Desarrollo de Aplicaciones Multiplataforma

Informática y Comunicaciones

IES Valle del Jerte
Plasencia
informatica.iesvalledeljerteplasencia.es

PROGRAMACIÓN DE SERVICIOS Y PROCESOS

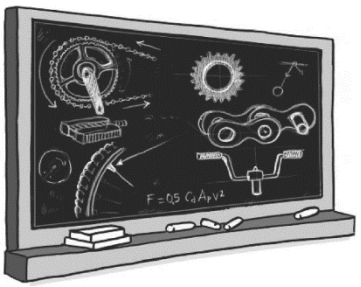
UT1: Hilos. Sincronizaciones en Java

2º DAM

Contenido

1. PROGRAMAS, PROCESOS Y CONCURRENCIA

Los programas son elementos de gestión centrales del sistema operativo. Desde el punto de vista del *núcleo* del sistema operativo tienen dos partes bien definidas: la *imagen de memoria* y las *tablas de control de procesos*.



La *imagen de memoria* es la memoria física ocupada por el código y datos del programa. Se diferencian cuatro partes según su contenido:

- **Código o texto**: El programa ejecutable cargado en memoria.
- **Datos**: La zona de memoria donde se almacenan las constantes y variables estáticas del programa.
- **Pila *(stack):*** La zona donde se almacenan los argumentos de las funciones, sus valores de retorno y las variables automáticas (locales).
- **Zona de memoria dinámica *(heap + malloc):*** La zona de almacenamiento de memoria asignada dinámicamente.

Las *tablas de control de procesos* son estructuras variadas y complejas con información de estado de cada proceso. Por ejemplo, los valores de los registros del procesador cuando el proceso fue interrumpido, tablas de páginas, de entrada-salida, información de propiedad, estadísticas de uso, etc.

Podríamos definir un **programa** como un conjunto de instrucciones. Consiste en una secuencia de líneas de código que dicen qué hacer con un conjunto de datos de entrada para producir algún tipo de salida. Se trata de algo estático. Puede compararse con el concepto de *clase* en el ámbito de la Programación Orientada a Objetos (POO).

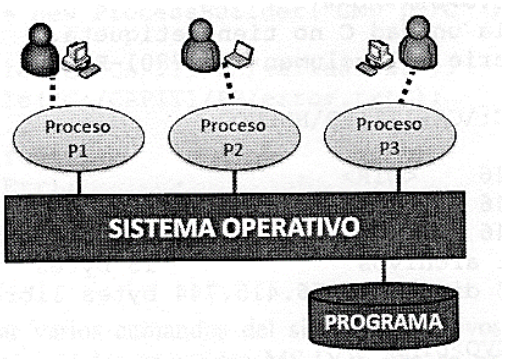
Para que el programa pueda hacer algo de verdad hay que ponerlo en ejecución.

Una primera definición incompleta de **proceso** sería la de un programa en ejecución. Es decir, un proceso es algo más que las líneas de código de un programa. Un proceso es una entidad dinámica. Puede compararse con el concepto de objeto en el ámbito de la POO.

De igual manera que en POO puede haber múltiples objetos de una clase determinada, aquí puede haber múltiples procesos que corresponden al mismo programa.

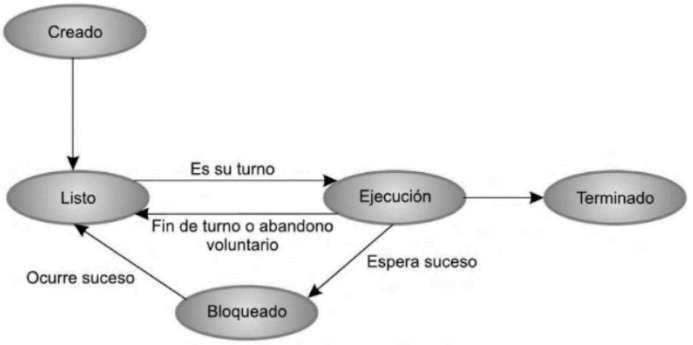
Como ejemplo consideremos un servidor de aplicaciones donde reside una aplicación web y existen varios usuarios ejecutando esa aplicación. Cada instancia del programa es un proceso.

En la siguiente figura puede observarse el ejemplo mencionado donde existe un programa almacenado en disco y 3 instancias de ese programa ejecutándose. Son 3 procesos, cada uno con su propia información.



1.1. Ciclo de vida de un proceso

En la siguiente figura puede apreciarse el ciclo de vida que suele seguir un proceso. Este ciclo de vida es prácticamente estándar en todos los Sistemas Operativos.



En un principio, un proceso no existe. En algún momento es creado. La forma de crearlo variará en función del lenguaje que se esté utilizando. Una vez creado el proceso, éste pasa al estado denominado **Listo.** Este estado significa que el proceso está en condiciones de hacer uso de la CPU en cuanto se le dé la oportunidad. El encargado de darle la oportunidad de usar la CPU es el denominado *planificador de procesos* o *scheduler*, que suele formar parte del SO.

Como lo normal es que haya más procesos que procesadores, no todos los procesos que pueden hacer uso de la CPU en un momento determinado pueden hacerlo realmente. Esos procesos permanecen en el estado listo hasta que el planificador decide darles tiempo de CPU. Cuando el planificador decide dar tiempo de CPU a un proceso, éste pasa al estado de **Ejecución**.

Como puede comprobarse, un proceso también puede pasar de **Ejecución** a **Listo**. Esta decisión la toma el planificador. El planificador sigue algún tipo de política de planificación para asignar la CPU a los distintos procesos. Una forma bastante justa y extendida de hacerlo es mediante la asignación de **rodajas de tiempo** (*cuantum*) a cada uno de los procesos, de tal forma que cuando un proceso cumple su tiempo de permanencia en el procesador, éste es desalojado y pasado al estado Listo. En este estado esperará una nueva oportunidad para pasar a Ejecución. También es posible que un proceso abandone voluntariamente la CPU y pase de esta forma al estado listo.

También podemos apreciar en la figura la presencia del estado **Bloqueado**. Un proceso puede pasar de Ejecución a Bloqueado cuando ha de esperar porque ocurra un determinado evento o suceso. Ejemplos de eventos pueden ser la espera por la terminación de una operación de Entrada/Salida, la espera por la finalización de una tarea por parte de otro proceso, un bloqueo voluntario durante un cierto período de tiempo, etc. Una vez que ocurre el evento por el que se está esperando, el proceso pasa al estado Listo. Al acto de cambiar un proceso de estado se le denomina *cambio de contexto*.

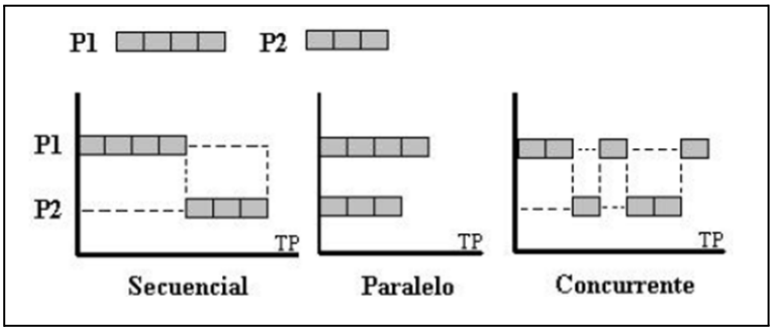
1.2. Programación concurrente

Según el diccionario de la RAE, una de las acepciones de la palabra concurrencia es

"Acaecimiento o concurso de varios sucesos en un mismo tiempo".

Si en esta definición sustituimos la palabra suceso por proceso, ya tenemos una primera aproximación a lo que va a ser la concurrencia en informática: *la existencia simultánea de varios procesos en ejecución.*

Dos procesos serán **concurrentes** cuando la primera instrucción de uno de ellos se ejecuta después de la primera instrucción del otro y antes de la última. Es decir, existe un solapamiento en la ejecución de sus instrucciones. No tienen por qué ejecutarse exactamente al mismo tiempo, simplemente es suficiente con el hecho de que exista un intercalado entre la ejecución de sus instrucciones. Si se ejecutan al mismo tiempo los dos procesos, entonces tenemos una situación de **programación paralela**. La programación concurrente es un paralelismo potencial.



Sin embargo, esto sólo es una parte de la verdad. No necesariamente un proceso tiene por qué ser todo el programa en ejecución, sino que puede ser parte de él.

Dicho de otra forma, un programa, al ponerse en ejecución, puede dar lugar a más de un proceso, cada uno de ellos ejecutando una parte del programa. Continuando con el mismo ejemplo, la aplicación/programa web puede dar lugar a más de un proceso: uno que controla las acciones del usuario con la interfaz, otro que hace las peticiones al servidor, etc.

Así pues, una definición más acertada de **proceso** es la de una actividad asíncrona susceptible de ser asignada a un procesador. Una definición bastante extendida es que un proceso es un programa en ejecución, pero no es muy exacto pues realmente un programa puede estar compuesto por diversos procesos. Un Sistema Operativo no deja de ser un programa con varios procesos que se ejecutan al mismo tiempo.

Ejemplo de concurrencia vs paralelismo: Construcción de una habitación de ladrillos.

Secuencial: El albañil llega, pone ladrillos hasta que finaliza una pared. Pasa a la siguiente pared, la finaliza. Así hasta que termina las cuatro paredes.

Paralelismo: Supongamos que hay 2 albañiles. Cada albañil construye dos paredes. Se construye la habitación en ½ tiempo.

