

Programación multihilo

Programación Servicios y procesos tema 02

Hilo

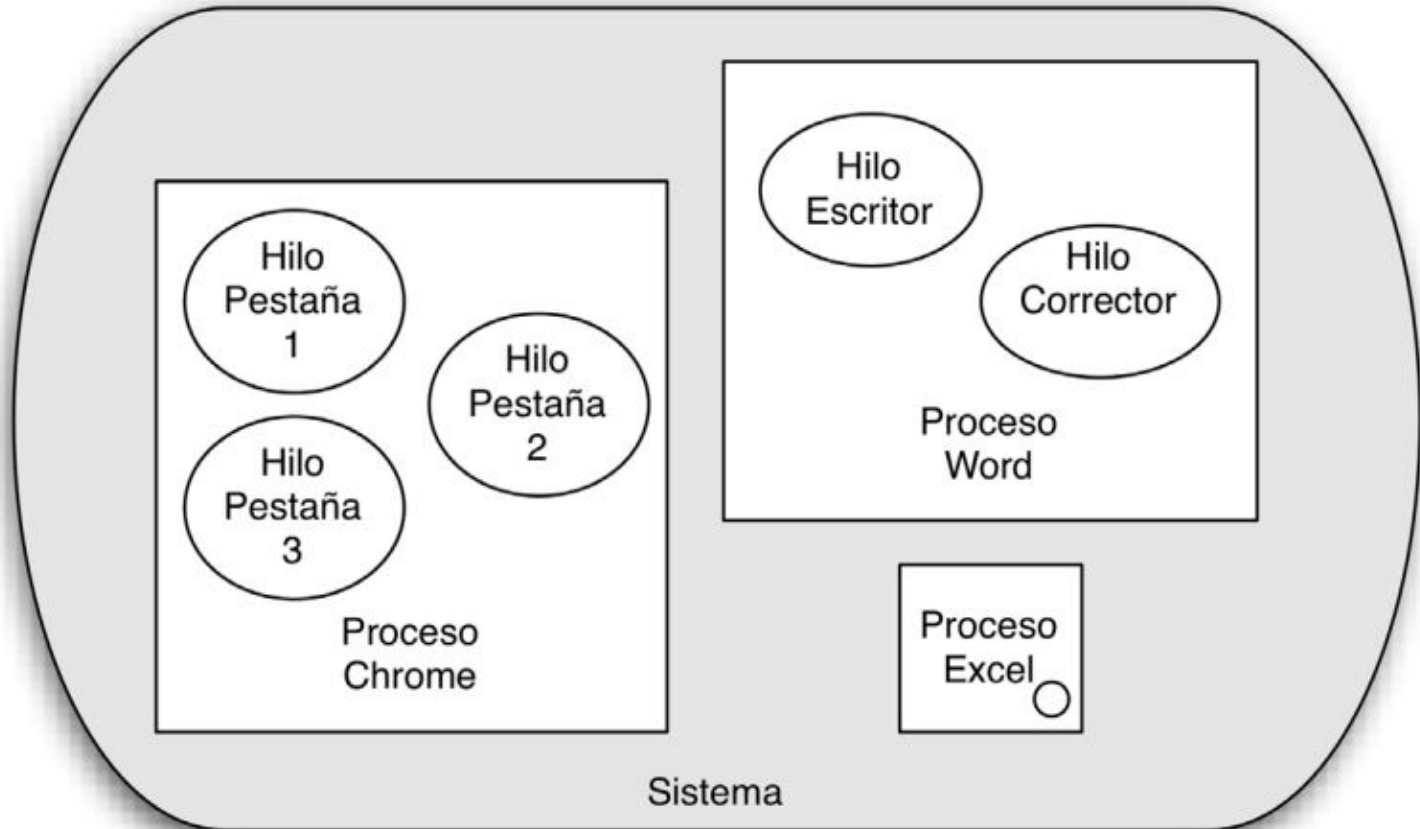
▶ Hilo (thread):

- Unidad básica de utilización de la CPU, y más concretamente de un *core* del procesador.
- Secuencia de código que está en ejecución, pero dentro del contexto de un proceso.

