

UNIDAD 2 - PROGRAMACIÓN MULTIHILO

Programación de Servicios y Procesos

DAM-2º

Profesoras : Raquel Barreales Quintanilla

UNIDAD 2 - PROGRAMACIÓN MULTIHILO - contenido

- Hilos o subprocessos. Gestión de hilos
- Creación de hilos en Java
- Interfaz Runnable
- Estados de un hilo
- Interrumpir un hilo
- Suspensión de un hilo
- Método join()
- Prioridad de hilos
- Sincronización de hilos
- Modelo productor-consumidor

Programación multihilo

- Para programar concurrentemente podemos dividir nuestro programa en hilos o subprocessos.
- La programación multihilo permite ejecutar diferentes subprocessos o hilos de ejecución al mismo tiempo. Nos permite realizar diferentes tareas en una aplicación simultáneamente.
- Un subprocesso o hilo es la unidad de procesamiento más pequeña que puede programar un sistema operativo. Un hilo (hebra, thread) es una secuencia de código en ejecución dentro del contexto de un proceso.
- Esta es una tarea que se puede ejecutar al mismo tiempo que otra tarea.

Hilos

- Podemos definir un proceso como una entidad formada por una o más unidades de ejecución denominadas **hilos** (threads) y un conjunto de recursos asociados.
- Los procesos no comparten memoria entre ellos, son independientes. Llevan información sobre su estado e interactúan con otros procesos a través de mecanismos de comunicación dados por el sistema.
- El proceso es el que puede acceder de forma protegida a los recursos del sistema y se encarga de su gestión entre los **hilos**.
- Los hilos no pueden ejecutarse ellos solos, necesitan de la supervisión de un proceso padre para ejecutarse.
- Cuando se crea un proceso, el sistema operativo crea un **hilo primario** el cual puede a su vez crear más **hilos**.
- El **proceso** sigue en ejecución mientras al menos uno de sus **hilos** de ejecución siga activo.
- A los hilos se les conoce a menudo como procesos ligeros.

Programación secuencial:



Programación concurrente con 4 procesadores y 4 hilos ejecutándose a la vez



Ejemplos de uso de hilos

Ejemplos:

- Un programa que controla sensores en una fábrica, cada sensor puede ser un hilo independiente y recoge un tipo de información y todos deben controlarse de forma simultánea.
- Un procesador de textos, puede tener un hilo comprobando la gramática mientras se escribe y otro hilo guardando el texto en disco cada cierto tiempo.
- En un servidor web, un hilo puede atender las peticiones entrantes y crear un hilo por cada cliente que tenga que servir.

Creación de hilos en Java

Los Hilos o los “Threads” en Java, son básicamente una forma de poder **ejecutar varios procesos simultáneamente** en nuestros programas en Java.

En Java existen dos clases para crear hilos, ambas pertenecen al paquete `java.lang`:

- Extendiendo la clase **Thread**.
- O implementando la interfaz **Runnable**.

La clase Thread

Thread permite que un programa funcione de manera más eficiente al hacer varias cosas al mismo tiempo.

La forma más simple de añadir funcionalidad de hilo a una clase es extender la clase **Thread**, es decir, crear una subclase de esta. Esta subclase debe sobrescribir el método `run()` con las acciones que el hilo debe desarrollar.

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

La clase Thread cont.

Para crear un objeto hilo con el comportamiento de NombreHilo escribo:

```
NombreHilo h = new NombreHilo();
```

Para inicia su ejecución utilizamos el método start()

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

La clase Thread – Ejemplo PrimerHilo



```
public class PrimerHilo extends Thread {  
    //propiedades, constructores y métodos de la clase  
    private int x;  
    public PrimerHilo(int x) {  
        this.x = x;  
    }  
    public void run(){  
        //acciones que lleva a cabo el hilo  
        for(int i=0; i<x;i++){  
            System.out.println("En el Hilo...."+i); }  
    }  
}
```

Ver código Ejemplo- PrimerHilo

Creamos una clase **HiloEjemplo1** que extiende de Thread.

En el constructor del hilo nos informará del hilo que se está creando
"CREANDO HILO: " + hilo

La misión del hilo (se implementa en el método **run()**) es visualizar un mensaje donde se muestre el hilo que se está ejecutando y el contenido de un contador (bucle while $\text{contador} \leq 5$)

En otra clase **Crear3Hilos** crea tres hilos, una vez que crea los tres visualiza por pantalla "3 HILOS CREADOS..."

```
public class HiloEjemplo1 extends Thread {  
  
    private int c; //cuenta cada hilo  
    private int hilo;  
  
    //constructor  
    public HiloEjemplo1(int hilo ){  
        this.hilo = hilo;  
        System.out.println("Creando hilo"+ hilo);  
    }  
    //metodo run  
    public void run(){  
        c=0;  
        //el hilo cuenta del 0 al 5  
        while (c<=5) {  
            System.out.println("Hilo: " + hilo + " Contador: "+ c);  
            c++;  
        }  
    }  
}
```

Código Crear3Hilos

```
public class Crear3Hilos {  
    public static void main(String[] args) {  
        HiloEjemplo1 hilo = null;  
        for (int i=0; i<3 ; i++) {  
            hilo = new HiloEjemplo1(i+1);  
            hilo.start();  
        }  
        System.out.println("he creado 3 hilos");  
    }  
}
```

El resultado variará en cada ejecución del programa :

Creando hilo1

Creando hilo2

Creando hilo3

he creado 3 hilos

Hilo: 2 Contador: 0

Hilo: 3 Contador: 0

Hilo: 3 Contador: 1

Hilo: 1 Contador: 0

Hilo: 3 Contador: 2

Hilo: 2 Contador: 1

HiloEjemplo2

Crear una clase que extienda de Thread

Dentro de la clase se define el constructor, el método **run()** con la funcionalidad que realizara el hilo y el método **main()** donde se **creara 3 hilos.**

La misión del hilo, descrita en el **método run()**, será visualizar un mensaje donde se muestre el nombre del hilo que se está ejecutando y el contenido de un contador .

Se utiliza una variable para mostrar el nombre del hilo que se ejecuta, esta variable se pasa al constructor y este se lo pasa al constructor base Thread mediante la palabra **super.** , para acceder a este nombre se usa. el método **getName()**

Programación de Servicios y Procesos

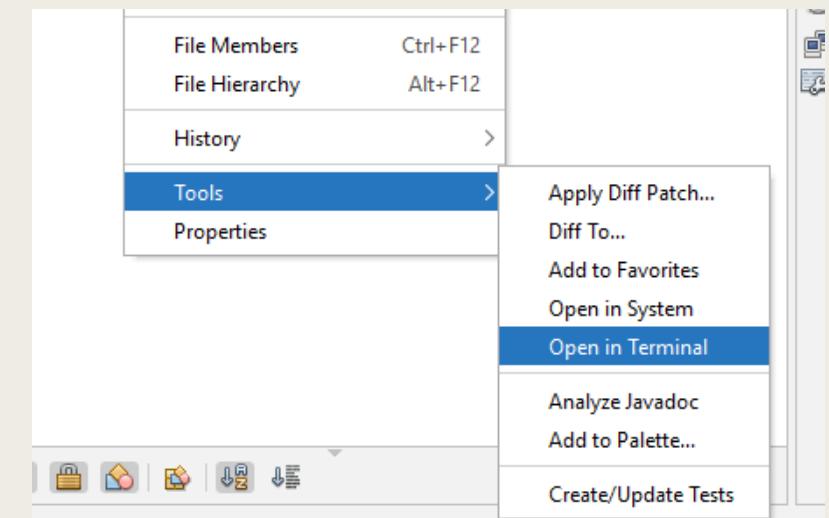
```
public class HiloEjemplo2 extends Thread {  
    //constructor  
    public HiloEjemplo2(String nombre){  
        super(nombre);  
        System.out.println("Creando hilo:" + getName());  
    }  
    //run  
    public void run(){  
        for (int i = 0; i < 3; i++) {  
            System.out.println("Ejecutando hilo: " + getName() + " - contador: " + i);  
        }  
    }  
  
    public static void main(String[] args) {  
        //creacion de 3 hilos  
        HiloEjemplo2 h1 = new HiloEjemplo2("hilo1");  
        HiloEjemplo2 h2 = new HiloEjemplo2("hilo2");  
        HiloEjemplo2 h3 = new HiloEjemplo2("hilo3");  
        h1.start();  
        h2.start();  
        h3.start();  
        System.out.println("Se han iniciado los 3 hilos");  
    }  
}
```

Salidas

- Creando hilo:hilo1
- Creando hilo:hilo2
- Creando hilo:hilo3
- Se han iniciado los 3 hilos
- Ejecutando hilo:hilo1
- Ejecutando hilo:hilo2
- Ejecutando hilo:hilo3
- Creando hilo:hilo1
- Creando hilo:hilo2
- Creando hilo:hilo3
- Se han iniciado los 3 hilos
- Ejecutando hilo: hilo1 - contador: 0
- Ejecutando hilo: hilo2 - contador: 0
- Ejecutando hilo: hilo3 - contador: 0
- Ejecutando hilo: hilo2 - contador: 1
- Ejecutando hilo: hilo1 - contador: 1
- Ejecutando hilo: hilo1 - contador: 2
- Ejecutando hilo: hilo2 - contador: 2
- Ejecutando hilo: hilo3 - contador: 1
- Ejecutando hilo: hilo3 - contador: 2

¿Cómo lo haríamos utilizando 2 clases?

- HiloEjemplo1_2
- UsaHiloEejmplo1_2
- Compila la primera clase desde el terminal
- Ejecuta la segunda desde Netbeans , visualStudio
- Para abrir el terminal en Netbeans – Tools – Open in Terminal
- Para abrir el terminal en Visual Studio Code CTRL + ñ



Programación de subprocesos o programación multihilos

La programación de hilos o subprocesos se basa en qué directiva se debe seguir para decidir qué hilo toma el control del procesador y en qué momento.

También programar cuándo deja de ejecutarse.

Java incluye las siguientes características:

- A todos los hilos o subprocesos se les asigna una prioridad. Esto no implica que en un momento dado se esté ejecutando uno que tenga menor prioridad.
- Se debe garantizar que todos los subprocesos se ejecuten en algún momento.

Programación de Servicios y Procesos

MÉTODOS	MISIÓN
start()	El hilo comienza la ejecución. La máquina virtual de Java llama al método run() de este hilo
boolean isAlive()	Comprueba si el hilo está vivo
sleep(long mils)	El hilo actualmente en ejecución pasa a dormir temporalmente durante el número de milisegundos especificado
run()	Es el cuerpo del hilo. Es llamado por el método start() después de que el hilo 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 cadena incluyendo nombre de hilo, prioridad y grupo de hilos
long getId()	Devuelve el identificador del hilo
void yield()	Hace que el hilo que está en ejecución, pare temporalmente y permita que otros hilos se ejecuten
String getName()	Devuelve el nombre del hilo
setName(String name)	Cambia el nombre del hilo por el que se pasa como argumento

MÉTODOS	MISIÓN
<code>int getPriority()</code>	Devuelve la prioridad del hilo
<code>setPriority(int p)</code>	Cambia la prioridad del hilo (rango entre 1 y 10)
<code>void interrupt()</code>	Interrumpe la ejecución del hilo
<code>boolean isInterrupted()</code>	Comprueba si el hilo actual ha sido interrumpido
<code>Thread currentThread()</code>	Devuelve una referencia al objeto hilo que se está ejecutando actualmente
<code>boolean isDaemon()</code>	Comprueba si el hilo es un hilo Daemon (servicio)
<code>setDaemon(booleano n)</code>	Estable este hilo como hilo Daemon, o como hilo de usuario
<code>void stop()</code>	Detiene el hilo

¿Qué es el hilo Daemon en java?