Concurrencia: Puede haber varios albañiles, suponemos que sólo hay uno. El albañil pone 2 ladrillos en una pared, luego 2 en otra, luego 2 en otra… así hasta que termina las cuatro paredes. Las cuatro paredes se han llevado a cabo concurrentemente. Puede haber más de 1 albañil.

Actividad. De manera individual o por parejas, intentad buscar un ejemplo de Concurrencia vs Paralelismo en la vida real, de manera similar a la construcción de los 4 muros de la casa.

¡Sed originales!

1.2.1. Ventajas y desventajas de la programación concurrente

La principal ventaja de la programación concurrente es la **velocidad de ejecución**. Al subdividir un programa en procesos, éstos se pueden "repartir" entre procesadores o gestionar en un único procesador según importancia. Pero la compartición de recursos (fundamentalmente memoria) tiene riesgos y puede provocar errores difíciles de detectar y depurar.

Se llamó *problemas de concurrencia* a los errores ocasionados por el acceso no controlado a recursos compartidos. Los problemas más habituales y estudiados son:

Figura 1.16. Juego del pañuelo.

- **Exclusión mutua (o secciones críticas)**. Hay recursos en el sistema que deben ser accedidos en exclusión mutua. Un problema de exclusión mutua muy genérico y que ilustra el problema: si varios procesos en un ordenador envían diferentes trabajos de impresión se debe asegurar que las páginas no se intercalen. Es decir, se debe asegurar la exclusión mutua en el acceso a la impresora. Cuando esto ocurre, hay que garantizar que, si un proceso adquiere el recurso, otros procesos deberán esperar a que sea liberado. De lo contrario, el resultado puede ser imprevisto. En nuestro juego del pañuelo, el pañuelo ha de adquirirse en exclusión mutua, o lo coge un jugador o lo coge otro. Si lo cogen los dos a la vez, puede llegar a romperse, llevando a un malfuncionamiento del sistema.

- **Condición de sincronización**. Hay situaciones en las que un proceso debe esperar por la ocurrencia de un evento para poder seguir ejecutándose. Cuando esto ocurre, hay que garantizar que el proceso no prosigue hasta que no se produce el evento. De lo contrario, el resultado puede ser imprevisto. En nuestro ejemplo, un jugador ha de esperar a que digan su número para poder salir corriendo. Si sale corriendo antes, llevaría a un malfuncionamiento del sistema.

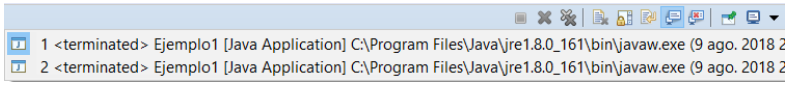
- **Interbloqueo.** Se produce una situación de interbloqueo cuando todos los procesos están esperando que ocurra un evento que nunca se producirá. Hay que garantizar que no se producen este tipo de situaciones. En nuestro ejemplo se produciría si un jugador se guarda el pañuelo y se va para su casa. El juez esperaría que le devolvieran el pañuelo y los jugadores esperarían que el juez dijese su número, pero ninguno de estos eventos va a ocurrir nunca. Se suele conocer también con el nombre de deadlock o abrazo mortal.

2. PROCESOS EN JAVA

2.1. ¿Qué nos permite Java?

Procesos: Es lo que llevamos haciendo a lo largo de todo el ciclo formativo. Creamos un programa con sus variables. Y este programa se ejecuta secuencialmente.

Si queremos tener más de un proceso, solamente tenemos que ejecutar más de un programa de Java (nos ponemos encima de la clase principal y pulsamos botón derecho **Run As→Java Aplication**)



Ejemplo 1. Prueba el siguiente ejemplo en Eclipse. Lanza varias ejecuciones a la vez de dicho programa para comprobar cómo varios **procesos** son ejecutados **concurrentemente**.

```
package es.iesvjp.psp;

public class Ejemplo1 {
	public static void main(String[] args) {
	System.out.println("Este es un proceso que cuenta hasta 100");
		for (int i = 1; i <= 100; i++) {
			System.out.println(i);
			try {
				Thread.sleep(50);
			} catch (InterruptedException e) {
				e.printStackTrace();
			}

		}

	}

}
```

2.2. Hilos

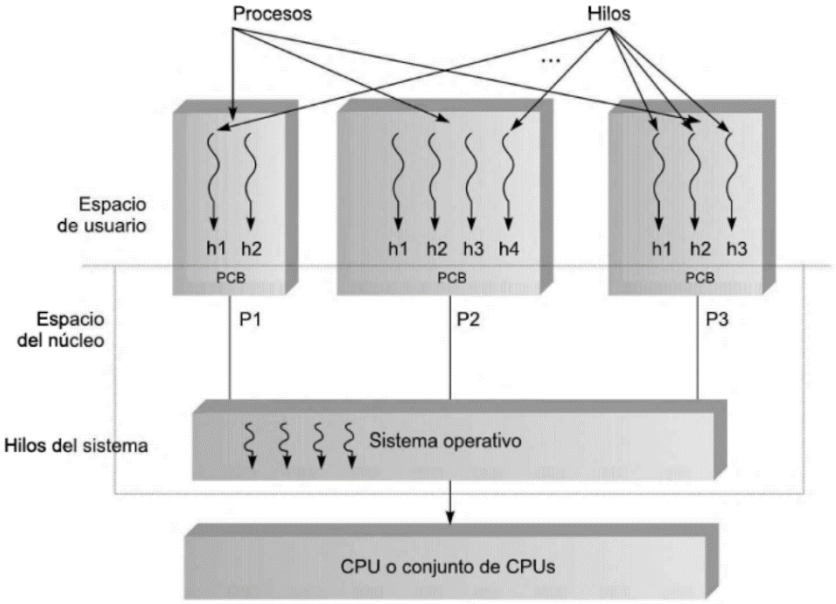
De la misma manera que un Sistema Operativo puede ejecutar varios procesos al mismo tiempo bien sea por concurrencia o paralelismo, dentro de un proceso puede haber varios hilos de ejecución. Por tanto, **un hilo puede definirse como cada secuencia de control dentro de un proceso que ejecuta sus instrucciones de forma independiente**.

En la siguiente figura podemos ver cómo sobre el hardware subyacente (una o varias CPU's) se sitúa el Sistema Operativo. Sobre éste se sitúan los procesos (P_i) que pueden ejecutarse concurrentemente y dentro de éstos se ejecutan los hilos (h_j) que también se pueden ejecutar de forma concurrente dentro del proceso. Es decir, tenemos concurrencia a dos niveles, una entre procesos y otra entre

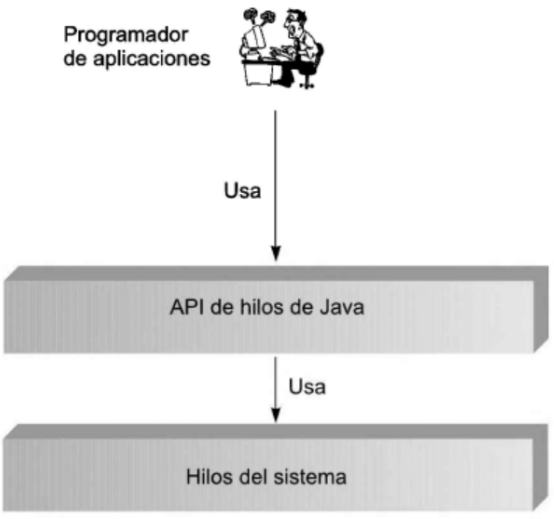
hilos de un mismo proceso. Si por ejemplo tenemos dos procesadores, se podrían estar ejecutando al mismo tiempo el hilo 1 del proceso 1 y el hilo 2 del proceso 3. Otra posibilidad podría ser el hilo 1 y el hilo 2 del proceso 1.

Los procesos son entidades pesadas. La estructura del proceso está en la parte del núcleo y, cada vez que el proceso quiere acceder a ella, tiene que hacer algún tipo de llamada al sistema, consumiendo tiempo extra de procesador. Por otra parte, los cambios de contexto entre procesos son costosos en cuanto a tiempo de computación se refiere.

Por el contrario, la estructura de los hilos reside en el espacio de usuario, con lo que **un hilo es una entidad ligera**. Los hilos comparten la información del proceso (código, datos, etc.). Si un hilo modifica una variable del proceso, el resto de hilos verán esa modificación cuando accedan a esa variable. Los cambios de contexto entre hilos consumen poco tiempo de procesador, de ahí su éxito.

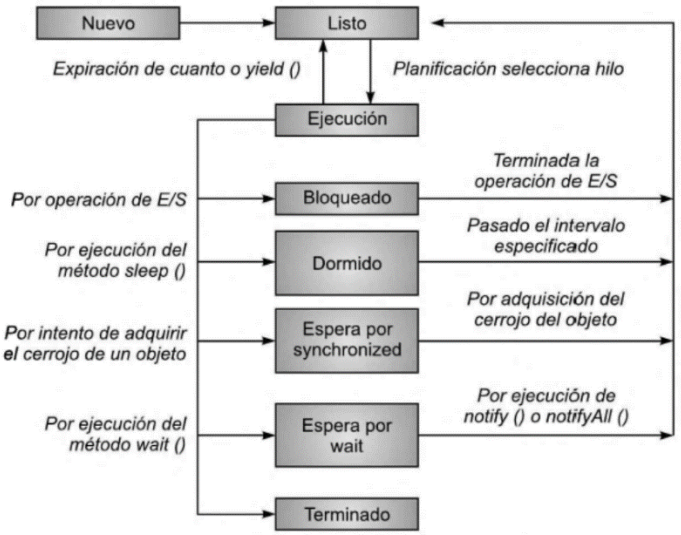


Podemos hablar de dos niveles de hilos: aquellos que nosotros usaremos para programar y que pueden crearse desde lenguajes de programación como Java, y aquellos otros hilos del propio SO que sirven para dar soporte a nuestros hilos de usuario y que son los hilos del sistema. Cuando nosotros programamos con hilos, hacemos uso de un API (*Application Program Interface*) proporcionado por el lenguaje, el SO o un tercero mediante una librería externa. El implementador de este API será el que haya usado los hilos del sistema para dar soporte de ejecución a los hilos que nosotros creamos en nuestros programas.