▶ Diferencia con procesos:

- Los hilos se ejecutan dentro del contexto de un proceso. Dependen de un proceso para ejecutarse.
- Los procesos son independientes y tienen espacios de memoria diferentes.
- Dentro de un mismo proceso pueden coexistir varios hilos ejecutándose que compartirán la memoria de dicho proceso.

Hilo versus Proceso

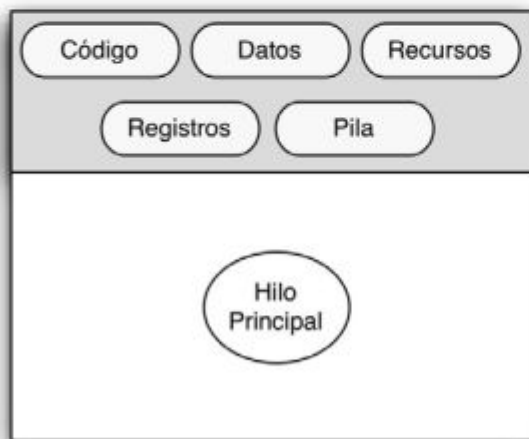


Multitarea

- ▶ Multitarea: ejecución simultanea de varios hilos:
 - Capacidad de respuesta. Los hilos permiten a los procesos continuar atendiendo peticiones del usuario aunque alguna de las tareas (hilo) que esté realizando el programa sea muy larga.
 - Compartición de recursos. Por defecto, los *threads* comparten la memoria y todos los recursos del proceso al que pertenecen.
 - La creación de nuevos hilos no supone ninguna reserva adicional de memoria por parte del sistema operativo.
 - Paralelismo real. La utilización de *threads* permite aprovechar la existencia de más de un núcleo en el sistema en arquitecturas *multicore*.

Recursos compartidos por hilos

- ▶ Los procesos mantienen su propio espacio de direcciones y recursos de ejecución mientras que los hilos dependen del proceso.
 - Comparten con otros hilos la sección de código, datos y otros recursos.
 - Cada hilo tiene su propio contador de programa, conjunto de registros de la CPU y pila para indicar por dónde se está ejecutando.



Proceso con un único thread



Proceso con varios threads

Estados de un hilo

- ▶ Los hilos pueden cambiar de estado a lo largo de su ejecución.
- ▶ Se definen los siguientes estados:
 - Nuevo: el hilo está preparado para su ejecución pero todavía no se ha realizado la llamada correspondiente en la ejecución del código del programa.
 - Listo: el proceso no se encuentra en ejecución aunque está preparado para hacerlo. El sistema operativo no le ha asignado todavía un procesador para ejecutarse.
 - *Runnable*: el hilo está preparado para ejecutarse y puede estar ejecutándose.
 - Bloqueado: el hilo está bloqueado por diversos motivos esperando que el suceso suceda para volver al estado *Runnable*.
 - Terminado: el hilo ha finalizado su ejecución.

La clase Thread

- Para añadir la funcionalidad de hilo a una clase, debe heredar de la clase Thread.
- Debe sobrescribir el método run(). También cuenta con los métodos start y stop.
- Lo probamos con un proyecto con dos clases: Hilo1 y UsaHilo1
- En Hilo1, definimos la clase que hereda de Thread con el método run()
- En UsaHilo1, creamos objetos basados en la clase Hilo1

```
public class Hilo1 extends Thread{  
    public Hilo1(String nombre) {  
        super(nombre);  
        System.out.println("CREANDO " + getName());  
    }  
    public void run() {  
        System.out.println("Ejecutando HILO:" + getName());  
    }  
}
```

```
public class UsaHilo1 {  
    public static void main(String[] args){  
        for(int n=0; n<5; n++) {  
            Hilo1 h1 = new Hilo1( nombre: "Hilo" + n);  
            h1.start();  
        }  
    }  
}
```

