

2.5.1. CREAR Y ARRANCAR HILOS

Para crear un hilo extendemos la clase **Thread** o implementamos la interfaz **Runnable**. La siguiente línea de código crea un hilo donde *MiHilo* es una subclase de **Thread** (o una clase que implementa la interfaz **Runnable**), se le pasan dos argumentos que se deben definir en el constructor de la clase y se utilizan, por ejemplo, para iniciar variables del hilo:

```
MiHilo h = new MiHilo("Hilo 1", 200);
```

Si todo va bien en la creación del hilo tendremos en *h* el objeto hilo. Para arrancar el hilo usamos el método **start()** de esta manera si extiende **Thread**:

```
h.start();
```

Y si implementa **Runnable** lo arrancamos así:

```
new Thread(h).start();
```

Lo que hace este método es llamar al método **run()** del hilo que es donde se colocan las acciones que queremos que haga el hilo, cuando finalice el método finalizará también el hilo.

2.5.2. SUSPENSIÓN DE UN HILO

En ejemplos anteriores usamos el método **sleep()** para detener un hilo un número de milisegundos. Realmente el hilo no se detiene, sino que se queda “dormido” el número de milisegundos que indiquemos. Lo utilizábamos en los ejercicios de los contadores para que nos diese tiempo a ver cómo se van incrementando de 1 en 1.

El método **suspend()** permite detener la actividad del hilo durante un intervalo de tiempo indeterminado. Este método es útil cuando se realizan applets con animaciones y en algún momento se decide parar la animación para luego continuar cuando lo decida el usuario. Para volver a activar el hilo se necesita invocar al método **resume()**.

El método **suspend()** es un método obsoleto y tiende a no utilizarse porque puede producir situaciones de interbloqueos. Por ejemplo, si un hilo está bloqueando un recurso y este hilo se suspende puede dar lugar a que otros hilos que esperaban el recurso queden “congelados” ya que el hilo suspendido mantiene los recursos bloqueados. Igualmente el método **resume()** también está en desuso.

Para suspender de forma segura el hilo se debe introducir en el hilo una variable, por ejemplo *suspender* y comprobar su valor dentro del método **run()**, es lo que se hace en la llamada al método *suspender.esperandoParaReanudar()* del ejemplo siguiente. El método *Suspende()* del hilo da valor *true* a la variable para suspender el hilo. El método *Reanuda()* da el valor *false* para que detenga la suspensión y continúe ejecutándose el hilo:

```
class MyHilo extends Thread {
    private SolicitaSuspender suspender = new SolicitaSuspender();
    //petición de SUSPENDER HILO
    public void Suspende() { suspender.set(true); }
    //petición de CONTINUAR
    public void Reanuda() { suspender.set(false); }

    public void run() {
        try {
            while(haya trabajo por hacer) {
                . . . . .
                suspender.esperandoParaReanudar(); //comprobar
                . . . . .
            }
        }
    }
}
```

```

    } catch (InterruptedException exception) { }
  }
}

```

Para mayor claridad, se envuelve la variable (a la que se hacía alusión anteriormente) en la clase *SolicitaSuspend*, en esta clase se define el método *set()* que da el valor *true* o *false* a la variable y llama al método *notifyAll()*, este notifica a todos los hilos que esperan (han ejecutado un *wait()*) un cambio de estado sobre el objeto. En el método *esperandoParaReanudar()* se hace un *wait()* cuando el valor de la variable es *true*, el método *wait()* hace que el hilo espere hasta que le llegue un *notify()* o un *notifyAll()*;

```

public class SolicitaSuspend {
    private boolean suspend;

    public synchronized void set(boolean b) {
        suspend = b; //cambio de estado sobre el objeto
        notifyAll();
    }

    public synchronized void esperandoParaReanudar()
        throws InterruptedException {
        while (suspend)
            wait(); //SUSPENDER HILO HASTA RECIBIR notify() o notifyAll()
    }
}

```

El método *wait()* sólo puede ser llamado desde dentro de un método sincronizado (*synchronized*). Estos tres métodos: *wait()*, *notify()* y *notifyAll()*, se usan en sincronización de hilos. Forman parte de la clase **Object**, y no parte de **Thread** como es el caso de *sleep()*, *suspend()* y *resume()*. Se tratarán más adelante.

ACTIVIDAD 2.4

Partimos de las clases anteriores *MyHilo* y *SolicitaSuspend*. Vamos a modificar la clase *MyHilo*. Se define una variable contador y se inicia con valor 0. En el método *run()* y dentro de un bucle que controle el fin del hilo mediante una variable, se incrementa en 1 el valor del contador y se visualiza su valor, se incluye también un *sleep()* para que podamos ver los números. Haz una llamada al método *esperandoParaReanudar()* para suspender el hilo, el *sleep()* lo podemos hacer antes o después. Crea en la clase un método que devuelva el valor del contador. Al finalizar el bucle visualiza un mensaje.