2.2.1. Estados de un hilo

El ciclo de vida de un hilo comprende diversos estados por los que puede ir pasando. La siguiente figura muestra estos estados y los métodos que provocan el paso de un estado a otro. **Nuevo** es el estado en el que se encuentra un hilo cuando se crea con el operador `new`. Al lanzarlo con `start()` pasa al estado **Listo**, es decir, es susceptible de acaparar el procesador si el planificador se lo permite. Cuando obtiene el procesador pasa al estado **Ejecución**.



Una vez que el hilo está en ejecución puede pararse por diversos motivos:

- Por hacer una operación de E/S. Saldrá de este estado al completarse dicha operación.
- Por ejecutar el método `sleep(milisegundos)`. Saldrá de este estado al cumplirse el tiempo especificado como parámetro.
- Por intentar adquirir el cerrojo de un objeto. Hablaremos del cerrojo de un objeto en el apartado 2.6. Sincronización de Hilos: Monitores
- Por ejecutar el método `wait()`. Saldrá de este estado cuando se ejecute el método `notify()` o `notifyAll()` por parte de otro hilo. Hablaremos de esto en el apartado *2.6. Sincronización de Hilos: Monitores*

2.3. Hilos en Java

Nada más empezar un programa en Java, existe un hilo de ejecución que es el indicado por el método **main()**, es el denominado **hilo principal**. Si no se crean más hilos desde el hilo principal, tendremos algo como lo representado en la Figura 2.10, donde sólo existe un hilo de ejecución que va recorriendo los distintos objetos participantes en el programa según se vayan produciendo las distintas llamadas entre métodos de los mismos.

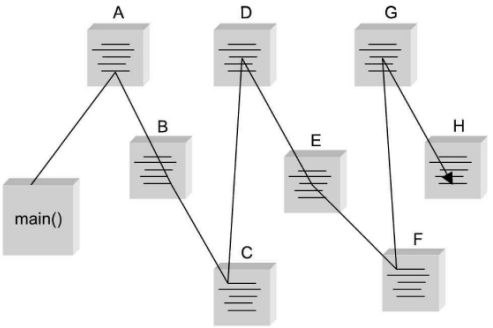


Figura 2.10. Varios objetos y un solo hilo.

Sin embargo, si desde el hilo principal se crean por ejemplo dos hilos, tendremos algo como lo representado en la Figura 2.11, en la que se puede ver cómo el programa se está ejecutando al mismo tiempo por tres sitios distintos: el hilo principal más los dos hilos creados. Los hilos pueden estar ejecutando código en diferentes objetos, código diferente en el mismo objeto o incluso el mismo código en el mismo objeto y al mismo tiempo.

Es necesario en este punto ver la diferencia entre objeto e hilo. Un objeto es algo estático, tiene una serie de atributos y métodos. Quien realmente ejecuta esos métodos es el hilo de ejecución. En ausencia de hilos sólo hay un hilo que va recorriendo los objetos según se van produciendo las llamadas entre métodos de los objetos. Podría darse el caso del objeto E en la Figura 2.11, donde es posible que tres hilos de ejecución distintos estén ejecutando el mismo método al mismo tiempo.

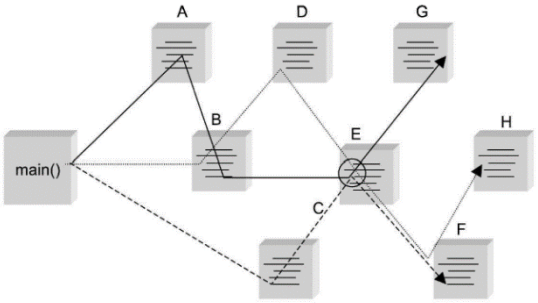


Figura 2.11. Tres hilos «atravesando» varios objetos.

2.3.1. Constructores y métodos de un hilo

Constructor	Función
`Thread()`	Crea un nuevo objeto Thread. Este constructor tiene el mismo efecto que `Thread (null,null, gname)`, donde `gname` es un nombre generado automáticamente y que tiene la forma " Thread- "+n, donde n es un entero asignado consecutivamente.
`Thread (String name)`	Crea un nuevo objeto Thread, asignándole el nombre *name.*
`Thread (Runnable target)`	Crea un nuevo objeto Thread. *target* es el objeto que contiene el método run( ) que será invocado al lanzar el hilo con start().
`Thread (Runnable target, String name)`	Crea un nuevo objeto Thread, asignándole el nombre *name. target* es el objeto que contiene el método run ( ) que será invocado al lanzar el hilo con start ().

Método	Función
`start()`	Hace que el hilo comience la ejecución; La MVJ llama al método run() de este hilo.
`boolean isAlive()`	Comprueba si el hilo está vivo. Devuelve `true` si el hilo ha sido inicializado y no ha terminado de forma anormal o no ha sido abortado, de lo contrario devuelve `false`.
`sleep(long milisegundos)`	Hace que el hilo actualmente en ejecución pase a dormir durante el número de milisegundos especificado. Puede lanzar la excepción: *InterruptedException*
`run()`	Es el cuerpo del hilo. Es llamado por start() después de que el hilo se haya inicializado. **Es el único método de la interfaz RUNNABLE**
`String toString()`	Devuelve una representación en formato cadena de este hilo. Incluye el nombre, la prioridad y el grupo de hilos.
`Long getId()`	Devuelve el identificador del hilo
`void yield()`	Hace que el hilo actual pare para dejar la ejecución de otros hilos. Cede el puesto a otros hilos. Es decir, pasa del estado *Ejecución* a *Listo*.
`String getName()`	Devuelve el nombre del hilo
`setName(String name)`	Cambia el nombre del hilo
`int getPriority()`	Devuelve la prioridad del hilo
`setPriority(int p)`	Cambia la prioridad del hilo a la del valor p
`void interrupt()`	Interrumpe la ejecución del hilo
`void join()`	Espera a que este hilo muera. Puede lanzar la excepción: *InterruptedException*
`void join (long millis)`	Espera como mucho *millis* milisegundos para que este hjlo muera. Puede lanzar la excepción: *InterruptedException*
`boolean interrupted()`	Comprueba si el hilo ha sido interrumpido
`Thread currentThread()`	Devuelve una referencia al objeto hilo que se está ejecutando actualmente (similar a this)
`boolean isDaemon()`	Comprueba si el hilo es un Demonio
`setDaemon(boolean on)`	Establece este hilo como demonio o no.
`void stop()`	Detiene el hilo. Método deprecated (obsoleto)

2.4. Creación de hilos

Existen 2 formas de trabajar con hilos en cuanto a su creación se refiere:

- Heredando de la clase `Thread`.
- Implementando la interfaz `Runnable`.

2.4.1. Heredando de la clase Thread

La clase `Thread` tiene un método especial cuya signatura es `public void run ()`. Cuando queramos crear una clase cuyas instancias queremos que sean un hilo, tendremos que heredar de la clase `Thread` y redefinir este método. Dentro de este método se escribe el código que queremos que se ejecute cuando se lance a ejecución el hilo. Se podría decir que `main()` es a un programa lo que run() es a un hilo. Veamos un ejemplo en el que creamos una clase denominada `ThreadConHerencia`.

Si queremos que los objetos pertenecientes a esta clase sean hilos, entonces debemos extender la clase `Thread`. El constructor de la clase `ThreadConHerencia` recibe como parámetro el número del hilo para luego mostrarlo en el método run. En el método `run()` especificamos el código que ha de ejecutarse al lanzar el hilo: mostrar 4 veces el número del hilo.

```
package es.iesvjp.psp;

public class ThreadConHerencia extends Thread{
	int numHilo;

	public int getNumHilo() {
		return numHilo;
	}

	public ThreadConHerencia(int hilo) {
		this.numHilo=hilo;

	}

	public void run()
	{
		for (int i = 0; i < 4; i++) {
			System.out.println("Hilo " + this.getNumHilo());

		}

	}
}
```

Hasta aquí lo único que hemos hecho es crear una clase. Los objetos pertenecientes a esa clase serán hilos. En las siguientes líneas, desde el programa principal creamos dos objetos que son hilos: thH1 y thH2. Para ponerlos en ejecución hay que invocar el método `start()`. Este método pertenece a la clase Thread. Se encarga de hacer algunas inicializaciones propias del hilo y posteriormente invoca al método `run()` . Es decir, invocando `start()` se ejecutan en orden los métodos start (heredado) y run (redefinido).

```
package es.iesvjp.psp;

public class Main {

	public static void main(String[] args) {
		ThreadConHerencia thH1= new ThreadConHerencia(1);
		ThreadConHerencia thH2= new ThreadConHerencia(2);

		thH1.start();
		thH2.start();
	       System.out.println("Fin del hilo principal");
	}
}
```

Una posible salida para este programa sería:

```
Fin del hilo principal
Hilo 2
Hilo 1
Hilo 2
Hilo 1
Hilo 2
Hilo 1
Hilo 1
Hilo 2
```

Puede apreciarse cómo se intercalan las salidas de los dos hilos. Recordemos que tenemos tres hilos: el principal más los dos creados. Y entre ellos se pueden intercalar de cualquier forma; por tanto, no es extraño ver cómo el hilo principal acaba antes que los otros o cómo el hilo 2 comienza antes que el uno.