Métodos de la clase Thread

MÉTODOS	MISIÓN
start()	Hace que el hilo comience la ejecución; la máquina virtual de Java llama al método run() de este hilo.
boolean isAlive()	Comprueba si el hilo está vivo
sleep(long mills)	Hace que el hilo actualmente en ejecución pase a dormir temporalmente durante el número de milisegundos especificado. Puede lanzar la excepción <i>InterruptedException</i> .
run()	Constituye el cuerpo del hilo. Es llamado por el método start() después de que el hilo apropiado del sistema se haya inicializado. Si el método run() devuelve el control, el hilo se detiene. Es el único método de la interfaz Runnable .
String toString()	Devuelve una representación en formato cadena de este hilo, incluyendo el nombre del hilo, la prioridad, y el grupo de hilos. Ejemplo: Thread[HILO1,2,main]
long getId()	Devuelve el identificador del hilo.
void yield()	Hace que el hilo actual de ejecución pare temporalmente y permita que otros hilos se ejecuten.
String getName()	Devuelve el nombre del hilo.
setName(String name)	Cambia el nombre de este hilo, asignándole el especificado como argumento.

Ejemplo utilización sleep(milisegundos)

Modificamos el ejemplo anterior para que se detenga la ejecución del hilo que se llama Hilo2. Si no pasa nada raro (con hilos es posible), veremos en la consola que el hilo2 es el último en terminar su ejecución.

```
public class Hilo1 extends Thread{
    public Hilo1(String nombre) {
        super(nombre);
        System.out.println("CREANDO " + getName());
    }
    public void run() {
        if(getName().equals("Hilo2")){
            try {
                sleep( millis: 2000);
            }catch (InterruptedException ie){
                ie.printStackTrace();
            }
        }
        System.out.println("Ejecutando HILO:" + getName() + " con Id " + getId());
    }
}
```

Más métodos de la clase Thread

MÉTODOS	MISIÓN
int getPriority()	Devuelve la prioridad del hilo.
setPriority(int p)	Cambia la prioridad del hilo al valor entero p.
void interrupt()	Interrumpe la ejecución del hilo
boolean interrupted()	Comprueba si el hilo actual ha sido interrumpido.
Thread currentThread()	Devuelve una referencia al objeto hilo que se está ejecutando actualmente.
boolean isDaemon()	Comprueba si el hilo es un hilo Daemon. Los hilos daemon o demonio son hilos con prioridad baja que normalmente se ejecutan en segundo plano. Un ejemplo de hilo demonio que está ejecutándose continuamente es el recolector de basura (<i>garbage collector</i>).
setDaemon(boolean on)	Establece este hilo como hilo Daemon, asignando el valor <i>true</i> , o como hilo de usuario, pasando el valor <i>false</i> .
void stop()	Detiene el hilo. Este método está en desuso.
Thread currentThread()	Devuelve una referencia al objeto hilo actualmente en ejecución.
int activeCount()	Este método devuelve el número de hilos activos en el grupo de hilos del hilo actual.
Thread.State getState()	Devuelve el estado del hilo: NEW, RUNNABLE, BLOCKED, WAITING, TIMED_WAITING, TERMINATED

Comprobamos getState() en el constructor y en run()

```
public class Hilo1 extends Thread{
    public Hilo1(String nombre) {
        super(nombre);
        System.out.println("CREANDO " + getName());
        System.out.println("constructor: " + getState());
    }
    public void run() {
        if(getName().equals("Hilo2")){
            try {
                sleep( millis: 2000);|
            }catch (InterruptedException ie){
                ie.printStackTrace();
            }
        }
        System.out.println("Ejecutando HILO:" + getName() + " con Id " + getId());
        System.out.println(getState());
    }
}
```

Ejemplos de algunos de los métodos anteriores (Hilos02)

```
public class HiloEjemplo2 extends Thread {

    public void run() {
        System.out.println(
            "\nDentro del Hilo : " + Thread.currentThread().getName() +
            "\n\tPrioridad : " + Thread.currentThread().getPriority() +
            "\n\tID : " + Thread.currentThread().getId() +
            "\n\tHilos activos: " + Thread.currentThread().activeCount());
    }
    //
    public static void main(String[] args) {

        Thread.currentThread().setName("Principal");
        System.out.println(Thread.currentThread().getName());
        System.out.println(Thread.currentThread().toString());

        HiloEjemplo2 h = null;

        for (int i = 0; i < 3; i++) {
            h = new HiloEjemplo2(); //crear hilo
            h.setName("HILO"+i); //damos nombre al hilo
            h.setPriority(i+1); //damos prioridad
            h.start(); //iniciar hilo

            System.out.println(
                "\nInformacion del " + h.getName() + ": " + h.toString());
        }
        System.out.println("3 HILOS CREADOS...");
        System.out.println("Hilos activos: " + Thread.activeCount());
    } //
} // HiloEjemplo2
```