- Un subprocesso se puede crear como un *demonio* (*daemon*) a través del método `setDaemon(true)`
- Esto permitirá que el subprocesso se ejecute en segundo plano.
- Los subprocessos de Daemon pueden cerrarse en cualquier momento. El subprocesso de usuario se ejecuta completamente.
- Los hilos de Daemon se ejecutan con una prioridad baja.

Ejemplo uso métodos de Thread

Crear una clase HiloMetodos que extienda de Thread y que muestre el nombre del hilo, su prioridad y su identificación.

"Hilo ejecutándose:"

El main creará 3 hilos y les asignará un nombre (hilo1, hilo2, hilo3) y una prioridad y visualizará estas propiedades con el método `toString()`

"Información del HILO1:"

```
public class HiloMetodos extends Thread{  
  
    public void run() {  
        System.out.println("Ejecución del hilo: " + this.getName() + " Prioridad: " + this.getPriority() + " ID: " + this.getId());  
    }  
  
    public static void main(String[] args) {  
  
        HiloMetodos hilo = null;  
  
        for (int i=1; i<=3; i++) {  
            hilo = new HiloMetodos();  
            hilo.setName("hilo"+i);  
            hilo.setPriority(i);  
            hilo.start();  
  
            System.out.println("Información del " + hilo.getName() + ":" + hilo.toString());  
        }  
  
        System.out.println("3 HILOS CREADOS ...");  
    }  
}
```

```
public class HiloEjemplo2 extends Thread {  
  
    //metodo run  
    public void run() {  
        //prueba de metodos de hilos  
        System.out.println("Dentro del hilo " + Thread.currentThread().getName()  
                           + "\n\t Prioridad " + Thread.currentThread().getPriority()  
                           + "\n\t ID " + Thread.currentThread().getId());  
    }  
  
    //metodo main  
    public static void main(String[] args) {  
        //cambiamos el nombre  
        Thread.currentThread().setName(name: "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(); // creamos el hilo  
            h.setName("Hilo " + i); //damos un nombre  
            h.setPriority(i + 1); // damos prioridad  
            h.start(); //iniciamos el hilo  
            System.out.println("Informacion del " + h.getName() + ": " + h.toString());  
        }  
        System.out.println(: "3 hilos creados...");  
        System.out.println("Hilos activos " + Thread.activeCount());  
    }//main  
  
}//HiloEjemplo2
```

```
run:  
Principal  
Thread[Principal ,5,main]  
Informacion del Hilo 0: Thread[Hilo 0,1,main]  
Informacion del Hilo 1: Thread[Hilo 1,2,main]  
Informacion del Hilo 2: Thread[Hilo 2,3,main]  
3 hilos creados...  
Hilos activos 4  
Dentro del hilo Hilo 0  
    Prioridad 1  
        ID 15  
Dentro del hilo Hilo 2  
    Prioridad 3  
        ID 17  
Dentro del hilo Hilo 1  
    Prioridad 2  
        ID 16  
BUILD SUCCESSFUL (total time: 0 seconds)
```

Grupos de hilos

- Todo hilo en ejecución en Jva debe formar parte de un grupo. Por defecto, si no se especifica ningún grupo en el constructor, los hilos serán miembros del grupo main, que es creado por el sistema cuando arranca la aplicación.
- La clase ThreadGroup se utiliza para manejar grupos de hilos en las aplicaciones.
- La clase Thread proporciona constructores en los que se puede especificar el grupo del hilo que se está creando en el mismo momento de instanciarlo.
- Ejemplo de creación de grupo de hilos →
- Thread (grupo ThreadGoup, destino Runnable, nombre String)

```
public class HiloEjemplo2Grupos extends Thread {  
    public void run(){  
        System.out.println("Informacion del hilo:" +  
                           Thread.currentThread().toString());  
        for(int i= 0; i<1000; i++){  
            i++;  
        }  
        System.out.println(Thread.currentThread().getName() + "Finalizando ejecucion");  
    }//run  
    public static void main(String[] args) {  
        Thread.currentThread().setName(name: "Principal");  
        System.out.println(: Thread.currentThread().getName());  
        System.out.println(: Thread.currentThread().toString());  
        //agrupamos  
        ThreadGroup grupo = new ThreadGroup(name: "Grupo de hilos");  
        HiloEjemplo2Grupos h = new HiloEjemplo2Grupos();  
  
        Thread h1 = new Thread(group: grupo,target: h,name: "Hilo 1 ");  
        Thread h2 = new Thread(group: grupo,target: h,name: "Hilo 2 ");  
        Thread h3 = new Thread(group: grupo,target: h,name: "Hilo 3 ");  
  
        //inicio los hilos del grupo  
        h1.start();  
        h2.start();|  
        h3.start();  
  
        System.out.println(: "3 hilos creados");  
        System.out.println("Hilos activos: " + Thread.activeCount());  
    }//main  
}//HiloEjemplo2Grupos
```

Hilos TIC TAC

- Crea dos clases (hilos) que extiendan de thread. Uno de los hilos imprimirá TIC y el otro TAC, para que se pueda ver deberás utilizar la clase sleep() para que duerman un segundo sleep(1000)
- Utiliza try-catch para capturar la excepción de sleep
- Crea una clase que inicie los dos hilos de forma que se vea TIC TAC TIC TAC.....

Ejercicio 1

- Crea una clase que extienda de Thread cuya funcionalidad sea visualizar el mensaje “Hola Mundo”
- Crea un programa Java que visualiza el mensaje anterior 5 veces creando 5 hilos diferentes, usando la clase creada.
- Modifica el mensaje en el hilo para incluir el identificador del hilo.
- Prueba de nuevo el programa.

La interfaz Runnable

Para añadir la funcionalidad de hilo a una clase que deriva de otra clase, se utiliza la interfaz **Runnable**.

Esta interfaz añade la funcionalidad de hilo a una clase con solo implementarla.

La interfaz **Runnable** proporciona un único método, el método *run()*.

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

La interfaz Runnable

```
//Crear el objeto hilo  
NombreHilo h = new NombreHilo()  
  
//Iniciar su ejecución  
new Thread(h).start();  
  
*****  
  
Thread h1 = new Thread(h);  
  
h1.start();  
  
//en un solo paso  
new Thread(new NombreHilo()).start();
```

La interfaz Runnable

Con **Thread** se utiliza la herencia, es decir, no se podrá heredar otros atributos de otras clases, pues **Java** solo admite herencia simple.

Sin embargo, con **Runnable** se pueden implementar muchas interfaces.

Que opción es mejor? Depende de cómo quieras estructurar tu programa. Hay que tener en cuenta que en java no existe la herencia múltiple, por lo que en caso de querer implementar como un hilo una clase que ya herede de otra tendrás que utilizar la interfaz Runnable, en caso contrario podrás utilizar la forma que más te guste.

Ejemplo de Runnable

Vamos a crear una clase **THilo** que herede de **Thread** y otra clase **RHilo** que implemente la interfaz **Runnable**.

En la clase principal crearemos estos dos hilos e iremos visualizando en qué hilo está y un contador.

Programación de Servicios y Procesos

UT2

```
HilosTR.java ✘ THilo.java ✘ RHilo.java
1
2 public class THilo extends Thread{
3
4     private int id;
5
6     public THilo (int id) {
7         this.id=id;
8     }
9     public void run() {
10        for (int i=0; i<100; i++) {
11            System.out.println("[T]Ejecutándose el hilo de id -->" + id + "Contador:" + i);
12        }
13    }
14 }
15

public class RHilo implements Runnable{
16
17     private int id;
18
19     public RHilo(int id) {
20         this.id=id;
21     }
22
23     @Override
24     public void run() {
25
26         for (int i=0; i<100; i++) {
27             System.out.println("[R]Ejecutándose hilo -->" + id + "Contador:" + i);
28         }
29     }
30 }
```

```
public class HilosTR {  
    public static void main(String[] args) {  
        THilo hilo1 = new THilo(1);  
        hilo1.start();  
  
        Thread hilo2 = new Thread(new RHilo(2));  
        hilo2.start();  
  
        for (int i=0; i<100; i++) {  
            System.out.println("Ejecutándose hilo MAIN "+i);  
        }  
    }  
}
```

Estados de un hilo

- **New (Nuevo)**: cuando se crea un objeto hilo con el operador `new()`. En este momento el programa aún no ha comenzado la ejecución del código del método `run()` del hilo.
- **Runnable (Ejecutable)**: cuando se invoca al método `start()`, el hilo pasa a este estado. En este estado el hilo puede estar o no en ejecución, puesto que el sistema operativo tiene que asignar tiempo de CPU al hilo para que se ejecute.
- **Dead (Muerto)**: un hilo muere por varias razones, cuando el método `run()` finaliza con normalidad; y repentinamente debido a alguna excepción no capturada en el método `run()`. Es posible matar a un hilo invocando al método `stop()`, este método está en desuso y no se debe llamar ya que no libera los bloqueos de objetos que ha bloqueado. Más adelante, veremos cómo detener un hilo de forma segura.