Para probar las clases crea un método *main()*, en el que introducirás una cadena por teclado en un proceso repetitivo hasta introducir un *. Si la cadena introducida es S se suspenderá el hilo, si la cadena es R se reanudará el hilo. El hilo se lanzará solo una vez después de introducir la primera cadena. Al finalizar el proceso repetitivo visualizar el valor del contador y finalizar el hilo. Comprueba que todos los mensajes se visualicen correctamente.

Realiza el Ejercicio 8.

2.5.3. PARADA DE UN HILO

El método *stop()* detiene la ejecución de un hilo de forma permanente y ésta no se puede reanudar con el método *start()*:

```
h.stop();
```

Este método al igual que **suspend()**, **resume()** y **destroy()** han sido abolidos en Java 2 para reducir la posibilidad de interbloqueo. El método **run()** no libera los bloqueos que haya adquirido el hilo, y si los objetos están en un estado inconsistente los demás hilos podrán verlos y modificarlos en ese estado. En lugar de usar este método se puede usar una variable como se vio en el estado **Dead** del hilo.

El método **isAlive()** devuelve *true* si el hilo está vivo, es decir ha llamado a su método **run()** y aún no ha terminado su ejecución o no ha sido detenido con **stop()**; en caso contrario devuelve *false*. En ejemplos anteriores vimos cómo se usaba este método.

El método **interrupt()** envía una petición de interrupción a un hilo. Si el hilo se encuentra bloqueado por una llamada a **sleep()** o **wait()** se lanza una excepción *InterruptedException*. El método **isInterrupted()** devuelve *true* si el hilo ha sido interrumpido, en caso contrario devuelve *false*. El siguiente ejemplo usa interrupciones para detener el hilo. En el método **run()** se comprueba en el bucle **while** si el hilo está interrumpido, si no lo está se ejecuta el código. El método **interrumpir()** ejecuta el método **interrupt()** que lanza una interrupción que es recogida por el manejador (**catch**):

```
public class HiloEjemploInterrup extends Thread {
    public void run() {
        try {
            while (!isInterrupted()) {
                System.out.println("En el Hilo");
                Thread.sleep(10);
            }
        } catch (InterruptedException e) {
            System.out.println("HA OCURRIDO UNA EXCEPCIÓN");
        }
        System.out.println("FIN HILO");
    }

    public void interrumpir() {
        interrupt();
    }

    public static void main(String[] args) {
        HiloEjemploInterrup h = new HiloEjemploInterrup();
        h.start();
        for(int i=0; i<1000000000; i++) ;//no hago nada
        h.interrumpir();
    }
}
```

Un ejemplo de ejecución muestra la siguiente información:

```
En el Hilo
En el Hilo
HA OCURRIDO UNA EXCEPCIÓN
FIN HILO
```

Si en el código anterior quitamos la línea *Thread.sleep(10);* también hay que quitar el bloque **try-catch**, la interrupción será recogida por el método **isInterrupted()**, que será *true* con lo que la ejecución del hilo terminará ya que finaliza el método **run()**.

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 *h1.join()*, el hilo se queda en espera hasta que muera el

hilo sobre el que se realiza el `join()`, en este caso *h1*. En el siguiente ejemplo el método `run()` de la clase *HiloJoin* visualiza en un bucle *for* un contador que empieza en 1 hasta un valor *n* que recibe el constructor del hilo:

```
class HiloJoin extends Thread {
    private int n;
    public HiloJoin(String nom, int n) {
        super(nom);
        this.n=n;
    }
    public void run() {
        for(int i=1; i<= n; i++) {
            System.out.println(getName() + ": " + i);
            try {
                sleep(1000);
            } catch (InterruptedException ignore) {}
        }
        System.out.println("Fin Bucle "+getName());
    }
}
//
```

```
public class EjemploJoin {
    public static void main(String[] args) {
        HiloJoin h1 = new HiloJoin("Hilo1",2);
        HiloJoin h2 = new HiloJoin("Hilo2",5);
        HiloJoin h3 = new HiloJoin("Hilo3",7);
        h1.start();
        h2.start();
        h3.start();
        try {
            h1.join(); h2.join(); h3.join();
        } catch (InterruptedException e) {}
        System.out.println("FINAL DE PROGRAMA");
    }
}
//
```

En el método `main()` se crean 3 hilos, cada uno da un valor diferente a la *n*, el primero el valor más pequeño y el tercero el valor más grande, parece lógico que por los valores del contador el primer hilo debe terminar el primero y el tercer hilo el último. Llamando a `join()` podemos hacer que `main()` espere a la finalización de los hilos y cada hilo finalice en el orden marcado según la llamada a `join()`, cuando salgan del bloque `try-catch` los tres hilos habrán finalizado y el texto FINAL DE PROGRAMA se visualizará al final.