Ejemplo del Reloj

Vamos a crearnos una aplicación que muestre concurrentemente 10 veces la hora del sistema cada segundo y ejecute un beep también cada segundo. Para ello será necesario que nos creemos 2 hilos: uno para mostrar la Hora y otro para generar el Sonido beep:

```
package es.iesvjp.psp.hilos.Reloj;

import java.time.LocalDateTime;

public class Hora extends Thread {

	@Override
	public void run() { /* código concurrente */

		for (int i = 0; i < 10; i++) {
			System.out.println(LocalDateTime.now().toString());
			try {
				sleep(1000);/* esperar 1000 ms */
			} catch (InterruptedException e) {
				e.printStackTrace();
			}

		}

	}
}
```

```
package es.iesvjp.psp.hilos.Reloj;

public class Sonido extends Thread {
	@Override
	public void run() { /* código concurrente */

		for (int i = 0; i < 10; i++) {
			System.out.println("Beep!!");

			java.awt.Toolkit.getDefaultToolkit().beep();

			try {
				sleep(1000);
			} catch (InterruptedException e) {
				e.printStackTrace();
			}
		}
	}
}
```

Y la clase Reloj dónde inicializamos los dos hilos:

```
package es.iesvjp.psp.hilos.Reloj;

public class Reloj {
	public static void main(String[] args) {
		Hora hora = new Hora();
		Sonido sonido = new Sonido();
		hora.start();
		sonido.start();
	}
}
```

```
Beep!!
2018-08-09T23:05:50.147
2018-08-09T23:05:51.163
Beep!!
2018-08-09T23:05:52.167
Beep!!
2018-08-09T23:05:53.171
Beep!!
2018-08-09T23:05:54.176
Beep!!
2018-08-09T23:05:55.183
Beep!!
2018-08-09T23:05:56.192
Beep!!
2018-08-09T23:05:57.202
```

Ejercicio 1. Descárgate el programa del Reloj y ejecútalo. Una vez probado, añade una hebra que escriba "Se te está acabando el tiempo" en el terminal cada 5 s, y volver a ejecutar el programa con esta modificación.

Ejercicio 2. Crea dos clases (hilos) Java que extiendan la clase Thread. Uno de los hilos debe visualizar en pantalla en un bucle infinito la palabra TIC y el otro hilo la palabra TAC. Dentro del bucle utiliza el método sleep() para que nos dé tiempo a ver las palabras que se visualizan cuando lo ejecutemos, tendrás que añadir un bloque try-catch (para capturar la excepción InterruptedException). Crea después la función main() que haga uso de los hilos anteriores. ¿Se visualizan los textos TIC y TAC de forma ordenada (es decir TIC TAC TIC TAC ...)?

2.4.2. Implementando la interfaz Runnable

```
package es.iesvjp.psp;

public class ThreadConRunnable implements Runnable {
	String palabra;

	public ThreadConRunnable(String _palabra) {
		this.palabra = _palabra;
	}

	public void run() {
		for (int i = 0; i < 10; i++)
			System.out.println(palabra);

	}
}
```

Hasta aquí simplemente hemos creado una clase. Al contrario que antes, los objetos de esta clase no serán hilos pues no hemos heredado de la clase Thread. Si queremos que un objeto de la clase `ThreadConRunnable` se ejecute en un hilo independiente, entonces tendremos que crear un objeto de la clase `Thread` y pasarle como parámetro el objeto donde queremos que empiece su ejecución ese hilo. En las siguientes líneas de código se crean dos objetos de la clase `ThreadConRunnable` (a y b) y dos hilos (th1 y th2) a los que se le pasa como parámetro los objetos a y b. Posteriormente hay que invocar el método `start()` de la clase Thread que ya se encarga de invocar al método `run()` de los objetos a y b, respectivamente.

```
package es.iesvjp.psp;

public class MainRunnable {
	public static void main (String args[]) {
		ThreadConRunnable a= new ThreadConRunnable("Hilo 1");
		ThreadConRunnable b = new ThreadConRunnable("Hilo 2");
		Thread th1 = new Thread (a);
		Thread th2 = new Thread (b);
		th1.start();
		th2.start();
		System.out.println("Fin del hilo principal");
	}
}
```

Una posible salida de este ejemplo sería la misma de antes.

2.4.3. Comparando ambos métodos

La segunda forma de crear hilos puede parecer un poco más confusa. Sin embargo, es **más apropiada** y se utiliza más que la primera. Esto se debe a que como en Java no hay herencia múltiple, el utilizar la primera opción hipoteca nuestra clase para que ya no pueda heredar de otras clases. Sin embargo, con la segunda opción podemos hacer que el método run () de una clase se ejecute en un hilo y además esta clase herede el comportamiento de otra clase. En la medida de lo posible, deberíamos usar esta forma.

Ejercicio 3. Transforma el ejercicio del Reloj para que implemente la interfaz Runnable.

2.5. Planificación y prioridades

Java fue diseñado principalmente para obtener una gran portabilidad. Este objetivo hipoteca de alguna forma también el modelo de hilos a usar en Java pues se deseaba que fuera fácil de soportar en cualquier plataforma y, como sabemos cada plataforma hace las cosas de una forma diferente. Esto hace que el modelo de hilos sea demasiado generalista. Sus principales características son:

- Todos los hilos de Java tienen una prioridad y se supone que el planificador dará preferencia a aquel hilo que tenga una prioridad más alta. Sin embargo, no hay ningún tipo de garantía de que en un momento determinado el hilo de mayor prioridad se esté ejecutando.
- Las rodajas de tiempo pueden ser aplicadas o no. Dependerá de la gestión de hilos que haga la librería sobre la que se implementa la máquina virtual java.

Ante una definición tan vaga como ésta, uno se puede preguntar cómo es posible escribir código portable y, lo que es peor, el comportamiento que tendrá la aplicación dependiendo del sistema subyacente. Ante esto, lo que el programador no debe hacer es ningún tipo de suposición y ponerse siempre en el peor de los casos:

- Se debe asumir que los hilos pueden intercalarse en cualquier punto en cualquier momento.
- Nuestros programas no deben estar basados en la suposición de que vaya a haber un intercalado entre los hilos. Si esto es un requisito en nuestra aplicación, deberemos introducir el código necesario para que esto sea así.

2.5.1. Prioridades

Las prioridades de cada hilo en Java están en el rango de **1** ***(MIN PRIORITY)*** a **10** ***(MAX_PRIORITY)***. La prioridad de un hilo es inicialmente la misma que la del hilo que lo creó. Por defecto, todo hilo tiene la prioridad **5** ***(NORM_PRIORITY).*** El planificador siempre pondrá en ejecución aquel hilo con mayor prioridad (teniendo en cuenta lo dicho en el punto anterior). Los hilos de prioridad inferior se ejecutarán cuando estén bloqueados los de prioridad superior.

Las prioridades se pueden cambiar utilizando el método `setPriority(nuevaPrioridad)`. La prioridad de un hilo en ejecución se puede cambiar en cualquier momento. El método `getPriority()` devuelve la prioridad de un hilo.

El método `yield()` hace que el hilo actualmente en ejecución ceda el paso de modo que puedan ejecutarse otros hilos listos para ejecución. El hilo elegido puede ser incluso el mismo que ha dejado paso, si es el de mayor prioridad.

En el siguiente programa se puede ver cómo se crean dos hilos (th1 y th2) y al primero de ellos se le cambia la prioridad. Esto haría que th2 acaparase el procesador hasta finalizar pues nunca será interrumpido por un hilo de menor prioridad.

```
package es.iesvjp.psp.hilos.Ejemplos;

public class Ejemplo2 implements Runnable{

	String palabra;

	public Ejemplo2(String _palabra) {
		this.palabra = _palabra;
	}

	@Override
	public void run() {
		for (int i = 0; i < 100; i++) {
			System.out.println(palabra);

		}

	}
}
```

```
package es.iesvjp.psp.hilos.Ejemplos;

public class Main2 {

	public static void main ( String args [ ]) {
		Ejemplo2 ejem1 = new Ejemplo2("Hilo 1");
		Ejemplo2 ejem2 = new Ejemplo2("Hilo 2");

		Thread th1 = new Thread (ejem1);
		Thread th2 = new Thread (ejem2);
		th1.start();
		th1.setPriority(1);//bajamos la prioridad del hilo 1

		System.out.println ( "Prioridad del Hilo 1: "+th1.getPriority());
		th2.start() ;
		System.out.println ("Prioridad del Hilo 2: "+th2.getPriority());
	}
}
```

Uno de los posibles resultados sería:

```
Prioridad del Hilo 1: 1
Hilo 1
Hilo 1
Hilo 1
Hilo 1
Hilo 1
Hilo 1
Hilo 1
Hilo 1
Prioridad del Hilo 2: 5
Hilo 1
Hilo 2
Hilo 2
Hilo 2
Hilo 2
Hilo 2
Hilo 2
Hilo 2
```

.....

2.5.2. Hilos daemon

Los hilos de ejecución *demonio* también se llaman *servicios*, porque se ejecutan, normalmente, con prioridad baja y proporcionan un servicio básico a un programa o programas cuando la actividad de la máquina es reducida.

Los hilos demonio son útiles cuando un hilo debe ejecutarse en segundo plano durante largos períodos de tiempo. Un ejemplo de *hilo demonio* que está ejecutándose continuamente es el recolector de basura (*garbage collector*).