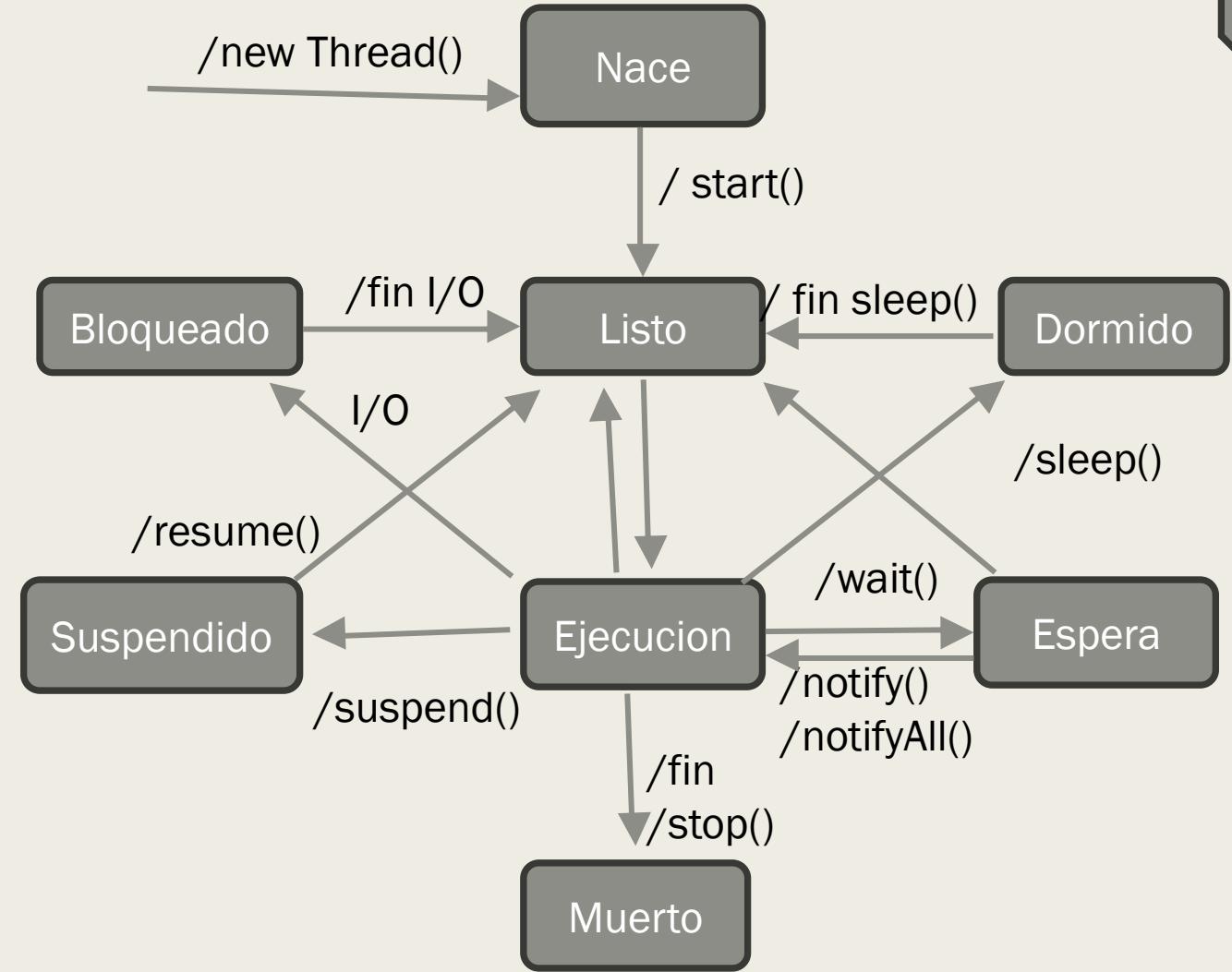
Estados de un hilo

Blocked (Bloqueado): en este estado podría ejecutarse el hilo, pero hay algo que lo impide. Un hilo entra en estado bloqueado cuando ocurre una de las siguientes acciones:

- Se llama al método **sleep()** del hilo.
- El hilo está esperando a que se complete una operación de **entrada/salida**.
- El hilo llama al método **wait()**. No se volverá ejecutable hasta que reciba los mensajes **notify()** o **notifyAll**.
- El hilo intenta bloquear un objeto que ya está bloqueado por otro hilo.
- Alguien llama al método **suspend()** del hilo. No se volverá ejecutable de nuevo hasta que reciba el mensaje **resume()**. Estos dos métodos también están en desuso.

Estados de un hilo

- El método **wait()** hace que el hilo en ejecución espere en estado dormido hasta que se le notifique que continúe.
- El método **notify()** informa a un hilo en espera de que continúe con su ejecución.
- El método **notifyAll()** es similar a **notify()** excepto que se aplica a todos los hilos en espera.
- Estos tres métodos solo pueden ser llamados desde un método o bloque de sincronización



CREAR Y ARRANCAR HILOS

Para crear un hilo extendemos la clase **Thread** o implementamos la interfaz **Runnable**.

```
// Se crea un hilo  
MiHilo h = new MiHilo("Hilo 1", 200);  
  
// Se arranca el hilo  
h.start(); // Si la clase hereda de Thread  
new Thread(h).start(); // Si implementa a Runnable
```

Lo que hace el método **start()** 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.

Actividad-Ejemplo de utilización de los métodos **currentThread()** y **sleep()**

Thread currentThread(): Devuelve una referencia al objeto hilo que se está ejecutando actualmente.

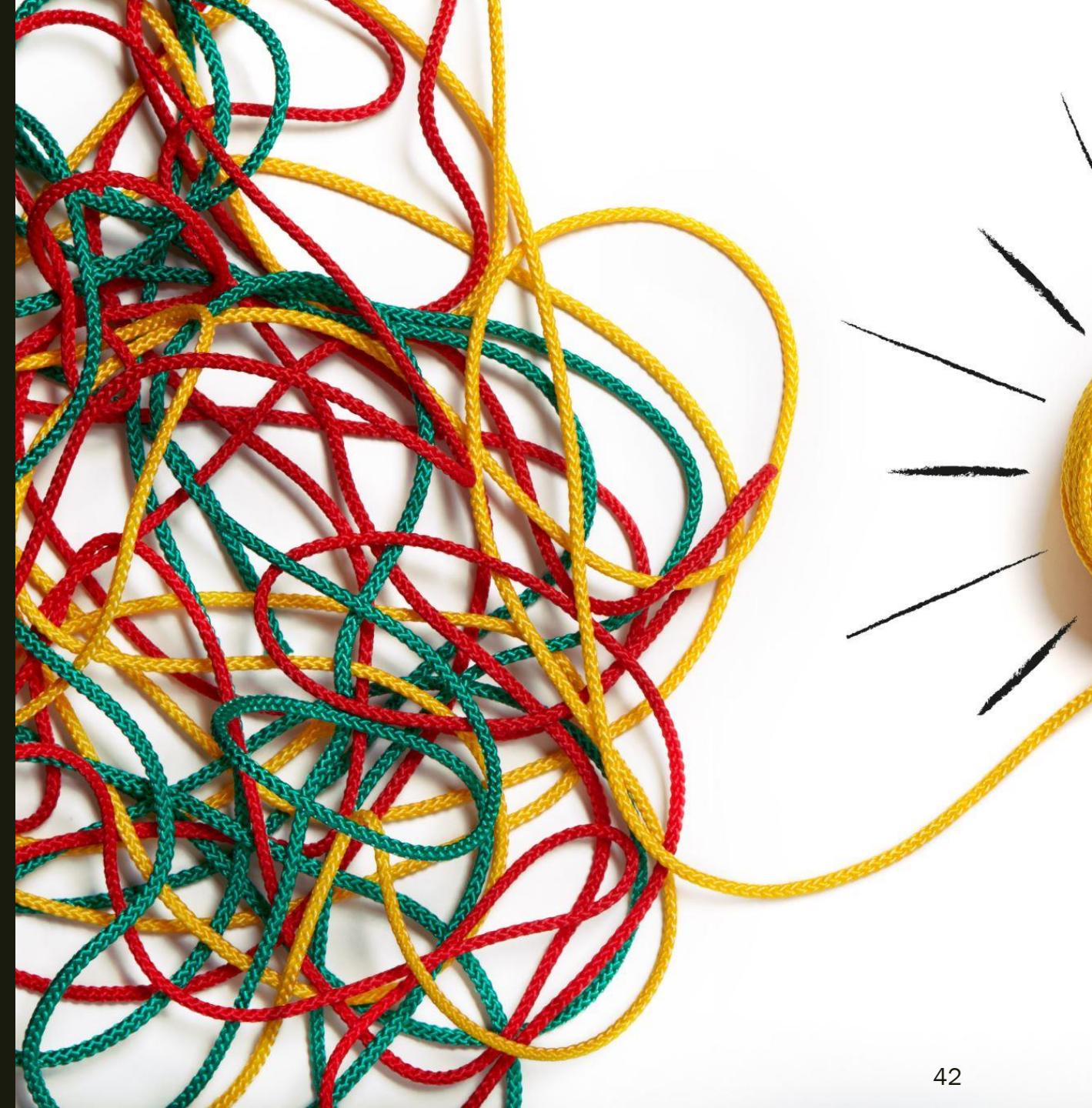
sleep(milisegundos): El hilo que está actualmente en ejecución pasa a dormir temporalmente durante el número de milisegundos especificado.

ACTIVIDAD

Creamos una clase MiHilo que deriva de la clase Thread.

A los hilos que crea el constructor MiHilo, los pondrá a “dormir” con el método **sleep()** durante el tiempo que se le pase como parámetro al constructor. También se pasa como parámetro el nombre del hilo.

Después se imprimirá el nombre del hilo y el tiempo que ha estado dormido.



```
public class MiHilo extends Thread{  
    private int retardo;  
    //constructor  
    public MiHilo(String s, int d) {  
        this.setName(s);  
        retardo = d;  
    }  
    public void run() {  
        try{  
            Thread.currentThread().sleep(retardo);  
        }catch(InterruptedException e){}  
        System.out.println(this.getName() + " tiempo de espera " + retardo + "  
        " + Thread.currentThread());  
    }//run  
}//clase
```

Para probar el hilo, creamos una clase con un método **main()** donde se generarán los argumentos que se le van a pasar al constructor para crear los hilos. El tiempo que va a dormir un hilo se decide aleatoriamente con **Math.random**.

```
public class Prueba {  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub  
        MiHilo hiloA = new MiHilo("hiloA", (int)  
            (Math.random()*2000));  
        MiHilo hiloB = new MiHilo("hiloB", (int)  
            (Math.random()*2000));  
        hiloA.start();  
        hiloB.start();  
        try {  
            Thread.currentThread().sleep(1000);  
        } catch(InterruptedException e) {}  
        System.out.println(Thread.currentThread());  
    } //main  
} //Clase
```

Detener un hilo

Hay dos formas de detener un hilo:

- Detener temporalmente durante un intervalo de tiempo indeterminado: **Suspensión.**
Antes **suspend()**
- Mata definitivamente, impide que el hilo pueda volver a reanudarse: **Interrupción.**
Antes **stop()**

Hoy en día suspend() y stop() están en desuso, en programas complejos pueden generar errores.

Interrumpir un hilo

En los ejemplos que hemos hecho hasta ahora todos los hilos han muerto de forma natural, al finalizar el método run().

Habrá hilos que en su método run contengan un bucle infinito y tendremos que tener una forma de interrumpirlo.

Con **Interrupted()** o **IsInterrupted()** comprobamos si el hilo ha sido interrumpido. Los dos devuelven un booleano.

Interrupted() se refiere al hilo actual y IsInterrupted() hay que poner delante el hilo al que se refiere (hilo1.IsInterrupted())

Si intentamos interrumpirlo cuando el hilo está dormido, no se entera.