La ejecución muestra la siguiente salida:

```
Hilo1: 1
Hilo3: 1
Hilo2: 1
Hilo2: 2
Hilo3: 2
Hilo1: 2
Hilo2: 3
Hilo3: 3
Fin Bucle Hilo1
Hilo2: 4
Hilo3: 4
Hilo2: 5
Hilo3: 5
```

```

Fin Bucle Hilo2
Hilo3: 6
Hilo3: 7
Fin Bucle Hilo3
FINAL DE PROGRAMA

```

Si en el ejemplo anterior quitamos los `join()` veremos que el texto FINAL DE PROGRAMA no se mostrará al final. El método `join()` puede lanzar la excepción *InterruptedException*, por ello se incluye en un bloque `try-catch`.

ACTIVIDAD 2.5

Modifica el applet de la Actividad 2.3 de manera que la finalización de los hilos no se realice con el método `stop()` sino que se realice de alguna de las formas seguras vistas anteriormente (usando interrupciones o usando una variable para controlar el fin del hilo).

Realiza el ejercicio 9.

2.6. GESTIÓN DE PRIORIDADES

En el lenguaje de programación Java, cada hilo tiene una prioridad. Por defecto, un hilo hereda la prioridad del hilo padre que le crea, esta se puede aumentar o disminuir mediante el método `setPriority()`. El método `getPriority()` devuelve la prioridad del hilo.

La prioridad no es más que un valor entero entre 1 y 10, siendo el valor 1 la mínima prioridad, `MIN_PRIORITY`; y el valor 10 la máxima, `MAX_PRIORITY`. `NORM_PRIORITY` se define como 5. El planificador elige el hilo que debe ejecutarse en función de la prioridad asignada; se ejecutará primero el hilo de mayor prioridad. Si dos o más hilos están listos para ejecutarse y tienen la misma prioridad, la máquina virtual va cediendo control de forma cíclica (*round-robin*).

El planificador es la parte de la máquina virtual de Java que decide qué hilo ejecutar en cada momento. Da más ventaja a hilos con mayor prioridad; hilos de igual prioridad en algún momento se ejecutarán.

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.

En el siguiente ejemplo se crea una clase que extiende `Thread`, se define una variable contador que será incrementada en el método `run()`, se define un método para obtener el valor de la variable y otro método para finalizar el hilo, en el constructor se le da nombre al hilo:

```

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(2);
        } catch (Exception e) { }
        c++;
    }
    System.out.println("Fin hilo " + this.getName());
}

}
//HiloPrioridad1

```

Varias ejecuciones del programa muestran las siguientes salidas, se puede observar que el máximo valor del contador lo obtiene el objeto hilo con prioridad máxima, y el mínimo el de prioridad mínima:

h2 (Prioridad Maxima): 4856

Fin hilo Hilo3

Fin hilo Hilo1

Fin hilo Hilo2

h1 (Prioridad Normal): 4855

h3 (Prioridad Minima): 4854

h2 (Prioridad Maxima): 4767

h1 (Prioridad Normal): 4684

h3 (Prioridad Minima): 4668

Fin hilo Hilo2

Fin hilo Hilo1

Fin hilo Hilo3

h2 (Prioridad Maxima): 4565

h1 (Prioridad Normal): 4545

h3 (Prioridad Minima): 4523

Fin hilo Hilo2

Fin hilo Hilo1

Fin hilo Hilo3

Pero no siempre ocurre esto. Podemos encontrar la siguiente salida en la que se observa que los valores de los contadores no dependen de la prioridad asignada al hilo:

h2 (Prioridad Maxima): 4822

Fin hilo Hilo3

Fin hilo Hilo2

Fin hilo Hilo1

h1 (Prioridad Normal): 4823

h3 (Prioridad Minima): 4823

h2 (Prioridad Maxima): 4518

Fin hilo Hilo2

h1 (Prioridad Normal): 4518

h3 (Prioridad Minima): 4517

Fin hilo Hilo1

Fin hilo Hilo3

En el siguiente ejemplo se asignan diferentes prioridades a cada uno de los hilos de la clase *EjemploHiloPrioridad2* que se crean. En la ejecución se puede observar que no siempre el hilo con más prioridad es el que antes se ejecuta:

```
public class EjemploHiloPrioridad2 extends Thread {
    EjemploHiloPrioridad2(String nom) {
        this.setName(nom);
    }

    public void run() {
        System.out.println("Ejecutando [" + getName() + "]");
        for (int i = 1; i <= 5; i++)
            System.out.println("\t(" + getName() + ": " + i + ")");
    }
}
```

```
public static void main(String[] args) {
    EjemploHiloPrioridad2 h1 = new EjemploHiloPrioridad2("Uno");
    EjemploHiloPrioridad2 h2 = new EjemploHiloPrioridad2("Dos");
    EjemploHiloPrioridad2 h3 = new EjemploHiloPrioridad2("Tres");
    EjemploHiloPrioridad2 h4 = new EjemploHiloPrioridad2("Cuatro");
    EjemploHiloPrioridad2 h5 = new EjemploHiloPrioridad2("Cinco");

    //asignamos prioridad
    h1.setPriority(Thread.MIN_PRIORITY);
    h2.setPriority(3);
    h3.setPriority(Thread.NORM_PRIORITY);
    h4.setPriority(7);
    h5.setPriority(Thread.MAX_PRIORITY);

    //se ejecutan los hilos
    h1.start();
    h2.start();
    h3.start();
    h4.start();
    h5.start();
}
```

Un ejemplo de ejecución muestra la siguiente salida:

```
Ejecutando [Cuatro]
  (Cuatro: 1)
  (Cuatro: 2)
  (Cuatro: 3)
  (Cuatro: 4)
  (Cuatro: 5)
Ejecutando [Tres]
  (Tres: 1)
  (Tres: 2)
  (Tres: 3)
  (Tres: 4)
  (Tres: 5)
Ejecutando [Cinco]
  (Cinco: 1)
  (Cinco: 2)
  (Cinco: 3)
  (Cinco: 4)
  (Cinco: 5)
Ejecutando [Dos]
  (Dos: 1)
  (Dos: 2)
  (Dos: 3)
  (Dos: 4)
  (Dos: 5)
Ejecutando [Uno]
  (Uno: 1)
  (Uno: 2)
  (Uno: 3)
  (Uno: 4)
  (Uno: 5)
```


A la hora de programar hilos con prioridades hemos de tener en cuenta que el comportamiento no está garantizado y dependerá de la plataforma en la que se ejecuten los programas y de las aplicaciones que se ejecuten al mismo tiempo. En la práctica casi nunca hay que establecer a mano las prioridades.

Cuando un hilo entra en ejecución y no cede voluntariamente el control para que puedan ejecutarse otros hilos, se dice que es un "hilo egoísta". Algunos sistemas operativos, como Windows, combaten estas situaciones con una estrategia de planificación por división de tiempos (*time-slicing* o tiempo compartido), que opera con hilos de igual prioridad que compiten por la CPU. En estas condiciones el sistema operativo divide el tiempo de proceso de la CPU en espacios de tiempo y asigna el tiempo de proceso a los hilos dependiendo de su prioridad. Así se impide que uno de ellos se apropie del sistema durante un intervalo de tiempo prolongado.

ACTIVIDAD 2.6

Prueba los ejemplos anteriores variando la prioridad y el orden de ejecución de cada hilo. Comprueba los resultados para el primer ejemplo y para el segundo.

Realiza el ejercicio 6

2.7. 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 siguiente ejemplo dos hilos comparten un objeto de la clase *Contador*. Esta clase define un atributo contador y tres métodos, uno de ellos incrementa una unidad su valor, el otro lo decremента y el tercero devuelve su valor; el constructor asigna un valor inicial al contador:

```
class Contador {
    private int c = 0; //atributo contador
    Contador(int c) { this.c = c; }
    public void incrementa() { c = c + 1; }
    public void decremента() { c = c - 1; }
    public int valor() { return c; }
} // CONTADOR
```

Para probar el objeto compartido se definen dos clases que extienden *Thread*. En la clase *HiloA* se usa el método del objeto contador que incrementa en uno su valor. En la clase *HiloB* se usa el método que decremента su valor. 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:

```
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(); //incrementa el contador
            try {
                sleep(100);
            } catch (InterruptedException e) {}
        }
    }
}
```

```

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

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(); //decrementa el contador
            try {
                sleep(100);
            } catch (InterruptedException e) {}
        }
        System.out.println(getName() + " contador vale " +
                           contador.valor());
    }
} // FIN HILOB

```

A continuación se crea el método *main()*, donde primero se define un objeto de la clase *Contador* y se le asigna el valor inicial de 100. A continuación, se crean los dos hilos pasándoles dos parámetros: un nombre y el objeto *Contador*. Seguidamente se inicia la ejecución de los hilos:

```

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();
    }
}