Antes de lanzar un hilo, éste puede ser definido como un *Daemon,* indicando que va a hacer una ejecución continua para la aplicación como tarea de fondo. Entonces la máquina virtual abandonará la ejecución cuando todos los hilos que no sean Daemon hayan finalizado su ejecución. Los hilos Daemon tienen la prioridad más baja.

Se usa el método `setDaemon(true)` para marcar un hilo como hilo demonio y se usa `getDaemon()` para comprobar ese indicador. Por defecto, la cualidad de demonio se hereda desde el hilo que crea el nuevo hilo. No puede cambiarse después de haber iniciado un hilo.

La propia MVJ pone en ejecución algunos hilos daemon cuando ejecutamos un programa. Entre ellos cabe destacar el *garbage collector* o recolector de basura, que es el encargado de liberar la memoria ocupada por objetos que ya no están siendo referenciados.

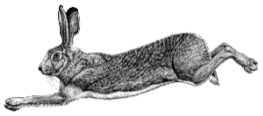
Ejercicios

5. Crea una clase que extienda de Thread y que llamaremos PintaDiezVeces. Dicha clase tendrá un atributo de tipo String que se inicializará en el constructor.

   El cometido de la clase *PintaDiezVeces* es el de pintar 10 veces por pantalla la cadena con la que se haya instanciado el objeto. La ejecución de PintaDiezVeces se realizará en el hilo principal.

   Desde el principal, arranca 5 hilos que pinten por pantalla 10 veces la cadena junto con el ID del hilo, su prioridad y su nombre. La cadena será diferente para cada hilo.

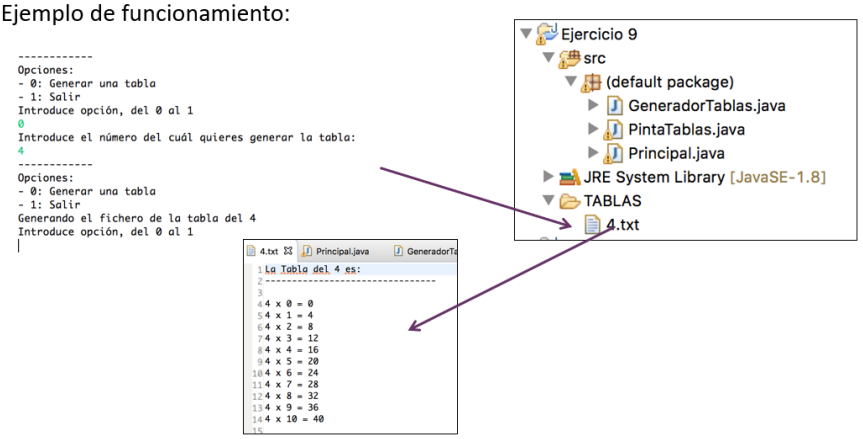
6. Vamos a simular la fábula de la tortuga, la liebre y el guepardo. La carrera consistirá en pintar el nombre de cada animal 100 veces por pantalla. Cada animal ejecutará su recorrido en un hilo distinto. Existirá una sola clase *CorredorAnimal.* Para diferenciar qué animal es cada hilo cambiaremos el nombre de cada hilo con el nombre del animal correspondiente. Además, utilizaremos este nombre para pintarlo durante la carrera. Revisa la documentación de prioridades de Java para darle a la tortuga la mínima prioridad, la liebre una prioridad media y el guepardo la máxima prioridad.



   ¿Qué animal crees que llegará antes a la meta? ¿Sigue algún patrón determinado la salida que se ha producido por pantalla? ¿Crees que funcionan correctamente las prioridades? **Intenta que la transición entre animales sea más fluida**.

7. La idea principal de este programa es escribir un programa concurrente que me permita generar las tablas de multiplicar de cualquier número que introduzca el usuario por teclado.
   Para ello, programaremos la clase **PintaTabla**, que extenderá de Thread y cuya tarea será guardar en la carpeta TABLAS (dentro de nuestro proyecto) un fichero con la tabla de multiplicar que solicite el usuario. **PintaTabla** recibirá un entero en el constructor, que será el número del cuál crearos la tabla de multiplicar.
   La clase **GeneradorTablas** será la encargada de gestionar la lógica de usuario del programa:
   - Le preguntará al usuario si quiere generar una tabla de multiplicar.
   - En caso afirmativo le pregunta al usuario el número de qué número quiere generar la tabla de multiplicar. Crea una instancia de **PintaTabla** para realizar dicha tarea.
   - Le preguntará al usuario si quiere salir.

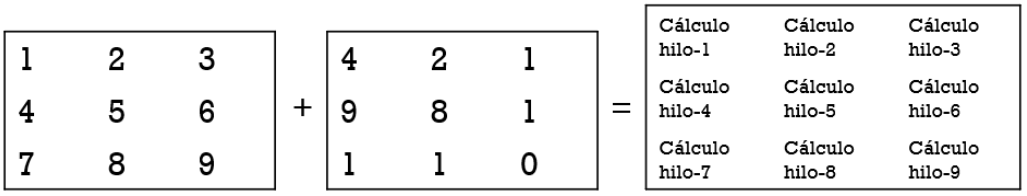
   Ejemplo de funcionamiento:



8. Vamos a crearnos un programa para sumar 3 matrices cuadradas. Para ello nos crearemos una clase Principal que contenga 3 matrices estáticas para que puedan ser accedidas desde cualquier lugar de nuestro programa:

```
public static int [][] matriz1 = {{1,2,3},{4,5,6},{7,8,9}};
public static int [][] matriz2 = {{4,2,1},{9,8,1},{1,1,0}};
public static int [][] matrizResultado = new int [3][3];
```

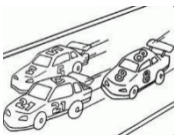
La clase Principal lanzará 9 hilos (uno por cada celda de las matrices). Cada hilo **se dormirá 1 milisegundo** y hará la suma parcial de cada posición de la matriz y lo almacenará en matrizResultado.



El principal pintará por pantalla el resultado final de la multiplicación de las matrices, es decir, pintará "matrizResultado". Ten en cuenta que para pintar la matriz resultado debemos espera a que terminen todos los hilos.

```
5   4   4
13  13  7
8   9   9
```

2.6. Sincronización de Hilos: Monitores



Cuando 2 o más procesos pueden acceder a las mismas variables y objetos al mismo tiempo, los datos pueden corromperse si un proceso "corre" lo suficientemente rápido como para vencer al otro, a este escenario se le conoce como **"condición de carrera".** Por ello, cuando dos o más procesos necesitan acceder de manera simultánea a un recurso compartido necesitan asegurarse de que sólo uno de ellos accede al mismo cada vez (**exclusión mutua**).

Java proporciona un soporte único, el **monitor**, es un objeto que se utiliza como cerrojo exclusivo. Solo uno de los hilos puede ser el propietario de un monitor en un instante dado. Los restantes hilos que estuviesen intentando acceder al monitor bloqueado quedan en suspenso hasta que el hilo propietario salga del monitor.

Todos los objetos de Java disponen de un monitor propio implícitamente asociado a ellos. La manera de acceder a un objeto monitor es llamando a un método marcado con la palabra clave **synchronized.** Durante todo el tiempo en que un hilo permanezca en un método sincronizado, los demás hilos que intenten llamar a un método sincronizado sobre la misma instancia tendrán que esperar. Para salir del monitor y permitir el control del objeto al siguiente hilo en espera, el propietario del monitor sólo tiene que volver del método.

Claves para la sincronización y el cerrojo de los objetos:

- Solo métodos y bloque de código pueden ser sincronizados, nunca una variable o clase.

```
public class Sincronizados2 {
  public void m1() {
    // cualquier código. Será no sincronizado
    synchronized (this) {
      // sólo esta parte es sincronizada
    }
    // cualquier código. Será no sincronizado
  }
}
```

- Cada objeto tiene solamente un cerrojo.
- No todos los métodos de una clase deben ser sincronizados, una misma clase puede tener métodos sincronizados y no sincronizados.
- Si una clase tiene ambos tipos de métodos, múltiples hilos pueden acceder a sus métodos no sincronizados, el único código protegido es aquel dentro de un método sincronizado.
- **Si un hilo pasa a estado "dormido" (sleep) no libera el o los cerrojos que pudiera llegar a tener, los mantiene hasta que se complete.**
- Si un hilo pasa a estado "espera por wait", SÍ libera el cerrojo o cerrojos que pudiera llegar a tener.

El bloque synchronized lleva entre paréntesis la referencia a **un objeto**. Cada vez que un thread intenta acceder a un bloque sincronizado le pregunta a **ese** objeto si no hay algún otro thread que ya tenga el *lock* para **ese** objecto. En otras palabras, le pregunta si no hay otro thread ejecutando algún bloque sincronizado con **ese** objeto (y recalco que es **ese** objeto porque en eso radica la clave para entender el funcionamiento)

Si el lock está tomado por otro thread, entonces el thread actual es suspendido y puesto en espera hasta que el lock se libere. Si el lock está libre, entonces el thread actual toma el lock del objeto y

entra a ejecutar el bloque. Al tomar el lock, cuando venga el proximo thread a intentar ejecutar un bloque sincronizado con **ese** objeto, será puesto en espera.

Ejemplo de sincronización de hilos:

Vamos a ver a continuación un ejemplo dónde es necesaria la sincronización de los hilos. 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. Veamos mediante clases como sería el ejemplo.