Programación de Servicios y Procesos



```
Hilo.java X Principal.java
1 |
2 public class Hilo extends Thread{
3
4     public void run() {
5         while(!interrupted()) {
6             System.out.println("Hola, soy un hilo activo");
7         }
8         System.out.println("El hilo ha muerto");
9     }
10 }
11 }
12

public class Principal {
    public static void main(String[] args) throws InterruptedException {
        Hilo hilo1 = new Hilo();
        // Lo inicializamos
        hilo1.start();
        // Duerme la clase principal
        Thread.sleep(3000);
        // Lo interrumpimos
        hilo1.interrupt();
    }
}
```

Si no ponemos sleep() lo hace tan rápido que ni siquiera llega a escribir "Hola, soy un hilo activo". Para dejar al hilo que durante unos segundos esté activo y reproduzca la frase, vamos a forzarlo con Thread.sleep() para meter un intervalo de tiempo, normalmente, entre start() e interrupt() habría código para que el hilo se viera activo.

Esto puede producir excepciones que tenemos que atraparlas con un try-catch o con un throws declaration

Vamos a ver ahora, que si dentro del run() metemos un sleep y tratamos de interrumpir el hilo cuando está dormido no hace caso de ningún aviso.

Dentro del bucle vamos a dormir al hilo 5 segundos (a los 3 segundos interrumpimos el hilo, pero como está dormido no se da cuenta, cuando pasan los 5 segundos se despierta y sigue la ejecución)

```
Thread.sleep(5000);
```

Al ejecutarlo comprobamos que al interrumpir el hilo como está dormido no ha podido interrumpirse y lanza una excepción.

```
*Hilo.java  X Principal.java
1
2 public class Hilo extends Thread{
3
4     public void run() {
5         while(!interrupted()) {
6
7             try {
8                 Thread.sleep(5000);
9                 System.out.println("Hola, soy un hilo activo");
10            } catch (InterruptedException e) {
11
12                e.printStackTrace();
13            }
14        }
15        System.out.println("El hilo ha muerto");
16    }
17 }
```

SUSPENSIÓN DE UN HILO

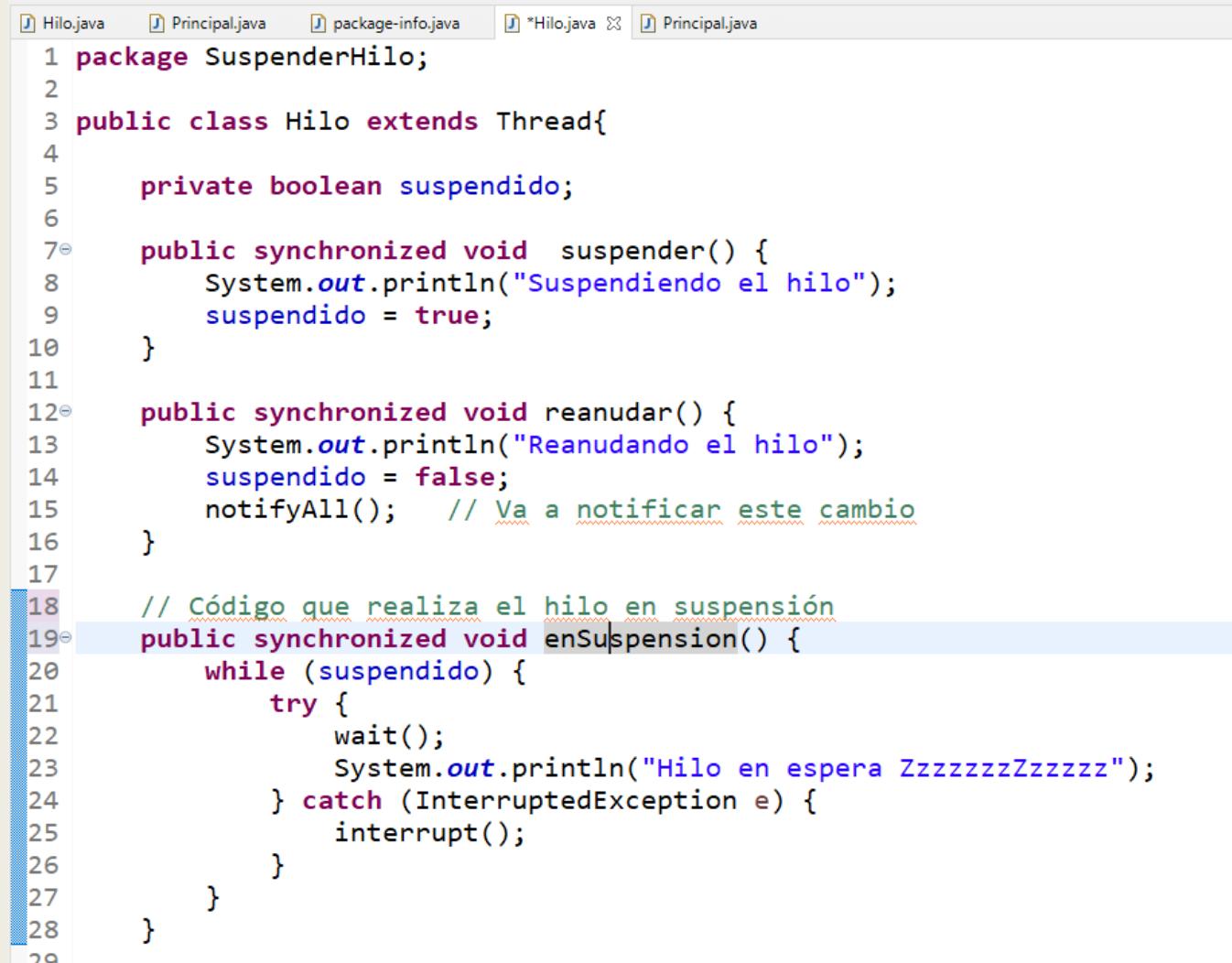
Con el método `sleep()` el hilo no se detiene, sino que se queda dormido el número de milisegundos que le indiquemos.

El método **suspend()** permite detener la actividad del hilo durante un intervalo de tiempo indeterminado. 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. El método `resume()` también está en desuso.

Para suspender de forma segura el hilo se realiza por medio de una variable y se comprueba su valor dentro del método run().

Necesitamos una variable booleana y tres métodos.



The screenshot shows a Java code editor with several tabs at the top: Hilo.java, Principal.java, package-info.java, *Hilo.java, and Principal.java. The code in Hilo.java is as follows:

```
1 package SuspenderHilo;
2
3 public class Hilo extends Thread{
4
5     private boolean suspendido;
6
7     public synchronized void suspender() {
8         System.out.println("Suspensionando el hilo");
9         suspendido = true;
10    }
11
12    public synchronized void reanudar() {
13        System.out.println("Reanudando el hilo");
14        suspendido = false;
15        notifyAll(); // Va a notificar este cambio
16    }
17
18    // Código que realiza el hilo en suspensión
19    public synchronized void enSuspension() {
20        while (suspendido) {
21            try {
22                wait();
23                System.out.println("Hilo en espera ZzzzzzZzzzz");
24            } catch (InterruptedException e) {
25                interrupt();
26            }
27        }
28    }
29}
```

The line `19 public synchronized void enSuspension()` is highlighted with a light blue background.

```
public void run() {  
    while (!isInterrupted()) {  
  
        enSuspension();  
        try {  
            sleep(500);  
        } catch (InterruptedException e) {  
            interrupt();  
        }  
        System.out.println("Hola soy un hilo activo");  
  
    }  
    System.out.println("El hilo ha muerto");  
}
```

The screenshot shows a Java code editor with the following code:

```
1 package SuspenderHilo;
2
3 public class Principal {
4
5
6    public static void main(String[] args) throws InterruptedException {
7
8        Hilo hilo1 = new Hilo();
9        hilo1.start();
10
11        Thread.sleep(2000);
12
13        hilo1.suspender();
14
15        Thread.sleep(2000);
16
17        hilo1.reanudar();
18
19        hilo1.interrupt();
20
21    }
22}
```

The code demonstrates the use of the `suspender()` and `reanudar()` methods on a `Hilo` object. It starts by creating a `Hilo` object and starting it. After a two-second sleep, it calls `suspender()`, causing the thread to be suspended. After another two-second sleep, it calls `reanudar()`, resuming the thread. Finally, it calls `interrupt()` to stop the thread.

- El método **wait()** hace que el hilo espere hasta que le llegue un **notify()** o un **notifyAll()**. Solo se puede llamar desde dentro de un método sincronizado (synchronized). Estos tres métodos se usan en sincronización de hilos y forman parte de la clase **Object** y no de **Thread** como es el caso de **sleep()**, **suspend()** y **resume()**.
- **Sincronizar métodos** nos permite prevenir inconsistencias cuando un objeto es accesible desde distintos hilos.
- Por ejemplo, si un hilo incrementa el contador, mientras otro lo decrementa, y otro lo lee, puede que el resultado no sea el esperado si hay cierta concurrencia.
- Para eso las lecturas y escrituras de un objeto compartido, se hacen a través de **métodos sincronizados**.

Método join()

- El método `join()` provoca que el hilo que hace la llamada espera la finalización de otros hilos.
- Por ejemplo, si en el hilo actual escribo `h1.join()`, el hilo actual se queda en espera hasta que muera el hilo `h1`.
- ¿Para qué sirve el método `join()`? Por ejemplo, si cada hilo está tomando unas medidas y luego hay que calcular una media, necesitamos que acaben para realizar lo siguiente.

Ejemplo join()

- Crearemos un hilo que extienda de Thread donde se le pasará como parámetros el nombre y un número (límite del contador). El hilo sacará por pantalla tantas veces como el límite del contador, un mensaje con su nombre y el contador.
- Por otro lado, tendremos el programa principal que hace uso de este hilo HiloJoin, el cual creará tres objetos de HiloJoin pasando como parámetros el nombre del hilo y el límite del contador. Se inicializarán y posteriormente se utilizará el método join aplicado a cada hilo creado. Después de esto imprimirá por pantalla el mensaje "Final del programa".

The screenshot shows a Java code editor with two tabs: "HiloJoin.java" and "Principal.java". The "HiloJoin.java" tab is active, displaying the following code:

```
1 package HilosJoin;
2
3 public class HiloJoin extends Thread{
4
5     private String nombre;
6     private int limite;
7
8     public HiloJoin(String nom, int n) {
9         this.nombre=nom;
10        this.limite=n;
11    }
12
13    public void run() {
14        for (int i=1; i<=limite; i++) {
15            System.out.println(nombre + ": " + i);
16        }
17        System.out.println("Fin Bucle " + nombre);
18    }
19}
```

Programación de Servicios y Procesos

```
1 package HilosJoin;
2
3 public class Principal {
4
5     public static void main(String[] args) {
6
7         HiloJoin h1 = new HiloJoin("hilo 1",3);
8         HiloJoin h2 = new HiloJoin("hilo 2",5);
9         HiloJoin h3 = new HiloJoin("hilo 3",7);
10
11         h1.start();
12         h2.start();
13         h3.start();
14
15         try {
16             h1.join();
17             h2.join();
18             h3.join();
19         } catch (InterruptedException e) {
20             e.printStackTrace();
21         }
22
23         System.out.println("FIN PROGRAMA");
24     }
25 }
```

RESULTADO DE LA EJECUCIÓN:

```
hilo 1: 1
hilo 3: 1
hilo 3: 2
hilo 3: 3
hilo 3: 4
hilo 3: 5
hilo 3: 6
hilo 3: 7
Fin Bucle hilo 3
hilo 2: 1
hilo 2: 2
hilo 2: 3
hilo 2: 4
hilo 2: 5
Fin Bucle hilo 2
hilo 1: 2
hilo 1: 3
Fin Bucle hilo 1
FIN PROGRAMA
```

Conclusiones ejemplo join()

- Observamos que hasta que no han terminado, muerto, los tres hilos no se imprime el mensaje "final de programa".
- Si quitamos las instrucciones de join, nos damos cuenta que esta frase aparece antes de que mueran los tres hilos.
- Este método sirve para esperar a que acabe de ejecutarse algún hilo para poder realizar lo siguiente.