```

Nos puede dar la impresión que al ejecutar los hilos el valor del contador en el hilo A debería ser 400, ya que empieza en 100 y le suma 300; y en B, 100 ya que se resta 300; pero no es así. Al ejecutar el programa los valores de salida pueden no ser los esperados y variará de una ejecución a otra:

```

HiloB contador vale 100
HiloA contador vale 100

```

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

```

HiloA contador vale 400
HiloB contador vale 100

```

```

HiloB contador vale 100
HiloA contador vale 100

```

```

HiloA contador vale 43
HiloB contador vale 43

```

```
HiloB contador vale 100
HiloA contador vale 400
```

```
HiloB contador vale -200
HiloA contador vale 100
```

2.7.1. BLOQUES SINCRONIZADOS

Una forma de evitar que esto suceda es hacer que las operaciones de incremento y decremento del objeto contador se hagan de forma atómica, es decir, si estamos realizando la suma nos aseguramos que nadie realice la resta hasta que no terminemos la suma. Esto se puede lograr añadiendo la palabra **synchronized** a la parte de código que queramos que se ejecute de forma atómica. Java utiliza los **bloques synchronized** para implementar las **regiones críticas** (que se tratarán en el Capítulo 1). El formato es el siguiente:

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

Los métodos `run()` de las clases *HiloA* e *HiloB* se pueden sustituir por los siguientes; para el *HiloB*:

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

Para el *HiloA*:

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

El bloque **synchronized** o **región crítica** (que aparece sombreado) 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. Es decir, le pregunta si no hay otro hilo ejecutando algún bloque sincronizado con ese objeto. Si está tomado por otro hilo, entonces el hilo actual se suspende y se pone en espera hasta que se libere el bloque. Si está libre, el hilo actual bloquea el objeto y ejecuta el bloque; el siguiente hilo que intente ejecutar un bloque sincronizado con ese objeto, será puesto en espera. El bloqueo del objeto se libera cuando el hilo que lo tiene tomado sale del bloque porque termina la ejecución, ejecuta un `return` o lanza una excepción.

La compilación y ejecución muestra la siguiente salida (se ha guardado el ejemplo anterior con los cambios en *CompartirInf2.java*):

HiloA contador vale 400
HiloB contador vale 100

Si se cambia el orden de la ejecución de los hilos, primero el HiloB y luego el HiloA, la salida será:

HiloB contador vale -200
HiloA contador vale 100

ACTIVIDAD 2.7

Crea un programa Java que lance cinco hilos, cada uno incrementará una variable contador de tipo entero, compartida por todos, 5000 veces y luego saldrá. Comprobar el resultado final de la variable. ¿Se obtiene el resultado correcto? Ahora sincroniza el acceso a dicha variable. Lanza los hilos primero mediante la clase **Thread** y luego mediante el interfaz **Runnable**. Comprueba el resultado.

2.7.2. MÉTODOS SINCRONIZADOS

Se debe evitar la sincronización de bloques de código 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 los 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:

```
public class ContadorSincronizado {
    private int c = 0;

    public synchronized void incrementa() {
        c++;
    }

    public synchronized void decrementa() {
        c--;
    }

    public synchronized int valor() {
        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.

Veamos mediante clases como sería el ejemplo de compartir una cuenta sin usar métodos sincronizados. Se define la clase *Cuenta*, define 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:

```
class Cuenta {
    private int saldo ;

    Cuenta (int s) { saldo = s; }    //inicializa saldo actual
    int getSaldo() { return saldo; } //devuelve el saldo
    void restar(int cantidad) {      //se resta la cantidad al saldo
        saldo = saldo - cantidad;
    }

    void RetirarDinero(int cant, String nom) {
        if (getSaldo() >= cant) {
            System.out.println(nom + ": SE VA A RETIRAR SALDO (ACTUAL ES: " +
                               getSaldo() + ")" );
            try {
                Thread.sleep(500);
            } catch (InterruptedException ex) { }

            restar(cant); //resta la cantidad del saldo

            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());
        }
    } //RetirarDinero
} //Cuenta
```

A continuación, se crea la clase *SacarDinero* que extiende **Thread** y usa la clase *Cuenta* para retirar el dinero. El constructor recibe una cadena, para dar nombre al hilo; y la cuenta que será compartida por varias personas. En el 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:


```
class SacarDinero extends Thread {
    private Cuenta c;
    String nom;
    public SacarDinero(String n, Cuenta c) {
        super(n);
        this.c = c;
    }
    public void run() {
        for (int x = 1; x <= 4; x++) {
            c.RetirarDinero(10, getName());
        }
    } // run
}
```

Por último se crea 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*, imaginemos que son las dos personas que comparten la cuenta, y se inician los hilos:

```
public class CompartirInf3 {
    public static void main(String[] args) {
        Cuenta c = new Cuenta(40);
        SacarDinero h1 = new SacarDinero("Ana", c);
        SacarDinero h2 = new SacarDinero("Juan", c);
        h1.start();
        h2.start();
    }
}
```

Al ejecutar puede ocurrir que se permita retirar saldo cuando este es 0:

```
Ana: SE VA A RETIRAR SALDO (ACTUAL ES: 40)
Juan: SE VA A RETIRAR SALDO (ACTUAL ES: 40)
      Ana retira =>10 ACTUAL(30)
Ana: SE VA A RETIRAR SALDO (ACTUAL ES: 30)
      Juan retira =>10 ACTUAL(20)
Juan: SE VA A RETIRAR SALDO (ACTUAL ES: 20)
      Ana retira =>10 ACTUAL(10)
Ana: SE VA A RETIRAR SALDO (ACTUAL ES: 10)
      Juan retira =>10 ACTUAL(0)
Juan No puede retirar dinero, NO HAY SALDO(0)
Juan No puede retirar dinero, NO HAY SALDO(0)
      Ana retira =>10 ACTUAL(-10)
SALDO NEGATIVO => -10
Ana No puede retirar dinero, NO HAY SALDO(-10)
SALDO NEGATIVO => -10
```



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 declaramos 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. La forma de declararlo es la siguiente:

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

O bien:

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

El método *RetirarDinero()* en nuestro ejemplo quedaría así:

```
synchronized void RetirarDinero(int cant, String nom) {
    //las mismas instrucciones que antes
}
```

La ejecución mostraría la siguiente salida, recuerda que de una ejecución a otra puede variar, pero en este caso el saldo no será negativo:

```
Ana: SE VA A RETIRAR SALDO (ACTUAL ES: 40)
    Ana retira =>10 ACTUAL(30)
Juan: SE VA A RETIRAR SALDO (ACTUAL ES: 30)
    Juan 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)
Ana No puede retirar dinero, NO HAY SALDO(0)
```

Se debe tener en cuenta que la sincronización disminuye el rendimiento de una aplicación, por tanto, debe emplearse solamente donde sea estrictamente necesario.

ACTIVIDAD 2.8

Crea una clase de nombre *Saldo*, con un atributo que nos indica el saldo, el constructor que da un valor inicial al saldo. Crea varios métodos uno para obtener el saldo y otro para dar valor al saldo, en estos dos métodos añade un `sleep()` aleatorio. Y otro método que reciba una cantidad y se la añada al saldo, este método debe informar de quién añade cantidad al saldo, la cantidad que añade, el estado inicial del saldo (antes de añadir la cantidad) y el estado final del saldo después de añadir la cantidad. Define los parámetros necesarios que debe de recibir este método y defínelo como **synchronized**.

Crea una clase que extienda **Thread**, desde el método `run()` hemos de usar el método de la clase *Saldo* que añade la cantidad al saldo. Averigua los parámetros que se necesita en el constructor. No debe visualizar nada en pantalla.

Crea en el método `main()` un objeto *Saldo* asignándole un valor inicial. Visualiza el saldo inicial. Crea varios hilos que compartan ese objeto *Saldo*. A cada hilo le damos un nombre y le asignamos una cantidad. Lanzamos los hilos y esperamos a que finalicen para visualizar el saldo final del objeto *Saldo*. Comprueba los resultados quitando **synchronized** del método de la clase *Saldo* que reciba la cantidad y se la añada al saldo.

Realiza el Ejercicio 11.

2.7.3. BLOQUEO DE HILOS

En el siguiente ejemplo creamos una clase que define un método que recibe un String y lo pinta:

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

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*. Los hilos usarán el método del objeto compartido para pintar una cadena, esta cadena es enviada al crear el hilo (*new HiloCadena (objeto compartido, cadena)*):

```
public class BloqueoHilos {
    public static void main(String[] args) {
        ObjetoCompartido com = new ObjetoCompartido();
        HiloCadena a = new HiloCadena (com, " A ");
        HiloCadena b = new HiloCadena (com, " B ");
        a.start();
        b.start();
    }
} //BloqueoHilos
```

La clase *HiloCadena* extiende **Thread**; en su método *run()* invoca al método *PintaCadena()* del objeto compartido dentro de un bucle for:

```
class HiloCadena extends Thread {
    private ObjetoCompartido objeto;
    String cad;

    public HiloCadena (ObjetoCompartido c, String s) {
        this.objeto = c;
        this.cad=s;
    }

    public void run() {
        for (int j = 0; j < 10; j++)
            objeto.PintaCadena (cad) ;
    } //run
} //HiloCadena
```

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". Parece que una primera aproximación para solucionarlo sería sincronizar el trozo de código que hace uso del objeto compartido (dentro del método *run()*):