Grupos de hilos, ThreadGroup (Ejemplo Hilos03)

La clase ThreadGroup se utiliza para manejar grupos de hilos en las aplicaciones Java. Puede crear nuevos hilos (Thread) dentro de un grupo basándose en un objeto Thread:

1. Creamos un objeto ThreadGroup
2. Creamos un objeto Thread (o de una clase que herede de Thread)
3. Creamos nuevos hilos usando el grupo y el hilo creados antes.

```
1 ThreadGroup grupo = new ThreadGroup( name: "Grupo de hilos");  
2 HiloEjemplo2Grupos h = new HiloEjemplo2Grupos();  
3 Thread h1 = new Thread(grupo, h, name: "Hilo 1");  
  Thread h2 = new Thread(grupo, h, name: "Hilo 2");  
  Thread h3 = new Thread(grupo, h, name: "Hilo 3");
```


Interfaz Runnable

- Tiene las mismas funcionalidades que Thread
- Se usa para las clases que necesitan heredar de otras clases y contar con las características de un hilo.

```
class NombreHilo implements Runnable {  
    //propiedades, constructores y métodos de la clase  
    public void run() {  
        //acciones que lleva a cabo el hilo  
    }  
}
```

Gestión de prioridades

La prioridad es un número entre 1 y 10. El planificador elige el hilo a ejecutar en función de la prioridad.

El hilo de mayor prioridad sigue funcionando hasta que:

- Cede el control llamando al método `yield()` para que otros hilos de igual prioridad se ejecuten.
- Deja de ser ejecutable (ya sea por muerte o por entrar en el estado de bloqueo).
- Un hilo de mayor prioridad se convierte en ejecutable (porque se encontraba dormido o su operación de E/S ha finalizado o alguien lo desbloquea llamando a los métodos `notifyAll()` / `notify()`).

El uso del método `yield()` devuelve automáticamente el control al planificador, el hilo pasa a un estado de listo para ejecutar. Sin éste método el mecanismo de multihilos sigue funcionando aunque algo más lentamente.

Ejemplos de prioridad en hilos

```
class HiloPrioridad1 extends Thread {  
    private int c = 0;  
    private boolean stopHilo= false;  
  
    public HiloPrioridad1(String nombre) { super(nombre); }  
    public int getContador() { return c; }  
    public void pararHilo() { stopHilo = true; }  
    public void run() {  
        while (!stopHilo) {  
            try {  
                Thread.sleep( millis: 2);  
            } catch (Exception e) { }  
            c++;  
        }  
        System.out.println("Fin hilo  "+this.getName());  
    }  
}
```

En la clase EjemploHiloPrioridad1 se definen 3 objetos de la clase HiloPrioridad1, a cada uno se le asigna una prioridad. El contador del hilo al que se le ha asignado mayor prioridad (probablemente) contará más deprisa que el de menos prioridad. Al finalizar cada hilo se muestran los valores del contador invocando a cada método getContador() del hilo

```
public class EjemploHiloPrioridad1 {  
    public static void main(String args[]) {  
        HiloPrioridad1 h1 = new HiloPrioridad1( nombre: "Hilo1");  
        HiloPrioridad1 h2 = new HiloPrioridad1( nombre: "Hilo2");  
        HiloPrioridad1 h3 = new HiloPrioridad1( nombre: "Hilo3");  
  
        h1.setPriority(Thread.NORM_PRIORITY);  
        h2.setPriority(Thread.MAX_PRIORITY);  
        h3.setPriority(Thread.MIN_PRIORITY);  
  
        h1.start();  
        h2.start();  
        h3.start();  
  
        try {  
            Thread.sleep( millis: 10000);  
        } catch (Exception e) { }  
  
        h1.pararHilo() ;  
        h2.pararHilo() ;  
        h3.pararHilo() ;  
  
        System.out.println("h2 (Prioridad Maxima): " + h2.getContador());  
        System.out.println("h1 (Prioridad Normal): " + h1.getContador());  
        System.out.println("h3 (Prioridad Minima): " + h3.getContador());  
    }  
}  
  
} // EjemploHiloPrioridad1
```