Actividad join

- Crear un hilo que implemente la interfaz Runnable.
- Se generará un nº aleatorio del 1 al 10 y mientras éste no sea 2, se escribe en pantalla el nombre del hilo actual y el nº aleatorio.
- Al final de su ejecución, imprime FIN y el nombre del hilo actual.

Actividad join

- Por otro lado, un programa hará uso de este hilo, creará dos hilos, los dará un nombre con setName, los inicializará y finalmente implementaremos el join a estos hilos antes de escribir por pantalla el último mensaje "Fin programa".

```
1 package JoinRunnable;
2
3 public class HiloJoinRun implements Runnable{
4
5     @Override
6     public void run() {
7
8         int aleatorio;      // Número aleatorio de 1 a 10
9
10        while ((aleatorio=(int)(Math.random() * 10) + 1)!=2) {
11            // Imprime el hilo actual y un número aleatorio hasta que éste sea 2
12            System.out.println(Thread.currentThread().getName() + "[" + aleatorio + "]");
13        }
14        try {
15            Thread.sleep(1000);
16        } catch (InterruptedException e) {
17            e.printStackTrace();
18        }
19        System.out.println(Thread.currentThread().getName() + "[--FIN--]");
20    }
21 }
```

```
1 package JoinRunnable;
2
3 public class Principal {
4
5     public static void main(String[] args) throws InterruptedException {
6
7         Thread hilo1 = new Thread(new HiloJoinRun());
8         Thread hilo2 = new Thread(new HiloJoinRun());
9
10        hilo1.setName("Hilo-A");
11        hilo2.setName("Hilo-B");
12
13        hilo1.start();
14        hilo2.start();
15
16        hilo1.join();
17        //hilo2.join();
18
19        System.out.println("FIN PROGRAMA");
20    }
21 }
```

Prioridad en los hilos

- En el lenguaje de programación Java, cada hilo tiene una prioridad.
- Por defecto un hilo hereda la prioridad del hilo padre que lo crea.
- Esta se puede aumentar o disminuir mediante el método **setPriority()**
- El método **getPriority()** devuelve la prioridad del hilo.
- La prioridad es un valor entero de 1 a 10
 - MIN_PRIORITY 1
 - MAX_PRIORITY 10
 - NORM_PRIORITY 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 la prioridad de forma cíclica (round-robin).

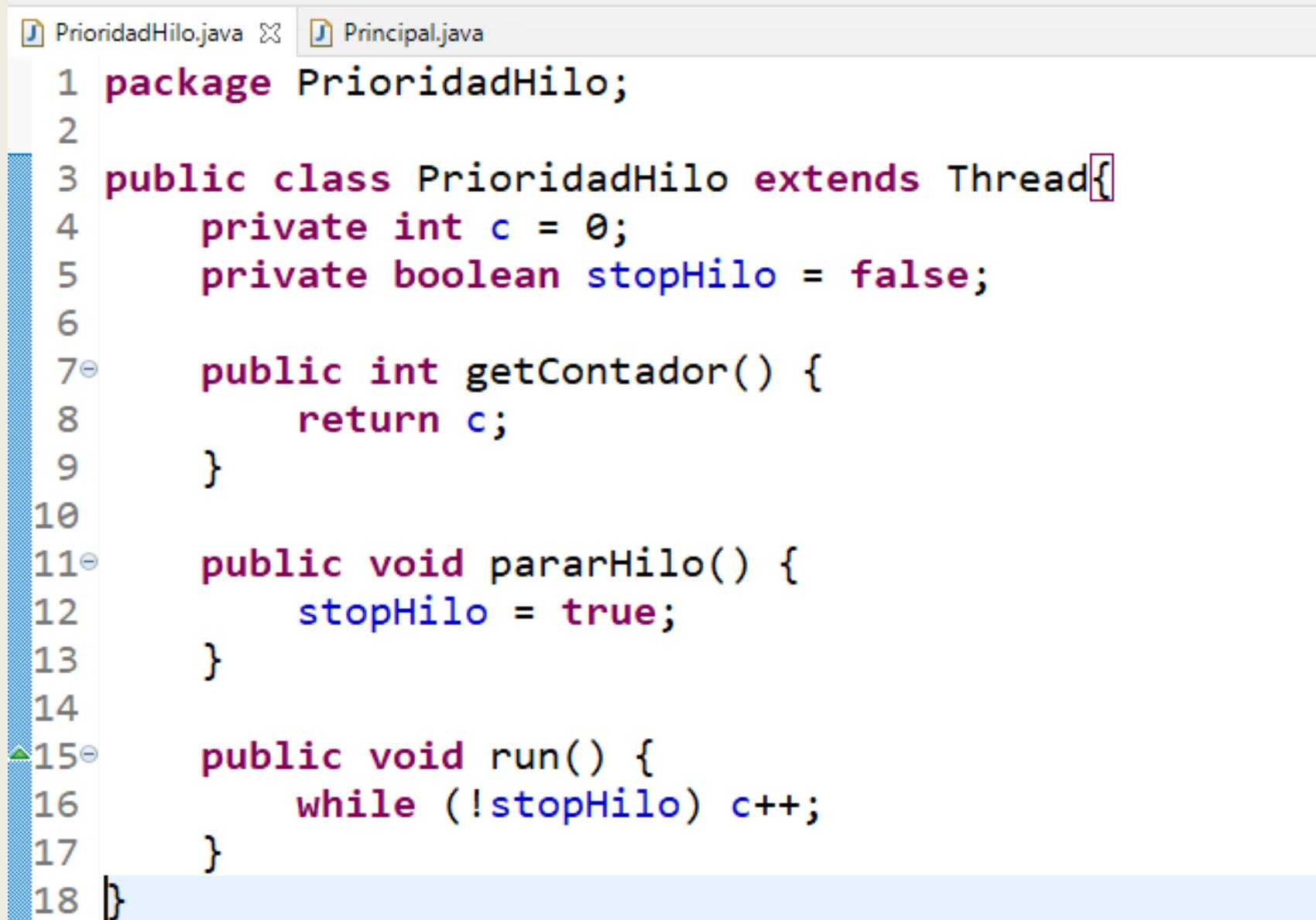
Prioridad en los hilos

El hilo de mayor prioridad sigue funcionando hasta que:

- Cede el control llamando al método `yield()`. El uso del método `yield()` hace que el hilo que está en ejecución, pare temporalmente y permita que otros hilos se ejecuten, es decir, devuelve automáticamente el control al planificador. Sin este método el mecanismo de multihilos sigue funcionando aunque más lentamente.
- Deja de ser ejecutable, sea por muerte o por entrar en estado de bloqueo o suspensión.
- 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()` o `notify()`).

Ejemplo Prioridad Hilos

- Definiremos un Hilo que extiende Thread, se definirá una variable contador que será incrementada en el método run() y una variable boolean stopHilo. Definiremos un método para obtener el valor de la variable, **getContador()** y otro método para finalizar el hilo **pararHilo()**.
- Se definirá la clase principal que hará uso de esta clase hilo creando en el main() 3 objetos hilo. A cada hilo se le asigna una prioridad. Se espera un intervalo de tiempo antes de parar los hilos. Al finalizar cada hilo se muestran los valores del contador invocando al método getContador() del hilo.



The screenshot shows a Java code editor with two tabs: "PrioridadHilo.java" and "Principal.java". The "PrioridadHilo.java" tab is active, displaying the following code:

```
1 package PrioridadHilo;
2
3 public class PrioridadHilo extends Thread{
4     private int c = 0;
5     private boolean stopHilo = false;
6
7     public int getContador() {
8         return c;
9     }
10
11    public void pararHilo() {
12        stopHilo = true;
13    }
14
15    public void run() {
16        while (!stopHilo) c++;
17    }
18 }
```

Programación de Servicios y Procesos

UT2

```
PrioridadHilo.java  Principal.java ✘
1 package PrioridadHilo;
2
3 public class Principal {
4
5     public static void main(String[] args) {
6
7         // Creamos tres hilos
8         PrioridadHilo hilo1 = new PrioridadHilo();
9         PrioridadHilo hilo2 = new PrioridadHilo();
10        PrioridadHilo hilo3 = new PrioridadHilo();
11
12        // Asignamos una prioridad a cada hilo
13        hilo1.setPriority(Thread.NORM_PRIORITY);
14        hilo2.setPriority(Thread.MAX_PRIORITY);
15        hilo3.setPriority(Thread.MIN_PRIORITY);
16
17        // Inicializamos los hilos
18        hilo1.start();
19        hilo2.start();
20        hilo3.start();
21
22        // Esperamos un intervalo de tiempo
23        try {
24            Thread.sleep(2000);
25        } catch (InterruptedException e) {
26            e.printStackTrace();
27        }
28
29        // Parar hilos
30        hilo1.pararHilo();
31        hilo2.pararHilo();
32        hilo3.pararHilo();
33
34        // Escribir en pantalla hilo y contador
35        System.out.println("hilo2 (Prioridad Máxima): " + hilo2.getContador());
36        System.out.println("hilo1 (Prioridad Normal): " + hilo1.getContador());
37        System.out.println("hilo3 (Prioridad Mínima): " + hilo3.getContador());
38
39    }
40 }
```

Conclusiones

- Se puede observar que el máximo valor del contador lo tiene el hilo con prioridad máxima y el mínimo el de prioridad mínima.
- En Windows al ejecutarlo varias veces obtendremos que casi siempre los valores del contador dependerá de la prioridad asignada al hilo.
- 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 tiempo (time-slicing o tiempo compartido). 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.
- Los subprocessos están programados para ejecutarse según su prioridad. Aunque los subprocessos se ejecutan durante el tiempo de ejecución, el sistema operativo asigna intervalos de tiempo de procesador a todos los subprocessos. Los detalles del algoritmo de programación utilizado para determinar el orden en que se ejecutan los subprocessos varían en función de cada sistema operativo. En algunos sistemas operativos, el subprocesso con la prioridad más alta (de entre los subprocessos que se pueden ejecutar) está programado siempre para que se ejecute primero. Si hay disponibles varios subprocessos con la misma prioridad, el programador recorre los subprocessos con dicha prioridad, dando a cada subprocesso un intervalo de tiempo fijo para su ejecución.
- En la práctica casi nunca hay que establecer a mano las prioridades.