Se define la clase **Cuenta**, con un atributo saldo y 2 métodos, uno devuelve el valor del saldo y otro realiza las comprobaciones para hacer la retirada de dinero, es decir que el saldo actual sea >= que la cantidad que se quiere retirar; el constructor inicia el saldo actual.

```
public class Cuenta {

	private double saldo;

	public Cuenta(double saldo) {
		this.saldo = saldo;
	}

	public double getSaldo() {
		return saldo;
	}

	public void setSaldo(double saldo) {
		this.saldo = saldo;
	}

	public void retirar(double cantidad, String persona) {
		if (getSaldo() >= cantidad) {
		//Hay suficiente saldo
		System.out.println("Se va a retirar dinero. El saldo actual es de: "
								+ getSaldo() + "€");
		saldo=saldo-cantidad;
		System.out.println(persona +" retira "+cantidad +" €. (Saldo actual:
								"+ getSaldo()+"€)");
		}
		else {
		//no hay suficiente saldo
		System.out.println(persona +" no puede retirar "+cantidad +" € porque
				el saldo actual es insuficiente: "+ getSaldo()+"€");
		}

		if(getSaldo()<0)
		{
			System.out.println("SALDO NEGATIVO:"+ getSaldo());
		}
	}
}
```

A continuación, creamos la clase *Persona* que extiende de **Thread** y usa la clase *Cuenta* para retirar el dinero. El constructor recibe una cadena que es el nombre de la persona (hilo), y la cuenta que será compartida por varias personas. En el método **run()** se repite un bucle 4 veces, donde se invoca al método *retirar()* de la clase *Cuenta* pasándole la cantidad a retirar (en este caso siempre es 10 €) y el

nombre de la persona:

```
package es.iesvjp.psp.hilos.Bancov1;

public class Persona extends Thread {
	private String nombre;
	private Cuenta cuenta;

	public Persona(String nombre, Cuenta cuenta) {
		this.nombre=nombre;
		this.cuenta = cuenta;
	}

	public String getNombre() {
		return nombre;
	}

	public void setNombre(String nombre) {
		this.nombre = nombre;
	}

	public void run()
	{
		for (int i = 0; i <4; i++) {

			this.cuenta.retirar(10, getNombre());
			try {
				Thread.sleep(1000);
			} catch (InterruptedException e) {
				// TODO Auto-generated catch block
				e.printStackTrace();
			}

		}
	}

}
```

Por último se crea la clase Principal con el método main(), donde primero se define un objeto de la clase Cuenta y se le asigna un saldo inicial de 50. A continuación se crean dos objetos de la clase Persona, y se inician los hilos:

```
package es.iesvjp.psp.hilos.Bancov1;

public class Principal {

	public static void main(String[] args) {
		Cuenta cuenta= new Cuenta(50);
		Persona ana= new Persona("Ana", cuenta);
		Persona juan= new Persona("Juan", cuenta);

		ana.start();
		juan.start();
	}
}
```

Ejecutamos el programa, y esta es una de las posibles ejecuciones:

```
Se va a retirar dinero. El saldo actual es de: 50.0€
Juan retira 10.0 €. (Saldo actual: 40.0€)
Se va a retirar dinero. El saldo actual es de: 50.0€
Ana retira 10.0 €. (Saldo actual: 30.0€)
```

```
Se va a retirar dinero. El saldo actual es de: 30.0€
Se va a retirar dinero. El saldo actual es de: 30.0€
Juan retira 10.0 €. (Saldo actual: 20.0€)
Ana retira 10.0 €. (Saldo actual: 10.0€)
Se va a retirar dinero. El saldo actual es de: 10.0€
Se va a retirar dinero. El saldo actual es de: 10.0€
Ana retira 10.0 €. (Saldo actual: 0.0€)
Juan retira 10.0 €. (Saldo actual: -10.0€)
SALDO NEGATIVO:-10.0
SALDO NEGATIVO:-10.0
Ana no puede retirar 10.0 € porque el saldo actual es insuficiente: -10.0€
Juan no puede retirar 10.0 € porque el saldo actual es insuficiente: -10.0€
SALDO NEGATIVO:-10.0
SALDO NEGATIVO:-10.0
```

Como podemos comprobar, el saldo actual no se muestra adecuadamente y además nos ha permitido retirar dinero cuando el saldo era 0€. Para evitar esta situación la operación de retirar dinero, método **retirar()** 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**. Cuando un hilo invoca un método synchronized, trata de tomar el bloqueo/cerrojo 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 método(){
//instrucciones atómicas...
}
```

El método retirar en nuestro ejercicio quedaría:

```
synchronized public void retirar(double cantidad, String persona){
 //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:

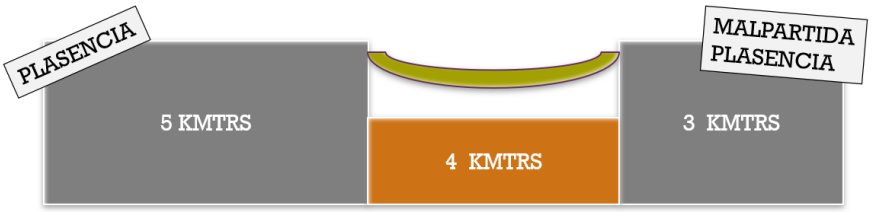
```
Se va a retirar dinero. El saldo actual es de: 50.0€
Juan retira 10.0 €. (Saldo actual: 40.0€)
Se va a retirar dinero. El saldo actual es de: 40.0€
Ana retira 10.0 €. (Saldo actual: 30.0€)
Se va a retirar dinero. El saldo actual es de: 30.0€
Ana retira 10.0 €. (Saldo actual: 20.0€)
Se va a retirar dinero. El saldo actual es de: 20.0€
Juan retira 10.0 €. (Saldo actual: 10.0€)
Se va a retirar dinero. El saldo actual es de: 10.0€
Ana retira 10.0 €. (Saldo actual: 0.0€)
Juan no puede retirar 10.0 € porque el saldo actual es insuficiente: 0.0€
Ana no puede retirar 10.0 € porque el saldo actual es insuficiente: 0.0€
Juan no puede retirar 10.0 € porque el saldo actual es insuficiente: 0.0€
```

ACTIVIDADES

9. La carretera hasta Malpartida de Plasencia está en obras en un tramo intermedio por consecuencia de un terremoto. Si suponemos que el tramo hasta llegar a Malpartida de Plasencia son 12 kms:

- Los primeros 5 kms no han sufrido percances.
- Los siguientes 4 kms han sido tragados por la tierra. Los ingenieros placentinos han colocado un puente colgante que soporta 1 coche en el tramo peligroso.
- Los últimos 3 kms se encuentran en perfecto estado.

Tu misión será realizar una simulación suponiendo que salen 4 coches desde Plasencia hasta el Casar (cada uno simulado por un hilo).
Utiliza las primitivas de Java como creas preciso para asegurarte que solamente un coche a la vez puede pasar por el puente colgante.
Por supuesto los otros coches pueden ir avanzando por la carretera mientras uno está en el puente.



Posible salida por pantalla:

```
El coche-1 va por el km1 de la carretera
El coche-1 va por el km2 de la carretera
El coche-1 va por el km3 de la carretera
El coche-0 va por el km1 de la carretera
El coche-1 va por el km4 de la carretera
El coche-1 va por el km5 de la carretera
El coche-0 va por el km2 de la carretera
El coche-0 va por el km3 de la carretera
El coche-0 va por el km4 de la carretera
El coche-0 va por el km5 de la carretera
        ***El coche-1 está atravesando el km 6 del puente
        ***El coche-1 está atravesando el km 7 del puente
        ***El coche-1 está atravesando el km 8 del puente
        ***El coche-1 está atravesando el km 9 del puente
El coche-1 va por el km10 de la carretera
        ***El coche-0 está atravesando el km 6 del puente
El coche-1 va por el km11 de la carretera
El coche-1 va por el km12 de la carretera
        ***El coche-0 está atravesando el km 7 del puente
        ***El coche-0 está atravesando el km 8 del puente
        ***El coche-0 está atravesando el km 9 del puente
El coche-0 va por el km10 de la carretera
El coche-0 va por el km11 de la carretera
El coche-0 va por el km12 de la carretera
```

10. Crea un programa en el que haya 3 hilos y el principal. Cada uno de estos 3 hilos tiene un NAME diferente que varía desde el “1” hasta el “3”. Cada uno realiza la siguiente tarea:
(Considerando X el nombre del hilo:)
    1. Pinta por pantalla: “Soy el hilo” + X + “ los primeros múltiplos son:”.
    2. El hilo X pinta los primeros 5 números que son múltiplos de X en diferentes líneas por pantalla.
    3. Pinta por pantalla: “Fin de los múltiplos de” + X.