Comunicación y sincronización de hilos

A menudo los hilos necesitan comunicarse unos con otros, la forma de comunicarse consiste usualmente en compartir un objeto.

En el ejemplo, dos hilos comparten un objeto de la clase Contador. Esta clase define un atributo contador y tres métodos: incrementa, decrementa y devuelve el valor. El constructor asigna un valor inicial al contador.

Hacemos dos clases que extienden Thread. En la clase HiloA se usa el método incrementa el contador. En la clase HiloB se decrementa. Se añade un sleep() intencionadamente para probar que un hilo se duerma y mientras el otro haga otra operación con el contador, así la CPU no realiza de una sola vez todo un hilo y después otro y podemos observar mejor el efecto.

Por último, solo queda crear una clase Main para crear un objeto de la clase HiloA y otro de la Hilo B e iniciarlos.

Al probarlo sin el método sleep() da la sensación de que la salida es la esperada, pero no siempre nos va a dar ese resultado.

Ejemplo de comunicación y sincronización

```
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; }  
} // CONTADOR
```

```
public class CompartirInf1 {  
    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();  
    }  
}
```

```
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(getName() + " contador vale " + contador.getValor());  
        }  
        System.out.println(getName() + " contador vale " + contador.getValor());  
    }  
} // FIN HILOA
```

En algunas ocasiones el resultado no es el esperado. El resultado debería ser 100, pero no siempre lo es.

Comunicación entre hilos

- ▶ Los *threads* se comunican principalmente mediante el intercambio de información a través de variables y objetos en memoria.
 - Los *threads* pertenecen al mismo proceso, y pueden acceder a toda la memoria asignada a dicho proceso utilizando las variables y objetos del mismo para compartir información, siendo este el método de comunicación más eficiente.
- ▶ Cuando varios hilos manipulan a la vez objetos compartidos, pueden ocurrir diferentes problemas:
 - Condición de carrera
 - Inconsistencia de memoria
 - Inanición
 - Interbloqueo
 - Bloqueo activo

Condiciones de carrera

- Si el resultado de la ejecución de un programa depende del orden concreto en que se realicen los accesos a memoria.

- ▶ Ej: sumar o restar 1 en ensamblador

registroX = cuenta

registroX = registroX (operación: suma o resta) 1

cuenta = registroX

- ▶ Supongamos *cuenta* vale 10, y dos hilos *sumador* y *restador* ejecutándose a la vez sobre *cuenta*. Puede suceder

T0: *sumador* registro1 = cuenta {registro1 = 10}

T1: *sumador* registro1 = registro1 + 1 {registro1 = 11}

T2: *restador* registro2 = cuenta {registro2 = 10}

T3: *restador* registro2 = registro2 - 1 {registro2 = 9}


T4: *sumador* cuenta = registro 1 {cuenta = 11}

T5: *restador* cuenta = registro2 {cuenta = 9}

- ▶ El resultado final *cuenta* = 9 es incorrecto. Condición de carrera

Bloques sincronizados

Para evitar problemas, las sumas y restas se deben hacer de forma atómica. Hasta que no termine una operación, no puede empezar la siguiente. Esto se puede lograr añadiendo la palabra `synchronized` a la parte de código que queramos que se ejecute de forma atómica. Así queda ahora el método `run` del ejemplo anterior.



```
public void run() {  
    synchronized (contador) {  
        for (int i = 0; i < 300; i++) {  
            contador.incrementa();  
        }  
        System.out.println(getName() + " contador vale "  
            + contador.getValor());  
    }  
}
```

El bloque `synchronized` o región crítica lleva entre paréntesis la referencia al objeto compartido `Contador`. Cada vez que un hilo intenta acceder a un bloque sincronizado le pregunta a ese objeto si no hay algún otro hilo que ya le tenga bloqueado. De esta manera el resultado de la ejecución siempre será el mismo.