Ejemplo prioridad

Cualquier aplicación de reproductor de video debería estar optando por la función "Programación de clases multimedia" en Windows. Esto le dará automáticamente hasta el 80% de una CPU. Podríamos querer bajar la prioridad para que el resto de procesos que se ejecutan mientras tanto vayan más rápido.

SINCRONIZACIÓN DE HILOS

- Los hilos se comunican unos con otros, esta comunicación muchas veces consiste en compartir objetos. Ponemos el ejemplo de un método donde una variable se incrementa, decrementa. Si varios hilos acceden a la vez puede darse inconsistencia, es decir, que un hilo obtenga un valor de la variable que no es correcto.
- Los métodos sincronizados (`synchronized`) garantizan que solo un hilo pueda acceder a ese método a la vez. El resto de los hilos se quedarán bloqueados hasta que finalice.
- Es fácil de implementar aunque resulta costoso en tiempo de ejecución, ya que los demás hilos tienen que esperar hasta que el hilo salga del método. Por lo cual, tan solo hay que utilizarlo cuando sea necesario.

Ejemplo de sincronización

- Cuenta bancaria compartida por dos usuarios que van a realizar operaciones de retirar dinero.
- Partimos de un saldo inicial en la cuenta. Ambos usuarios podrán ir retirando dinero de la cuenta mientras el saldo sea positivo.
- Cuando un usuario esté accediendo a la cuenta para retirar dinero, hemos de impedir que el otro usuario acceda también, pues podríamos encontrarnos con saldos negativos. Para evitar los saldos negativos añadiremos la palabra synchronized al método donde se realiza la operación.

Crearemos tres clases:

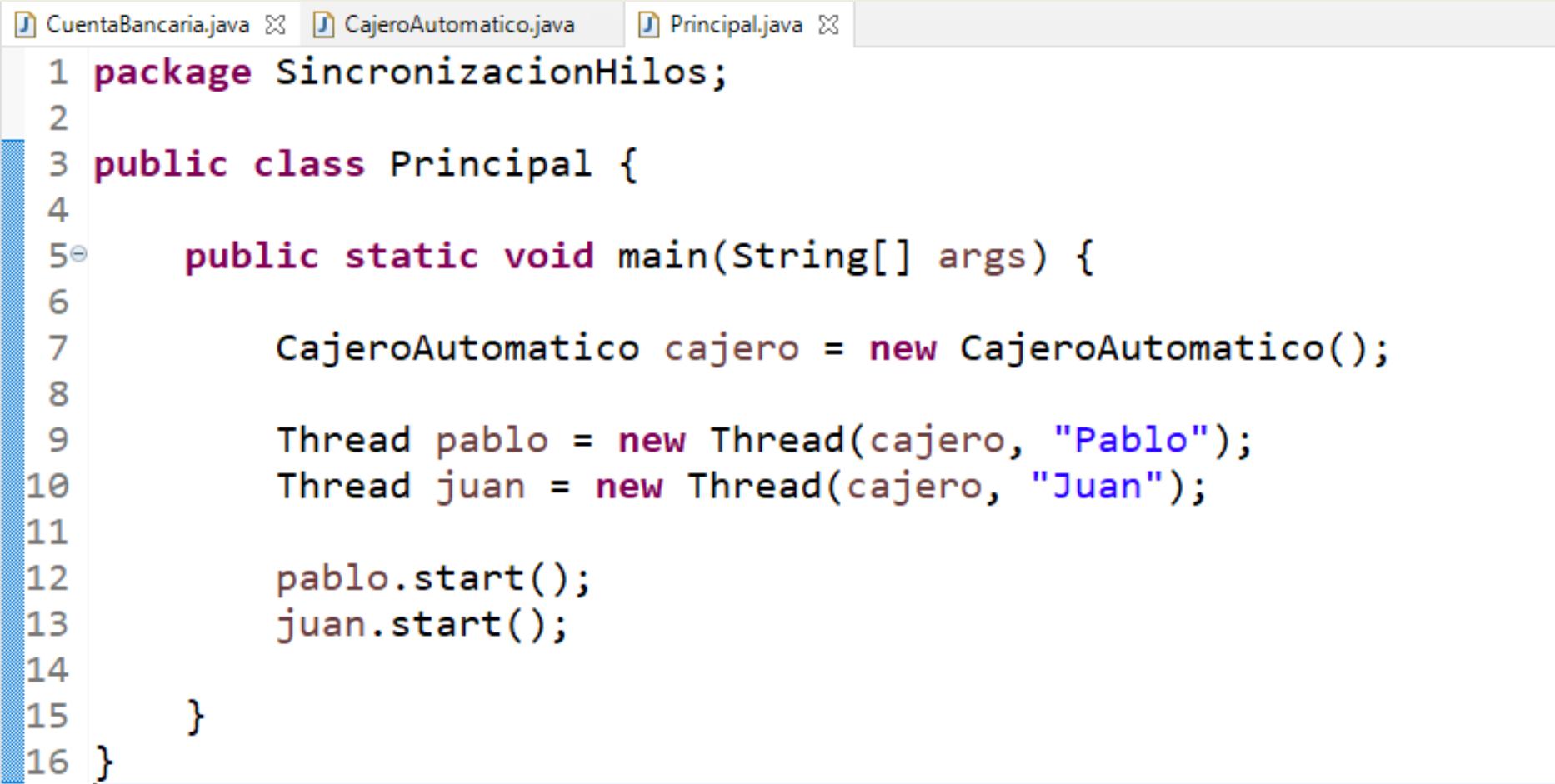
- **CuentaBancaria**: con un saldo inicial, un método *getSaldoActual* que nos indica el saldo que hay en la cuenta y otro método *retirarDinero* donde se irá descontando de la cuenta el dinero retirado.
- **CajeroAutomatico**: donde se van a realizar las transacciones a nuestra cuenta bancaria. Va a ser un hilo que implementa la interfaz **Runnable**.
- La clase **Principal** donde se hace uso del hilo creado (del cajero), se crearán dos objetos hilos (dos usuarios que hacen uso del cajero para sacar dinero)

Si los dos usuarios acceden a la vez para retirar dinero, es decir, entran en el método **RetirarDineroCuenta** los dos porque no existe sincronización, se puede dar el caso que no haya saldo suficiente y la cuenta se quede con saldo negativo. Sin embargo, si este método está sincronizado, solo puede acceder un hilo cada vez (solo un usuario puede retirar dinero en un momento dado), con lo cual la cuenta no llega a estar en negativo.

```
1 package SincronizacionHilos;
2
3 public class CuentaBancaria {
4
5     private int SaldoActual=100;
6
7     public int getSaldoActual() {
8         return SaldoActual;
9     }
10
11    public void retirarDinero(int CantidadRetirar) {
12
13        SaldoActual -= CantidadRetirar;
14    }
15 }
```

Programación de Servicios y Procesos

```
1 package SincronizacionHilos;
2
3 public class CajeroAutomatico implements Runnable {
4
5     CuentaBancaria cuenta = new CuentaBancaria();
6
7     @Override
8     public void run() {
9         for (int i=0; i<5; i++) {
10             RetirarDineroCuenta(20);
11             if (cuenta.getSaldoActual()<0) {
12                 System.out.println("Cuenta con saldo negativo");
13             }
14         }
15     }
16
17     private synchronized void RetirarDineroCuenta(int cantidad) {
18         // Llama a la cuenta bancaria para retirar el dinero
19         if (cuenta.getSaldoActual()>=cantidad) {
20             System.out.println();
21             System.out.println("Saldo actual: "+cuenta.getSaldoActual());
22             System.out.println("El usuario "+Thread.currentThread().getName()+
23                 " está retirando dinero por un valor de "+cantidad+" €");
24             // Descontar dinero
25             cuenta.retirarDinero(cantidad);
26             System.out.println("Transacción realizada con éxito!!!!"
27                 + " El nuevo saldo de la cuenta es: " + cuenta.getSaldoActual());
28         }
29         else {
30             System.out.println("Señor/a " + Thread.currentThread().getName() +
31                 ", no hay saldo suficiente en su cuenta");
32         }
33     }
34 }
```



```
1 package SincronizacionHilos;
2
3 public class Principal {
4
5     public static void main(String[] args) {
6
7         CajeroAutomatico cajero = new CajeroAutomatico();
8
9         Thread pablo = new Thread(cajero, "Pablo");
10        Thread juan = new Thread(cajero, "Juan");
11
12        pablo.start();
13        juan.start();
14
15    }
16}
```

El modelo productor-consumidor

Un problema típico de sincronización en la programación concurrente es el que representa el modelo **Productor-Consumidor**. 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 consumiendo o viceversa.

Ejemplo: Una aplicación donde un hilo escribe datos (productor) en un fichero y un segundo hilo lee los datos (consumidor) de ese fichero. Los hilos comparten un mismo recurso (el fichero) y deben sincronizarse para realizar su tarea correctamente.

Con `wait()` y `notifyAll()` implementaremos el bloqueo y desbloqueo de acceso a un recurso.

Ejemplo Productor-Consumidor

El productor va a ir poniendo números en el recurso compartido y el consumidor los va a ir cogiendo.

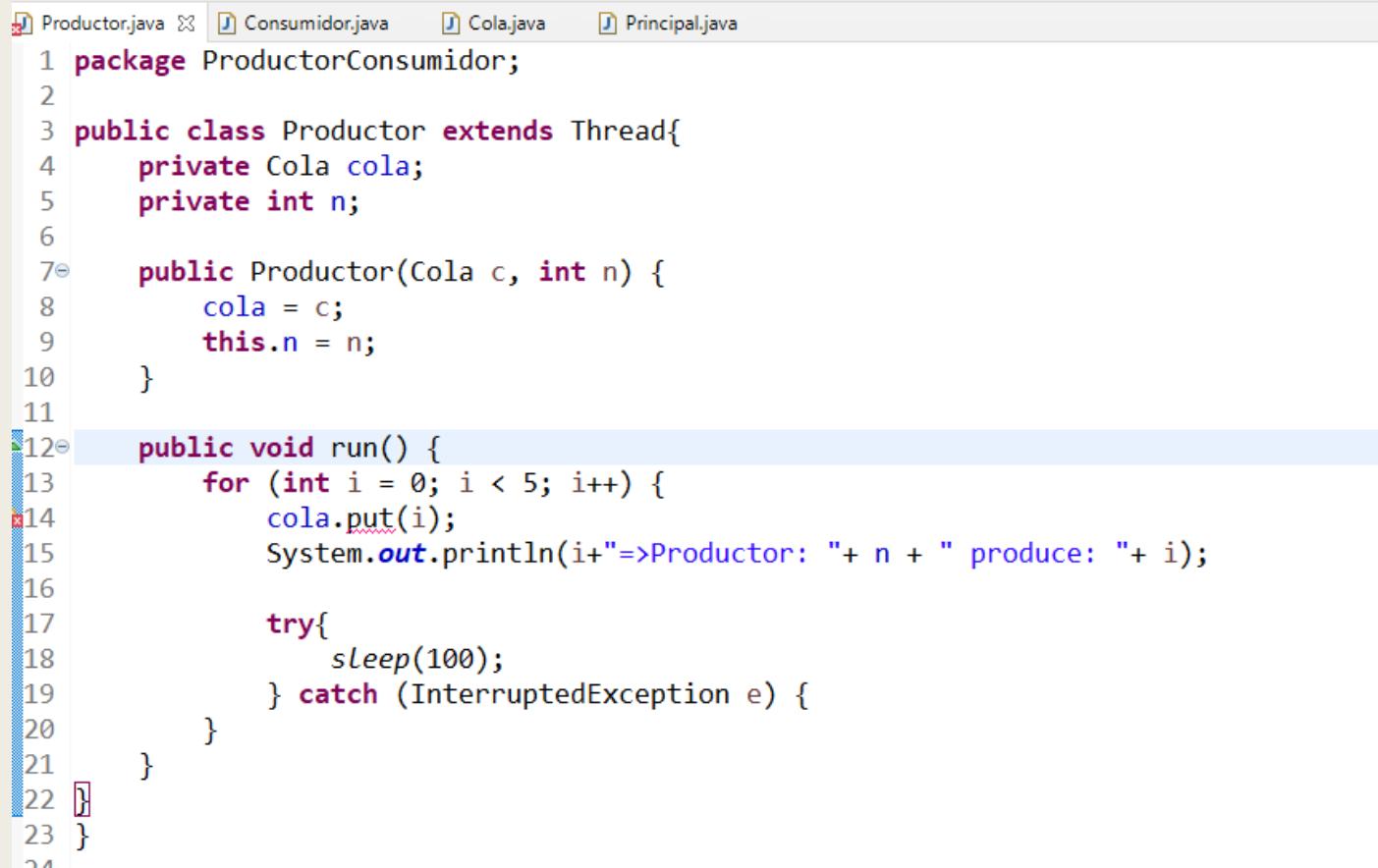
Se definen 3 clases, la clase *Cola* que será el objeto compartido entre el productor y consumidor.