    Controla que no se mezclen las tablas de números divisibles utilizando un objeto compartido entre los hilos.

```
Soy el hilo 1 los primeros múltiplos son:
Múltiplo[1]: 1
Múltiplo[2]: 2
Múltiplo[3]: 3
Múltiplo[4]: 4
Múltiplo[5]: 5
Fin de los múltiplos de 1
Soy el hilo 2 los primeros múltiplos son:
Múltiplo[1]: 2
Múltiplo[2]: 4
Múltiplo[3]: 6
Múltiplo[4]: 8
Múltiplo[5]: 10
Fin de los múltiplos de 2
Soy el hilo 3 los primeros múltiplos son:
Múltiplo[1]: 3
Múltiplo[2]: 6
Múltiplo[3]: 9
Múltiplo[4]: 12
Múltiplo[5]: 15
Fin de los múltiplos de 3
```

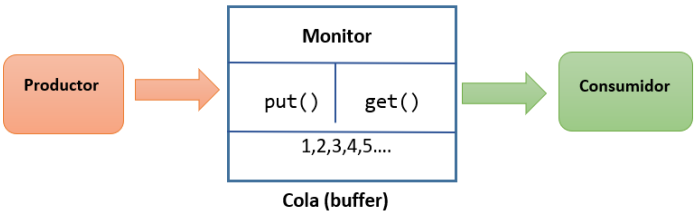
11. Imagina que tienes que crear una aplicación para controlar la publicidad que se reparte por tu ciudad. La empresa al inicio del día tiene una pila de 1000 flyers para repartir. La empresa tiene 5 repartidores, que irán cogiendo de uno en uno de la pila flyers para repartir. Simula el sistema para que siempre se entreguen 1000 y sólo 1000 flyers al final del día. Controla cuántos ha entregado cada empleado y realiza el recuento total desde el Principal una vez hayan terminado todos los Repartidores.

```
El repartidor Juan ha repartido 362 flyers.
El repartidor Marta ha repartido 166 flyers.
El repartidor Pedro ha repartido 127 flyers.
El repartidor María ha repartido 112 flyers.
El repartidor lucía ha repartido 233 flyers.
La suma de repartidos entre todos es: 1000
Al final quedan 0 flayers.
```

2.6.1. Comunicación entre hilos: 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 (**`put()`**), estos serán consumidos por el consumidor (**`get()`**). El recurso a compartir es la Cola con los números.

Para obtener un funcionamiento correcto, 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.

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 (*notify*) 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 (*wait*) 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():

- **`wait():`** le indica al hilo en curso que abandone el monitor y se vaya a dormir hasta que otro hilo entre en el mismo monitor y llame a notify().
- **`notify():`** despierta al primer hilo que realizó una llamada a wait() sobre el mismo objeto.
- **`notifyAll():`** despierta todos los hilos que realizaron una llamada a wait() sobre el mismo objeto. El hilo con mayor prioridad de los despertados es el primero en ejecutarse.

La forma correcta de invocar el método `wait()`, es dentro de un bucle `while()`. Así el hilo se bloqueará hasta que otro hilo le haga una llamada a `notify()` o `notifyAll()`. Sin embargo, hay que volver a hacer la comprobación de la condición, porque la notificación no significará necesariamente que se haya cumplido la condición. Por eso es necesario tener la llamada a wait() dentro de un bucle while que comprueba la condición. Si durante la espera se recibe la excepción `InterruptedException`, aún así lo que importa es comprobar, una vez más, la condición.

```
package es.iesvjp.psp.hilos.ProductorConsumidor;

public class Cola {
	private int numero;
	private boolean llena=false;//inicialmente la cola está vacía

	public synchronized void put(int valor)
	{ //esperamos mientras la cola esté llena, cuando se vacíe pondremos un
	                                                                    valor
		while(llena)
		{
			try {
				wait();
			} catch (InterruptedException e) {
				e.printStackTrace();
			}
		}
		this.numero=valor;//colocamos valor en la cola
		llena=true;
		System.out.println("=>Productor, produce:"+ valor);
		notifyAll();

	}

	public synchronized int get()
	{ //esperamos mientras la cola esté vacía, cuando esté llena cogeremos un
	                                                                    valor
		while(!llena)
		{
			try {
				wait();
			} catch (InterruptedException e) {

				e.printStackTrace();
			}
		}
		llena=false;
		System.out.println("<=Consumidor,consume:"+ this.numero);
		notifyAll();
		return this.numero;

	}
}
```

```
package es.iesvjp.psp.hilos.ProductorConsumidor;

public class Consumidor extends Thread {
	private Cola cola;

	public Consumidor(Cola cola) {
		this.cola = cola;
		}

	public void run() {
		for (int i = 1; i <= 5; i++) {
			int valor = cola.get(); // recoge el número

			try {
				sleep(1000);
			} catch (InterruptedException e) {

				e.printStackTrace();
			}
		}
	}
}
```

```
package es.iesvjp.psp.hilos.ProductorConsumidor;

public class Productor extends Thread {
	private Cola cola;

	public Productor(Cola cola) {
		this.cola = cola;
		}
	public void run() {
		for (int i = 1; i <= 5; i++) {
			cola.put(i); // pone el número
			try {
				sleep(1000);
			} catch (InterruptedException e) {
				e.printStackTrace();
			}


		}
	}
}
```

```
package es.iesvjp.psp.hilos.ProductorConsumidor;

public class Principal {
	public static void main(String[] args) {
		Cola cola= new Cola();
		Productor prod=new Productor(cola);
		Consumidor con=new Consumidor(cola);

		prod.start();
		con.start();
	}
}
```

NOTA: Los métodos **`wait(), notify,`** y **`notifyAll()`** requieren dos condiciones:

- Que el método donde se utilicen tenga el modificador **`synchronized`.**
- Que se capture (**`try..catch`**) la excepción **`InterruptedException`** en la llamada a **`wait().`**

CONSEJOS:

- Utilizar siempre un bucle `while` en lugar de un if para volver a comprobar que se cumple la condición cuando el hilo es puesto de nuevo en ejecución.
- Conviene utilizar `notifyAll()` en lugar de `notify` para la mayoría de las aplicaciones

ACTIVIDADES

12. A partir del ejemplo anterior realiza las modificaciones necesarias para que el productor vaya produciendo números de uno en uno y una serie de consumidores vayan consumiendo dichos números. El productor no podrá producir más de un número a la vez y los consumidores no podrán consumir nada si no se ha producido antes.

13. Realiza una simulación de dos hilos que pinten por pantalla "TIC" y "TAC" simulando el sonido de un reloj. Tienes que controlar, de alguna manera, que nunca un hilo pinta por pantalla dos veces seguidas. En cuyo caso, el funcionamiento del reloj sería erróneo.

Consejos:
Utiliza dos clases que extiendan de Thread, una para el "TIC" y otra para el "TAC".
Ambos hilos tendrán que compartir un objeto (token o testigo). Cada hilo, de manera sincronizada por el token, realizará estos pasos:

1. Comprobará si es su turno, en caso afirmativo, pinta "TIC" ó "TAC" correspondientemente y actualizará el turno del token para el otro.
2. Si no es su turno, no realizará nada.

14. La industria relojera está de enhorabuena, acaban de salir los iRelojes, lo último en tecnología. Estos relojes son más rápidos y precisos gracias a su triple hilo de cronometración. Ya no sólo cuentan TIC y TAC, ahora hacen TIC, TAC y TOC, con lo que se logra hasta un 50% más de rendimiento. En este ejercicio tendrás que simular cada uno de los sonidos del reloj, que será puesto en pantalla con un hilo diferente. La sincronización de los hilos se realizará mediante las primitivas: wait(),notify() y notifyAll()

    NOTA: El orden de ejecución es TIC luego TAC TOC

15. Imagínate que quieres simular un pintor de cadenas de texto. Cada letra de la cadena se pinta con un hilo diferente de tu programa. El programa sólo pinta las letras "A" y "B".

Controla tu programa para que se cumpla que:

- Se puedan pintar tantas 'A' seguidas como se quiera.
- Sólo se puede pintar una 'B' después de haber pintado una 'A'.
- Sólo se pintan 20 letras en total. Pon de tu parte para que las A's y B' se mezclen.

Ejemplos:

- AAAAAAAAABAAABABABAA es una cadena válida
- AABBAAABAAABB Es una cadena inválida. (tamaño y B's repetidas).

Ejemplos de ejecución:

```
AABABAABABABAABABABA
--------------------
A's: 12
B's: 8
```

```
AABABABAABABABAABAA
--------------------
A's: 13
B's: 7
```

```
AABAABABABABABABABAB
--------------------
A's: 11
B's: 9
```

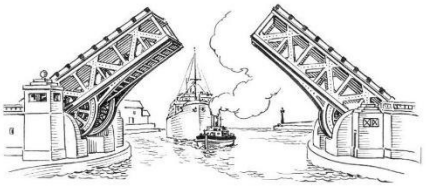
Sincroniza los procesos mediante un objeto **Controlador**.

16. El instituto *IES Valle del Jerte* está pasando por momentos económicos duros. La Directora nos ha pedido que reduzcamos los cuartos de baño de todo el Centro a una planta.

Para llevar a cabo esta medida de ahorro es necesario que se controle que no más de tres personas estén a la vez en el cuarto de baño.
Cada persona tarda 20ms (como mucho) entre que entra y sale del baño.
Después de ir al baño cada persona hace una espera de 0,5 segundos.
Realiza la simulación para 5 personas y pinta por pantalla la entrada y salida de personas del cuarto de baño.

```
Entra Juan---------- Hay 1
Entra Marcos---------- Hay 2
Entra Ana---------- Hay 3
¡Pedro espera porque el baño está lleno!
¡María espera porque el baño está lleno!
        Juan permanece en el baño 18 milisegundos.
        Marcos permanece en el baño 5 milisegundos.
        Ana permanece en el baño 20 milisegundos.
                Sale Marcos---------- Hay 2
Entra María---------- Hay 3
        María permanece en el baño 12 milisegundos.
¡Pedro espera porque el baño está lleno!
                Sale Juan---------- Hay 2
Entra Pedro---------- Hay 3
                Sale María---------- Hay 2
        Pedro permanece en el baño 4 milisegundos.
                Sale Ana---------- Hay 1
                Sale Pedro---------- Hay 0
Entra Marcos---------- Hay 1
        Marcos permanece en el baño 19 milisegundos.
Entra María---------- Hay 2
        María permanece en el baño 18 milisegundos.
Entra Juan---------- Hay 3
        Juan permanece en el baño 13 milisegundos.
¡Ana espera porque el baño está lleno!
¡Pedro espera porque el baño está lleno!
                Sale Marcos---------- Hay 2
Entra Pedro---------- Hay 3
        Pedro permanece en el baño 8 milisegundos.
¡Ana espera porque el baño está lleno!
                Sale Juan---------- Hay 2
```

17. El alcalde de nuestra ciudad nos pide ayuda para controlar un puente levadizo que se va a instalar en unos pocos meses.



