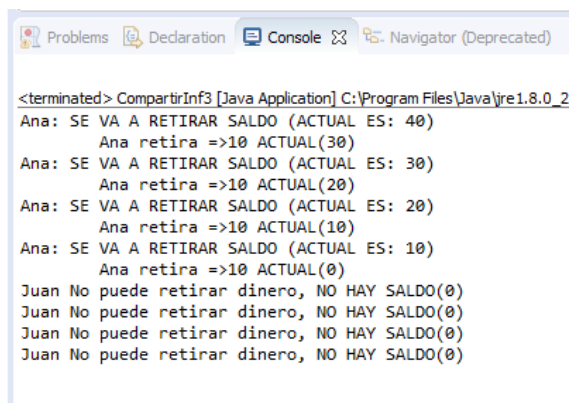
Comprobar que se puede retirar saldo cuando la cuenta está a 0.



```
<terminated> CompartirInf3 [Java Application] C:\Program Files\Java\jre1.8.0_27:
Ana: SE VA A RETIRAR SALDO (ACTUAL ES: 40)
Juan: SE VA A RETIRAR SALDO (ACTUAL ES: 40)
    Ana retira =>10 ACTUAL(20)
Ana: SE VA A RETIRAR SALDO (ACTUAL ES: 20)
    Juan retira =>10 ACTUAL(20)
Juan: SE VA A RETIRAR SALDO (ACTUAL ES: 20)
    Juan retira =>10 ACTUAL(0)
    Ana retira =>10 ACTUAL(0)
Ana No puede retirar dinero, NO HAY SALDO(0)
Ana No puede retirar dinero, NO HAY SALDO(0)
Juan No puede retirar dinero, NO HAY SALDO(0)
Juan No puede retirar dinero, NO HAY SALDO(0)
```

Para evitar esta situación la operación de retirar dinero , método RetirarDinero() de la clase Cuenta , debería ser atómica e indivisible, es decir si una persona está retirando dinero, la otra debería ser incapaz de retirarlo hasta que la primera haya realizado la operación. Para ello debemos declarar el método como synchronized. Sincronizar métodos permite prevenir inconsistencias cuando un objeto es accesible desde distintos hilos: si un objeto es visible para más de un hilo, todas las lecturas o escrituras de las variables de ese objeto se realizan a través de métodos sincronizados.

Cuando un hilo invoca un método synchronized, trata de tomar el bloqueo del objeto a que pertenezca. Si está libre , lo toma y se ejecuta. Si el bloqueo está tomado por otro hilo se suspende el que invoca hasta que aquel finalice y libere el bloqueo.



```
<terminated> CompartirInf3 [Java Application] C:\Program Files\Java\jre1.8.0_2
Ana: SE VA A RETIRAR SALDO (ACTUAL ES: 40)
    Ana retira =>10 ACTUAL(30)
Ana: SE VA A RETIRAR SALDO (ACTUAL ES: 30)
    Ana retira =>10 ACTUAL(20)
Ana: SE VA A RETIRAR SALDO (ACTUAL ES: 20)
    Ana retira =>10 ACTUAL(10)
Ana: SE VA A RETIRAR SALDO (ACTUAL ES: 10)
    Ana retira =>10 ACTUAL(0)
Juan No puede retirar dinero, NO HAY SALDO(0)
Juan No puede retirar dinero, NO HAY SALDO(0)
Juan No puede retirar dinero, NO HAY SALDO(0)
Juan No puede retirar dinero, NO HAY SALDO(0)
```