La clase *Productor* que produce números y los va colocando en la Cola para que sean consumidos por el Consumidor.

La clase *Consumidor* recoge los números de la Cola.

Clase Productor

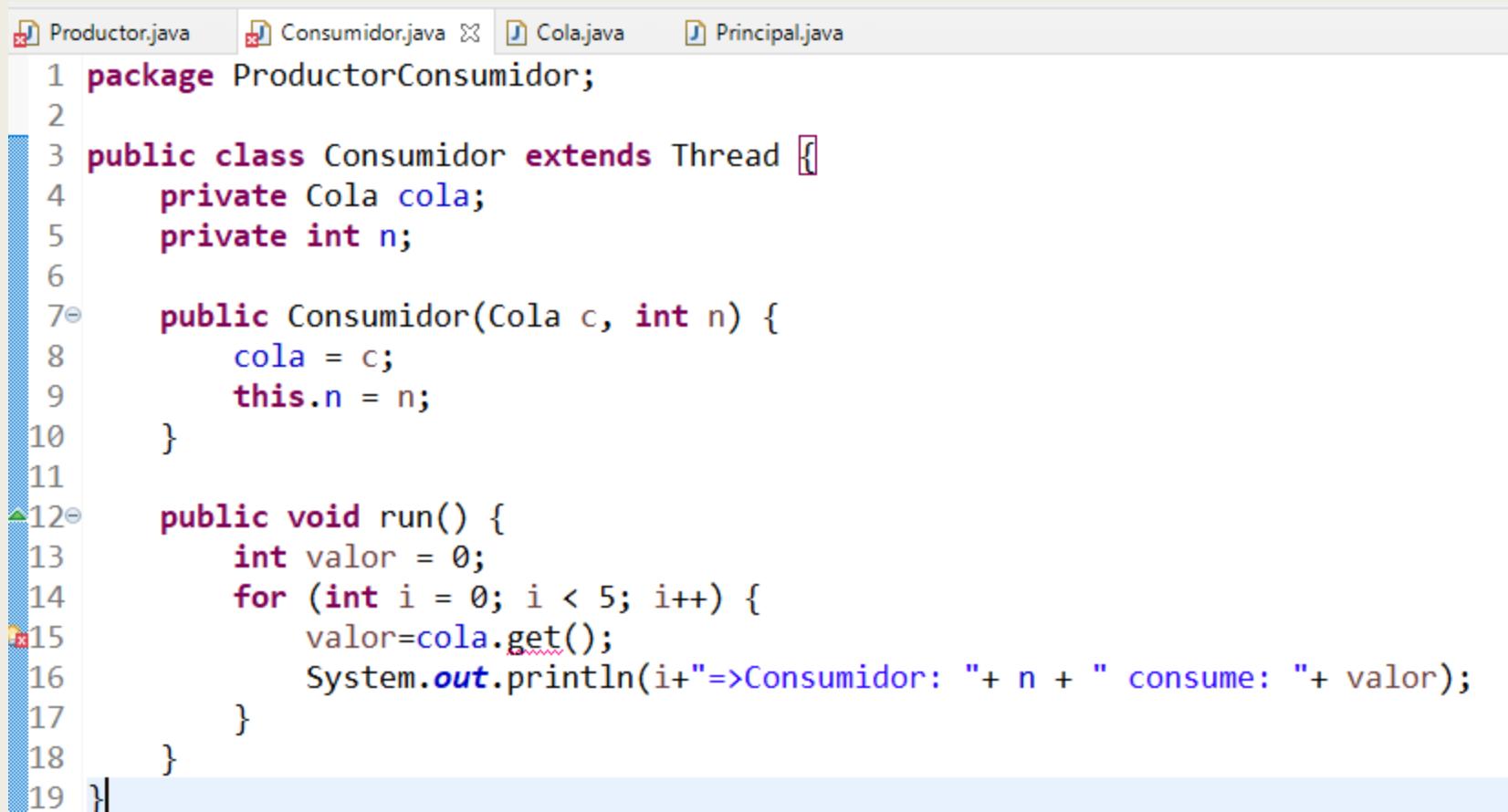
El productor genera números de 0 a 5 en un bucle for y los pone en el objeto Cola mediante el método put(), después se visualiza y se hace una pausa con sleep() con ello forzamos que el productor sea más lento que el consumidor.



```
1 package ProductorConsumidor;
2
3 public class Productor extends Thread{
4     private Cola cola;
5     private int n;
6
7     public Productor(Cola c, int n) {
8         cola = c;
9         this.n = n;
10    }
11
12    public void run() {
13        for (int i = 0; i < 5; i++) {
14            cola.put(i);
15            System.out.println(i+"=>Productor: "+ n + " produce: "+ i);
16
17            try{
18                sleep(100);
19            } catch (InterruptedException e) {
20            }
21        }
22    }
23 }
```

Clase Consumidor

Clase similar al Productor, salvo que en vez de poner un número en la cola lo coge llamando al método get(). Aquí no ponemos ninguna pausa, haciendo que el consumidor vaya más rápido que el productor.



The screenshot shows a Java IDE interface with four tabs at the top: 'Productor.java', 'Consumidor.java' (which is currently selected), 'Cola.java', and 'Principal.java'. The code for the 'Consumidor.java' class is displayed in the main editor area:

```
1 package ProductorConsumidor;
2
3 public class Consumidor extends Thread {
4     private Cola cola;
5     private int n;
6
7     public Consumidor(Cola c, int n) {
8         cola = c;
9         this.n = n;
10    }
11
12    public void run() {
13        int valor = 0;
14        for (int i = 0; i < 5; i++) {
15            valor=cola.get();
16            System.out.println(i+">Consumidor: "+ n + " consume: "+ valor);
17        }
18    }
19 }
```

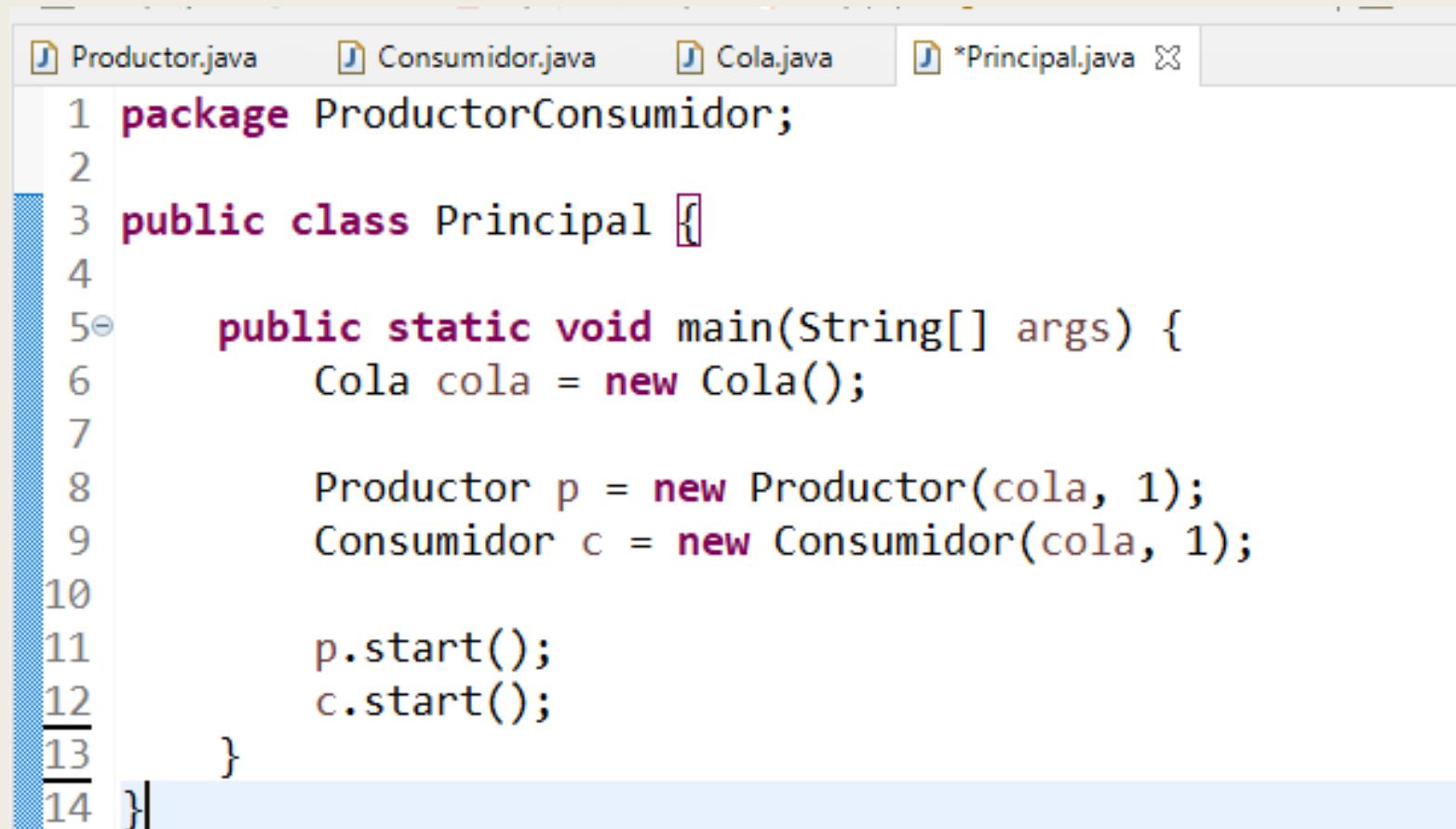
Clase Cola

La clase Cola tiene dos atributos y dos métodos. El atributo número donde se guarda el entero y el atributo disponible que indica si hay o no un número en la cola. El método put guarda un entero en el atributo número y pone el atributo disponible a true (cola llena). El método get devuelve el entero de la cola si existe.

```
1 package ProductorConsumidor;
2
3 public class Cola {
4     private int numero;
5     private boolean disponible = false; // inicialmente la cola está vacía
6
7     public int get() {
8         if (disponible) { // Si disponible es true hay un número en la cola para coger
9             disponible = false;
10            return numero;
11        }
12        return -1; // Devuelve -1 si no hay número disponible
13    }
14
15    public void put(int valor) {
16        numero = valor; //coloca el valor en la cola
17        disponible = true; //disponible para consumir, cola llena
18    }
19 }
20
```