Métodos sincronizados

Se debe evitar la sincronización de bloques de código y sustituirla siempre que sea posible por la sincronización de métodos, exclusión mutua de los procesos respecto a la variable compartida. El ejemplo del contador anterior quedaría:

```
class ContadorSincronizado{
    private int e=0;
    public synchronized void incrementa() {
        e++;
    }
    public synchronized void decrementa() {
        e--;
    }
    public synchronized int valor() {
        return e;
    }
}
```

El uso de métodos sincronizados implica que no es posible invocar dos métodos sincronizados del mismo objeto a la vez.

Ejemplo de ingresar / sacar dinero

Compartimos una cuenta sin usar métodos sincronizados. Se define la clase Cuenta con un atributo saldo y tres métodos, uno devuelve el valor del saldo, otro, resta al saldo una cantidad y el tercero realiza las comprobaciones para hacer la retirada de 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 el otro haga las operaciones. Tenemos el ejemplo en la clase `CompartirInf3` del proyecto `Hilos06 Sincronizados`.

Al ejecutarlo, comprobamos que hay casos en los que permite sacar dinero con saldo 0

Ejemplo ingresar/sacar dinero "sincronizado"

Para evitar esta situación la operación de retirar dinero, método RetirarDineroO 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 declaramos el método como synchronized.



```
synchronized void RetirarDinero(int cant, String nom) {  
    if (getSaldo() >= cant) {  
        System.out.println(nom+": SE VA A RETIRAR SALDO (ACTUAL ES: "+getSaldo()+ ")" );  
        try {  
            Thread.sleep( millis: 500);  
        } catch (InterruptedException ex) { }  
  
        restar(cant);  
  
        System.out.println("\t"+ nom+ " retira =>"+cant + " ACTUAL("+getSaldo()+)" );  
    } else {  
        System.out.println(nom+ " No puede retirar dinero, NO HAY SALDO("+getSaldo()+)" );  
    }  
    if (getSaldo() < 0) {  
        System.out.println("SALDO NEGATIVO => "+getSaldo());  
    }  
}  
}  
//retirar
```

Problemas de sincronización

- ▶ INCONSISTENCIA DE MEMORIA

Se produce cuando diferentes hilos tienen una visión diferente de lo que debería ser el mismo dato.

- ▶ INANICIÓN

- Cuando un proceso nunca llega a tomar el control de un recurso debido a que el resto siempre toman el control antes que él por diferentes motivos.

- ▶ INTERBLOQUEO

- Se produce cuando dos o más procesos o hilos están esperando indefinidamente por un evento que solo puede generar un proceso o hilo bloqueado

- ▶ BLOQUE ACTIVO

- Es similar a un interbloqueo, excepto que el estado de los dos procesos envueltos en el bloqueo activo cambia constantemente con respecto al otro.

Bloqueo de hilos

Una clase que define un método que recibe un String y lo muestra en pantalla:

```
class ObjetoCompartido1 {  
    public void PintaCadena (String s) { System.out.print(s); }  
} // ObjetoCompartido1
```

Para usarla definimos un método main en el que se crea un objeto de esa clase que además será compartido por dos hilos del tipo HiloCadena. Usando Synchronized, se pretende mostrar de forma alternativa los String que inicializa cada hilo y que la salida generada al ejecutar la función main sea la siguiente: "A B A B A B".

```
public class BloqueoHilosInicial {  
    public static void main(String[] args) {  
        ObjetoCompartido1 com = new ObjetoCompartido1();  
        HiloCadena1 a = new HiloCadena1 (com, s: " A ");  
        HiloCadena1 b = new HiloCadena1 (com, s: " B ");  
        a.start();  
        b.start();  
    }  
} //BloqueoHilosInicial
```

```
class HiloCadena1 extends Thread {  
    private ObjetoCompartido1 objeto;  
    String cad;  
    public HiloCadena1 (ObjetoCompartido1 c, String s) {  
        this.objeto = c;  
        this.cad=s;  
    }  
    public void run() {  
        synchronized (objeto) {  
            for (int j = 0; j < 10; j++)  
                objeto.PintaCadena(cad);  
        } //fin bloque synchronized  
  
        System.out.print("\n"+cad + " finalizado");  
    } //run  
} //HiloCadena1
```