```
synchronized (objeto) {
    for (int j = 0; j < 10; j++)
        objeto.PintaCadena (cad) ;
}
```

Pero al ejecutarlo, la salida no es la esperada ya que 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.

En el ejemplo, dentro del bloque sincronizado y después de pintar la cadena se invocará al método **notify()** del objeto compartido para despertar al hilo que esté esperando el objeto (**notifyAll()** cuando varios hilos esperan el objeto). Inmediatamente después se llama al método **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 correctamente y ninguno quede bloqueado:

```
public void run() {
    synchronized (objeto) {

        for (int j = 0; j < 10; j++) {
            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
```

La ejecución del ejemplo muestra la siguiente salida:

```
A B A B A B A B A B A B A B A B A B
B finalizado
A finalizado
```

Los métodos **notify()** y **wait()** pueden ser invocados sólo desde dentro de un método sincronizado o dentro de un bloque o una sentencia sincronizada.

2.7.4. EL MODELO PRODUCTOR-CONSUMIDOR

Un problema típico de sincronización es el que representa el modelo **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, etc.

Por ejemplo, imaginemos una aplicación donde un hilo (el productor) escribe datos en un fichero mientras que un segundo hilo (el consumidor) lee los datos del mismo fichero; en este caso los hilos comparten un mismo recurso (el fichero) y deben sincronizarse para realizar su tarea correctamente.

EJEMPLO PRODUCTOR-CONSUMIDOR

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 esta en el estado *Not Runnable* (no ejecutable):

```
public class Productor extends Thread {
    private Cola cola;
    private int n;

    public Productor(Cola c, int n) {
        cola = c;
        this.n = n;
    }

    public void run() {
        for (int i = 0; i < 5; i++) {
            cola.put(i); //pone el número
            System.out.println(i + ">Productor : " + n
                               + ", produce: " + i);
            try {
                sleep(100);
            } catch (InterruptedException e) { }
        }
    }
}
```

La clase *Consumidor* es muy similar a la clase *Productor*, solo que en lugar de poner un número en el objeto *Cola* lo recoge llamando al método *get()*. En este caso no se ha puesto pausa, con esto hacemos que el consumidor sea más rápido que el productor:

```
public class Consumidor extends Thread {
    private Cola cola;
    private int n;

    public Consumidor(Cola c, int n) {
        cola = c;
        this.n = n;
    }

    public void run() {
        int valor = 0;
        for (int i = 0; i < 5; i++) {
            valor = cola.get(); //recoge el número
            System.out.println(i + ">Consumidor: " + n
                               + ", consume: " + valor);
        }
    }
}
```

```

    }
}

```

La clase *Cola* define 2 atributos y dos métodos. En el atributo *numero* se guarda el número entero y el atributo *disponible* se utiliza para indicar si hay disponible o no un número en la cola. El método *put()* guarda un entero en el atributo *numero* y hace que este esté disponible en la cola para que pueda ser consumido poniendo el valor *true* en *disponible* (cola llena). El método *get()* devuelve el entero de la cola si está disponible (*disponible=true*) y antes pone la variable a *false* indicando cola vacía; si el número no está en la cola (*disponible=false*) devuelve -1;

```

public class Cola {
    private int numero;
    private boolean disponible = false; //inicialmente cola vacia

    public int get() {
        if(disponible) {           //hay número en la cola
            disponible = false;    //se pone cola vacia
            return numero;         //se devuelve
        }
        return -1;                //no hay número disponible, cola vacia
    }

    public void put(int valor) {
        numero = valor;           //coloca valor en la cola
        disponible = true;        //disponible para consumir, cola llena
    }
}

```

En el método *main()* que usa las clases anteriores creamos 3 objetos, un objeto de la clase *Cola*, un objeto de la clase *Productor* y otro objeto de la clase *Consumidor*. Al constructor de las clases *Productor* y *Consumidor* le pasamos el objeto compartido de la clase *Cola* y un número entero que lo identifique:

```

public class Produc_Consum {
    public static void main(String[] args) {
        Cola cola = new Cola();
        Productor p = new Productor(cola, 1);
        Consumidor c = new Consumidor(cola, 1);
        p.start();
        c.start();
    }
}

```

Al ejecutar se produce la siguiente salida, en la que se puede observar que el consumidor va más rápido que el productor (al que se le puso un *sleep()*) y no consume todos los números cuando se producen; el numerito de la izquierda de cada fila representa la iteración:

```

0=>Productor1 : produce: 0
0=>Consumidor1: consume: 0
1=>Consumidor1: consume: -1
2=>Consumidor1: consume: -1
3=>Consumidor1: consume: -1
4=>Consumidor1: consume: -1
1=>Productor1 : produce: 1
2=>Productor1 : produce: 2
3=>Productor1 : produce: 3
4=>Productor1 : produce: 4

```

En la iteración 0, el productor produce un 0 e inmediatamente el consumidor lo consume, la cola se queda vacía. En la iteración 1 el consumidor consume -1 que indica que la cola está vacía porque la iteración 1 del productor no se ha producido. En la iteración 2 pasa lo mismo el consumidor toma -1 porque el productor aún no ha dejado valor en la cola. Y así sucesivamente. La salida deseada es la siguiente: en cada iteración el productor produce un número (llena la cola) e inmediatamente el consumidor lo consume (la vacía):

```
0=>Productor1 : produce: 0
0=>Consumidor1: consume: 0
1=>Productor1 : produce: 1
1=>Consumidor1: consume: 1
2=>Productor1 : produce: 2
2=>Consumidor1: consume: 2
3=>Productor1 : produce: 3
3=>Consumidor1: consume: 3
4=>Productor1 : produce: 4
4=>Consumidor1: consume: 4
```

ACTIVIDAD 2.9

Prueba las clases *Productor* y *Consumidor* quitando el método *sleep()* del productor o añadiendo uno al consumidor para hacer que uno sea más rápido que otro. ¿Se obtiene la salida deseada?

Para obtener la salida anterior es necesario que los hilos estén sincronizados. Primero hemos de declarar los métodos *get()* y *put()* de la clase Cola como **synchronized**, de esta manera el productor y consumidor no podrán acceder simultáneamente al objeto Cola compartido; es decir el productor no puede cambiar el valor de la cola cuando el consumidor esté recogiendo su valor; y el consumidor no puede recoger el valor cuando el productor lo esté cambiando:

```
public synchronized int get() {
    //instrucciones
}
public synchronized void put(int valor) {
    //instrucciones
}
```

En segundo lugar, es necesario mantener una coordinación entre el productor y el consumidor de forma que cuando el productor ponga un número en la cola avise al consumidor de que la cola está disponible para recoger su valor; y al revés, cuando el consumidor recoja el valor de la cola debe avisar al productor de que la cola ha quedado vacía. A su vez, el consumidor deberá esperar hasta que la cola se llene y el productor esperará hasta que la cola esté nuevamente vacía para poner otro número.

Para mantener esta coordinación usamos los métodos *wait()*, *notify()* y *notifyAll()* vistos anteriormente. Estos sólo pueden ser invocados desde dentro de un método sincronizado o dentro de un bloque o una sentencia sincronizada.

El método *get()* tiene que esperar a que la cola se llene (Figura 2.6), esto se realiza en el bucle *while*: mientras la cola esté vacía, es decir *disponible* es *false* (*while (!disponible)*), espero (*wait*). Se sale del bucle cuando llega un valor, en este caso se vuelve a poner *disponible* a *false* (porque se va a devolver quedando la cola vacía de nuevo), se notifica a todos los hilos que comparten el objeto este hecho y se devuelve el valor (Figura 2.7):

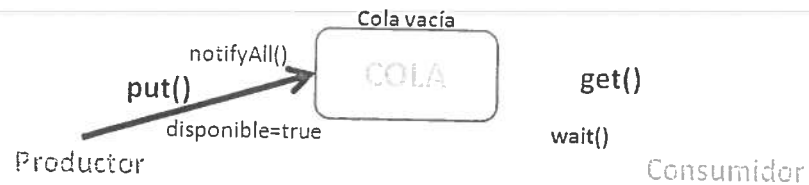


Figura 2.6. Método get() espera.



Figura 2.7. Método get() devuelve valor, put() espera.

Método get() sincronizado	Método get() anterior
<pre> public synchronized int get() { while (!disponible) { try { wait(); } catch (InterruptedException e) { } } //visualizar número disponible = false; notify(); return numero; } </pre>	<pre> public int get() { if(disponible) { disponible=false; return numero; } return -1; } </pre>

El método *put()* tiene que esperar a que la cola se vacíe para poner el valor, entonces espera (*wait*) mientras haya valor en la cola (*while (disponible)*). Cuando la cola se vacía, *disponible* es *false*, entonces se sale del bucle, se asigna el valor a la cola, se vuelve a poner disponible a *true* (porque la cola está llena) y se notifica a todos los hilos que comparten el objeto este hecho:

Método put() sincronizado	Método put() anterior
<pre> public synchronized void put(int valor) { while (disponible){ try { wait(); } catch (InterruptedException e) { } } numero = valor; disponible = true; //visualizar número notify(); } </pre>	<pre> public void put(int valor) { numero = valor; disponible=true; } </pre>