El puente está preparado para que pasen peatones por él. Suponemos que el puente es lo suficientemente ancho como para que pasen tantos viandantes como se quiera a la vez.

La característica especial de este puente es que se eleva para permitir el paso de barcos. Por motivos de seguridad, el puente SÓLO puede elevarse cuando no haya peatones cruzando el puente

Cada vez que un barco se vaya a disponer a entrar, ningún peatón va a poder acceder al puente. Aquellos que todavía estén dentro van a salir ordenadamente antes de elevar el puente. Es decir, los barcos tienen prioridad siempre excepto en el caso en el que ya haya peatones en el puente, en el que esperaremos a que salgan.

Programa una aplicación concurrente que simule el paso de peatones y barcos siguiendo las características mencionadas en el enunciado. Para ello utiliza:

- 5 hilos que representen peatones. Van a pasar por el puente 5 veces en cada ejecución de la simulación.
- 1 hilo barco que va a cruzar el puente 5 veces.

Posible salida por pantalla:

```
Entra María por el puente. Hay 1
        María está en el puente
Entra Ana por el puente. Hay 2
        Ana está en el puente
Entra Marcos por el puente. Hay 3
        Marcos está en el puente
                =>Sale Marcos del puente. Quedan 2
Entra Juan por el puente. Hay 3
        Juan está en el puente
Entra Pedro por el puente. Hay 4
        Pedro está en el puente
                =>Sale Juan del puente. Quedan 3
Entra Marcos por el puente. Hay 4
        Marcos está en el puente
                =>Sale Ana del puente. Quedan 3
                =>Sale María del puente. Quedan 2
*************** BARCO QUIERE PASAR ***************
                =>Sale Marcos del puente. Quedan 1
                =>Sale Pedro del puente. Quedan 0
        Titanic está cruzando....<__P__>
                El barco Titanic salió del puente.
Entra Pedro por el puente. Hay 1
        Pedro está en el puente
Entra Marcos por el puente. Hay 2
        Marcos está en el puente
Entra María por el puente. Hay 3
        María está en el puente
```

18. Los alumnos **fumadores** de 2º DAM se pasan el día liando cigarros y fumando. Como entre clase y clase no tienen tiempo se esconden en el **baño** para fumar, pero solamente puede entrar un alumno a fumar al baño. Para liar un cigarro necesitan tres ingredientes: *tabaco, papel y cerillas*. Cada fumador dispone de un surtido ilimitado de uno de los tres ingredientes. Cada fumador tiene un ingrediente diferente, es decir, un fumador tiene una cantidad infinita de tabaco, el otro de papel y el otro de cerillas. En el cuarto de baño se esconde otro alumno (**estanquero**) que dispone de dos de los tres ingredientes. El estanquero dispone de unas reservas infinitas de cada uno de los tres ingredientes y escoge de forma aleatoria cuáles son los ingredientes que pondrá encima de la repisa del baño. Cuando los ha puesto, el fumador que tiene el otro ingrediente puede fumar, el resto de los fumadores no. Para ello coge los ingredientes, se lía un cigarro y se lo fuma, una vez que se lo fuma abandona el baño. Cuando termina de fumar vuelve a repetirse el ciclo. En resumen, el ciclo que debe repetirse es:

"estanquero pone ingredientes□ fumador hace cigarro□ fumador fuma□ fumador termina de fumar → estanquero pone ingredientes → ..."

Es decir, en cada momento hay solamente un fumador fumando un cigarrillo, el resto esperará a que termine y mientras la repisa esté llena o haya alguien fumando el estanquero también esperará.

Consejos:

- Para simplificar el problema considera que los ingredientes son tres enteros 1, 2 y 3.
- Representa cada fumador por el número de ingrediente que tiene, es decir, el fumador 1 posee el ingrediente 1.
- En el baño, el estanquero pone dos ingredientes más, por lo que la repisa se identificará con el entero del ingrediente que falta, es decir, si repisa = 1 significa que en la repisa están los ingredientes 2 y 3, pero no el 1. Para representar que la repisa está vacía ponemos la variable repisa a 0.

```
El fumador 1 tiene tabaco, necesita CERILLAS Y PAPEL
El fumador 2 tiene cerillas, necesita TABACO Y PAPEL
El fumador 3 tiene papel, necesita TABACO Y CERILLAS
        =>El estanquero repone ingredientes:en la repisa hay TABACO Y CERILLAS
        El fumador 3 ya tiene todos los ingredientes. Empieza a fumar
        ==~~ ==~~ Fumador 3 fumando ==~~ ==~~
                El fumador 3 ya terminó de fumar. Abandona el baño
El fumador 3 tiene papel, necesita TABACO Y CERILLAS
        =>El estanquero repone ingredientes:en la repisa hay TABACO Y PAPEL
        El fumador 2 ya tiene todos los ingredientes. Empieza a fumar
        ==~~ ==~~ Fumador 2 fumando ==~~ ==~~
                El fumador 2 ya terminó de fumar. Abandona el baño
El fumador 2 tiene cerillas, necesita TABACO Y PAPEL
        =>El estanquero repone ingredientes:en la repisa hay CERILLAS Y PAPEL
        El fumador 1 ya tiene todos los ingredientes. Empieza a fumar
        ==~~ ==~~ Fumador 1 fumando ==~~ ==~~
                El fumador 1 ya terminó de fumar. Abandona el baño
El fumador 1 tiene tabaco, necesita CERILLAS Y PAPEL
        =>El estanquero repone ingredientes:en la repisa hay TABACO Y CERILLAS
        El fumador 3 ya tiene todos los ingredientes. Empieza a fumar
        ==~~ ==~~ Fumador 3 fumando ==~~ ==~~
                El fumador 3 ya terminó de fumar. Abandona el baño
        =>El estanquero repone ingredientes:en la repisa hay TABACO Y CERILLAS
El fumador 3 tiene papel, necesita TABACO Y CERILLAS
        El fumador 3 ya tiene todos los ingredientes. Empieza a fumar
        ==~~ ==~~ Fumador 3 fumando ==~~ ==~~
                El fumador 3 ya terminó de fumar. Abandona el baño
        =>El estanquero repone ingredientes:en la repisa hay TABACO Y PAPEL
        El fumador 2 ya tiene todos los ingredientes. Empieza a fumar
        ==~~ ==~~ Fumador 2 fumando ==~~ ==~~
El fumador 3 tiene papel, necesita TABACO Y CERILLAS
                El fumador 2 ya terminó de fumar. Abandona el baño
El fumador 2 tiene cerillas, necesita TABACO Y PAPEL
        =>El estanquero repone ingredientes:en la repisa hay CERILLAS Y PAPEL
        El fumador 1 ya tiene todos los ingredientes. Empieza a fumar
        ==~~ ==~~ Fumador 1 fumando ==~~ ==~~
```

EJERCICIOS DE AUTOEVALUACIÓN

1. Diga si las afirmaciones son verdaderas (V) o falsas (F):

Pregunta	V	F
Una clase no puede tener métodos sincronizados y no sincronizados.		
A un método no sincronizado pueden acceder muchos hilos a la vez.		
Si un hilo ejecuta el método `wait()`, pierde el bloqueo del monitor en el caso de que tenga.		
Si un hilo ejecuta el método `sleep()`, pierde el bloqueo del monitor en caso de que lo tenga.		
`wait()`, `notify()` y `notifyAll()` son métodos que siempre se ejecutan en bloques de código sincronizado.		

2. Completa el siguiente enunciado con las palabras que se proporcionan más abajo:

   Si dos …….. quieren ejecutar un método ……. de una clase y los dos usan la misma …….. de la clase para llamar al método, únicamente uno de los dos podrá ……el código en un momento determinado. El otro hilo deberá …….que el primer termine su ejecución.

   hilos, sincronizado, instancia, ejecutar, esperar

3. Completa el siguiente enunciado con las palabras que se proporcionan más abajo:

   Para garantizar ……. tendremos que garantizar que en una ……. no puede haber más …….. en ejecución. Los …….. que no se encuentran en la región crítica no pueden ……. otros procesos. Todos los procesos que quieren entrar a …….. la región crítica esperan un tiempo……..…

   bloquear, exclusión mutua, procesos, de un proceso, ejecutar, limitado, sección crítica

4. Relaciona los conceptos correctamente:

1. La palabra reservada que permite sincronizar métodos es.	a. interfaz
2. `Runnable` es una.	b. en espera
3. El método `yield()` envía el hilo en el estado de…	c. interbloqueo
4.Cuando dos hilos están ejecutando concurrentemente y cada uno tiene un bloqueo exclusivo que el otro necesita para continuar se produce.	d. preparado
5. El método `wait()` envía al hilo en el estado de	e. `synchronized`

5. Diga si las afirmaciones son verdaderas (V) o falsas (F):

Pregunta	V	F
Un proceso siempre tiene un estado		
Un proceso y una aplicación son sinónimos		
El encargado de dar a los procesos la oportunidad de usar la CPU es el planificador de procesos o scheduler		
Dos procesos serán secuenciales cuando la primera instrucción de uno de ellos se ejecuta después de la primera instrucción del otro y antes de la última		

6. Indica qué partes del código podemos sincronizar en Java:

Pregunta	Respuesta
Métodos de una clase	
Variables	
Bloques de código	
Una clase	