Bloqueo de hilos (2)

Pero al ejecutarlo, la salida no es la esperada. La sincronización evita que dos llamadas a métodos o bloques sincronizados del mismo objeto se mezclen; pero no garantiza el orden de las llamadas; y en este caso nos interesa que las llamadas al método `PintaCadena` se realicen de forma alternativa. Se necesita por tanto mantener una cierta coordinación entre los dos hilos, para ello se usan los métodos `wait()`, `notify()` y `notifyAll()`:

- `Objeto.wait()`: un hilo que llama al método `wait()` de un cierto objeto queda suspendido hasta que otro hilo llame al método `notify()` o `notifyAll()` del mismo objeto.
- `Objeto.notify()`: despierta sólo a uno de los hilos que realizó una llamada a `wait()` sobre el mismo objeto notificándole de que ha habido un cambio de estado sobre el objeto. Si varios hilos están esperando el objeto, solo uno de ellos es elegido para ser despertado, la elección es arbitraria.
- `Objeto.notifyAll()`: despierta todos los hilos que están esperando el objeto.

Bloqueo de hilos (3)

En el ejemplo, dentro del bloque sincronizado, tras pintar la cadena se llama al método `notify()` del objeto compartido para despertar al hilo que esté esperando el objeto (`notifyAll()` cuando varios hilos esperan el objeto). Después se llama al `wait()` del objeto para que el hilo quede suspendido y el que estaba en espera tome el objeto para pintar la cadena; el hilo permanecerá suspendido hasta que se produzca un `notify()` sobre el objeto. El último `notify()` es necesario para que los hilos finalicen sin bloqueos.

```
public void run() {  
    synchronized (objeto) {  
        for (int i = 0; i < 10; i++) {  
            objeto.PintaCadena(cad);  
            objeto.notify(); //aviso que ya he usado el objeto  
            try {  
                objeto.wait(); //esperar a que llegue un notify  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
        }  
    }  
    objeto.notify(); //despertar a todos los wait sobre el objeto  
} //fin bloque synchronized  
System.out.print("\n" + cad + " finalizado");  
} //run
```


El ejemplo productor-consumidor

Se produce cuando uno o más hilos producen datos a procesar y otros hilos los consumen. El problema surge cuando el productor produce datos más rápido que el consumidor los consuma, dando lugar a que el consumidor se salte algún dato. Igualmente, el consumidor puede consumir más rápido que el productor produce, entonces el consumidor puede recoger varias veces el mismo dato o puede no tener datos que recoger o puede detenerse.

Se definen 3 clases, la clase **Cola** que será el objeto compartido entre el productor y el consumidor; y las clases **Productor** y **Consumidor**. En el ejemplo el productor produce números y los coloca en una cola, estos serán consumidos por el consumidor. El recurso a compartir es la cola con los números. El productor genera números de 0 a 5 en un bucle for, y los pone en el objeto Cola mediante el método put(); después se visualiza y se hace un pausa con sleep(), durante este tiempo el hilo está en el estado Not Runnable (no ejecutable).

Después creamos otra clase con un método main() en el que creamos el consumidor y el productor.

Productor-Consumidor (2)

Para que todo funcione correctamente, los métodos put() y get() de la clase Cola deben tener la siguiente estructura:

```
public synchronized int get() {  
    while (!disponible) {  
        try {  
            wait();  
        } catch (InterruptedException e) { }  
    }  
    System.out.println("Se consume: " + numero);  
    disponible = false;  
    notify();  
    return numero;  
}
```

```
public synchronized void put(int valor) {  
    while (disponible){  
        try {  
            wait();  
        } catch (InterruptedException e) { }  
    }  
    numero = valor;  
    disponible = true;  
    System.out.println("Se produce: " + numero);  
    notify();  
}
```