Clase Principal

Se crean 3 objetos, un objeto Cola, un objeto Productor y un objeto Consumidor. Al constructor de las clases Productor y Consumidor pasamos el objeto Cola y un número entero que los identifica.



```
1 package ProductorConsumidor;
2
3 public class Principal {
4
5     public static void main(String[] args) {
6         Cola cola = new Cola();
7
8         Productor p = new Productor(cola, 1);
9         Consumidor c = new Consumidor(cola, 1);
10
11         p.start();
12         c.start();
13     }
14 }
```

Al compilar y ejecutar obtenemos lo

El consumidor al ir más rápido pues no le hemos puesto pausa al consumir se encuentra con que no hay número y devuelve -1.

Es necesario modificarlo para que exista una coordinación entre productor y consumidor, de forma que cuando el productor ponga un número en la cola avise al consumidor para que lo recoja y cuando lo recoja avise al productor para que ponga otro.

```
0=>Productor: 1 produce: 0
0=>Consumidor: 1 consume: 0
1=>Consumidor: 1 consume: -1
2=>Consumidor: 1 consume: -1
3=>Consumidor: 1 consume: -1
4=>Consumidor: 1 consume: -1
1=>Productor: 1 produce: 1
2=>Productor: 1 produce: 2
3=>Productor: 1 produce: 3
4=>Productor: 1 produce: 4
```

Sincronizar los hilos

- Declarar los métodos get() y put() como synchronized, de esta manera el consumidor y productor no podrán acceder a la vez al recurso compartido (objeto Cola).
- Para mantener la coordinación entre Productor y Consumidor usamos los métodos wait(), notify() y notifyAll()
*A veces nos interesa que un hilo se quede bloqueado a la espera de que ocurra algún evento, como la llegada de un dato para tratar o que el usuario termine de escribir algo en una interface de usuario. Todos los objetos java tienen el método **wait()** que deja bloqueado al hilo que lo llama y los métodos **notify()**, **notifyAll()** que desbloquea a los hilos bloqueados por **wait()**.*

Sincronizar los hilos

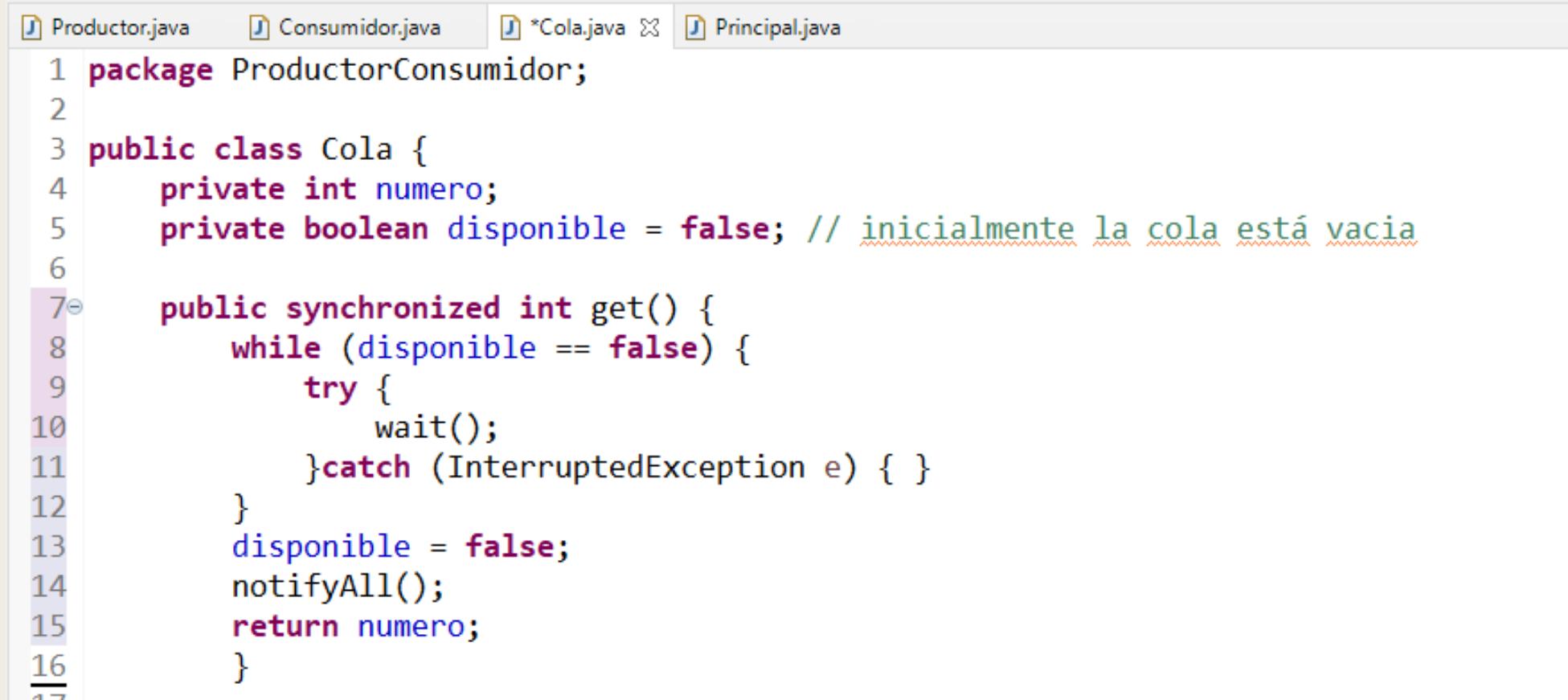
¿Quieres decirle a uno de los hilos de espera que sucedió algo, o quieres contarles a todos al mismo tiempo?

notify(): Cada llamada a **notify ()** despierta al primer hilo en la lista de espera, pero no al resto, que siguen dormidos.

notifyAll(): despierta todos los hilos que están esperando el objeto.

NOTA: Los métodos **notify()**, **notifyAll()** y **wait()** pueden ser invocados solo desde dentro de un método sincronizado o dentro de un bloque o sentencia sincronizada.

Método `get()` sincronizado: tiene que esperar hasta que la cola se llene (es decir, mientras está vacía, espera). Se sale del bucle cuando llega un valor y al consumirlo se pone a false y notifica a todos los hilos que comparten el objeto Cola y se devuelve el valor.



```
1 package ProductorConsumidor;
2
3 public class Cola {
4     private int numero;
5     private boolean disponible = false; // inicialmente la cola está vacía
6
7     public synchronized int get() {
8         while (disponible == false) {
9             try {
10                 wait();
11             }catch (InterruptedException e) { }
12         }
13         disponible = false;
14         notifyAll();
15         return numero;
16     }
17 }
```

Método put() sincronizado: tiene que esperar hasta que la cola se vacie para poner un valor, es decir, espera mientras haya un valor en la cola (disponible==true). Cuando la cola se vacie disponible es false, se sale del bucle, pone un valor y se vuelve a poner disponible a true porque la cola está llena. Y se notifica a todos los hilos que comparten este objeto.

```
public synchronized void put(int valor) {  
    while (disponible == true) {  
        try {  
            wait();  
        }catch (InterruptedException e) { }  
    }  
    numero = valor;  
    disponible = true;  
    notifyAll();  
    ;  
}
```

Si lo ejecutamos ahora con los métodos sincronizados, se obtiene lo siguiente:

```
0=>Productor: 1 produce: 0
0=>Consumidor: 1 consume: 0
1=>Productor: 1 produce: 1
1=>Consumidor: 1 consume: 1
2=>Consumidor: 1 consume: 2
2=>Productor: 1 produce: 2
3=>Consumidor: 1 consume: 3
3=>Productor: 1 produce: 3
4=>Consumidor: 1 consume: 4
4=>Productor: 1 produce: 4
```

ACTIVIDAD

Modifica la clase Productor para que envíe las cadenas PING y PONG (de forma alternativa) a la cola y la clase Consumidor tome la cadena de la cola y la visualice. La salida tiene que mostrar lo siguiente: PING PONG PING PONG PING PONG ...

Productor

```
package ProductorConsumidor;

public class Productor extends Thread{
    private Cola cola;
    private String cadenaSgte=" PING ";

    public Productor(Cola c, String cadena) {
        cola = c;
        this.cadenaSgte = cadena;
    }

    public void run() {
        for (int i = 0; i < 10; i++) {
            cola.put(cadenaSgte);
            if (cadenaSgte==" PING ")
                cadenaSgte=" PONG ";
            else
                cadenaSgte=" PING ";

            try{
                sleep(100);
            } catch (InterruptedException e) {}
        }
    }
}
```

Consumidor

```
1 package ProductorConsumidor;  
2  
3 public class Consumidor extends Thread {  
4     private Cola cola;  
5     private String cadena;  
6  
7     public Consumidor(Cola c, String cadena) {  
8         cola = c;  
9         this.cadena = cadena;  
10    }  
11  
12    public void run() {  
13        String cadena= "";  
14        for (int i = 0; i < 10; i++) {  
15            cadena=cola.get();  
16            System.out.println(cadena);  
17  
18        }  
19    }  
20}
```

Cola

```
package ProductorConsumidor;

public class Cola {
    private String cadena;
    private boolean disponible = false; // inicialmente la cola está vacía

    public synchronized String get() {
        while (disponible == false) {
            try {
                wait();
            }catch (InterruptedException e) { }
        }
        disponible = false;
        notifyAll();
        return cadena;
    }

    public synchronized void put(String valor) {
        while (disponible == true) {
            try {
                wait();
            }catch (InterruptedException e) { }
        }
        cadena = valor;
        disponible = true;
        notifyAll();
    }
}
```

Principal

```
package ProductorConsumidor;

public class Principal {

    public static void main(String[] args) {
        Cola cola = new Cola();

        Productor p = new Productor(cola, "");
        Consumidor c = new Consumidor(cola, "");

        p.start();
        c.start();
    }
}
```