

# UT2. Programación MultiHILO

## Tabla de contenido

<b>1.</b>	<b>Introducción .....</b>	<b>2</b>
<b>2.</b>	<b>Programación secuencial frente a programación concurrente. ....</b>	<b>3</b>
<b>2.1</b>	<b>Programación secuencial o de único hilo.....</b>	<b>3</b>
<b>2.2</b>	<b>Programación concurrente o multihilo.....</b>	<b>5</b>
<b>3.</b>	<b>Estados de un Hilo.....</b>	<b>10</b>
<b>4.</b>	<b>Problemas de Conurrencia.....</b>	<b>12</b>
<b>4.1</b>	<b>PROBLEMAS DE SINCRONIZACIÓN .....</b>	<b>15</b>
<b>4.2</b>	<b>MECANISMOS DE COMUNICACIÓN .....</b>	<b>16</b>
<b>5.</b>	<b>Actividades Guiadas .....</b>	<b>17</b>
<b>5.1</b>	<b>INTERRUPCIONES .....</b>	<b>17</b>
<b>5.2</b>	<b>COMPARTICIÓN DE INFORMACIÓN .....</b>	<b>18</b>
<b>5.3</b>	<b>SINCRONIZACIÓN BÁSICA: WAIT, NOTIFY, NOTIFYALL.....</b>	<b>20</b>
PRODUCTOR-CONSUMIDOR .....	22	
Desafíos:.....	22	
<b>5.4</b>	<b>SINCRONIZACIÓN BÁSICA: MÉTODO JOIN.....</b>	<b>25</b>
<b>5.5</b>	<b>SINCRONIZACIÓN AVANZADA: SEMÁFOROS .....</b>	<b>27</b>
<b>6.</b>	<b>Actividades Propuestas.....</b>	<b>29</b>

## 1. Introducción

---

Un **HILo**, también conocido como **THREAD**, es una pequeña unidad de computación que se ejecuta dentro del contexto de un proceso. Todos los programas utilizan hilos.

La ejecución de un proceso comienza con un único hilo, pero se pueden crear más sobre la marcha. Los distintos hilos de un mismo proceso comparten:

- Espacio de memoria asignado al proceso.
- Información de acceso a ficheros. Estos se utilizan no solo para almacenar datos, sino también para controlar dispositivos de entrada y salida (E/S).

En cambio, cada hilo tiene sus propios valores para:

- Registros del procesador.
- Estado de su pila (stack) que es una estructura de datos utilizada para almacenar información sobre las subrutinas activas, las variables locales y el contexto de ejecución del hilo1. La pila de un hilo incluye:
  - **Direcciones de retorno:** Indican dónde debe continuar la ejecución después de que una subrutina termina.
  - **Variables locales:** Almacenan datos temporales que solo son relevantes dentro del contexto de una subrutina.
  - **Contexto de ejecución:** Incluye registros del procesador y otros datos necesarios para reanudar la ejecución del hilo correctamente.

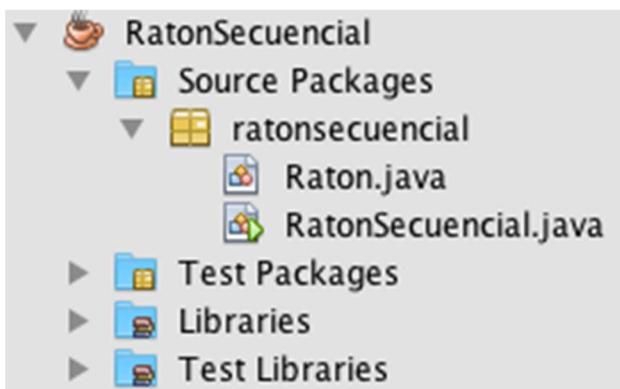
Este aislamiento permite que los hilos se ejecuten de manera independiente y asíncrona, aunque comparten el mismo espacio de memoria y otros recursos del proceso principal.

A continuación se citan algunas de las características que tienen los hilos:

- **Dependencia del proceso.** No se pueden ejecutar independientemente. Siempre se tienen que ejecutar dentro del contexto de un proceso.
- **Ligereza.** Al ejecutarse dentro del contexto de u proceso, no requiere generar procesos nuevos, por lo que son óptimos desde el punto de vista del uso de recursos. Se pueden generar gran cantidad de hilos sin que provoquen pérdidas de memoria.
- **Compartición de recursos.** Dentro de un mismo proceso, los hilos comparten espacio de memoria. Esto implica que pueden sufrir colisiones en los accesos a las variables provocando errores de concurrencia.
- **Paralelismo.** Aprovechan los núcleos del procesador generando un paralelismo real, siempre dentro de las capacidades del procesador.
- **Concurrencia.** Permiten atender de manera concurrente múltiples peticiones. Esto es especialmente importante en servidores web y bases de datos, por ejemplo.

## 2. Programación secuencial frente a programación concurrente.

### 2.1 Programación secuencial o de único hilo



El siguiente ejemplo ilustra cómo se comporta un programa que se ejecuta en un único hilo, así como las consecuencias que esto implica. El programa está compuesto por una única clase (representa un ratón) compuesta por dos atributos: el nombre y el tiempo en segundos que tarda en comer. En el método main se instancian varios objetos (ratones) y se invoca al método comer de cada uno de ellos. Este método muestra un texto por pantalla cuando comienza, realiza una pausa de la duración en segundos (con el método sleep de la clase Thread) que indica el parámetro tiempoAlimentación y, finalmente, muestra otro texto por pantalla cuando finaliza.

```
package ratonsecuencial;

public class Raton {

    private String nombre;
    private int tiempoAlimentacion;

    public Raton(String nombre, int tiempoAlimentacion) {
        this.nombre=nombre;
        this.tiempoAlimentacion=tiempoAlimentacion;
    }

    public void comer(){
        try{
            System.out.println("El raton"+this.nombre+"ha comenzado a alimentarse");
            Thread.sleep(this.tiempoAlimentacion*1000);
            System.out.println("El raton"+this.nombre+"ha terminado de alimentarse");
        }
        catch(InterruptedException e){
            e.printStackTrace();
        }
    }
}
```

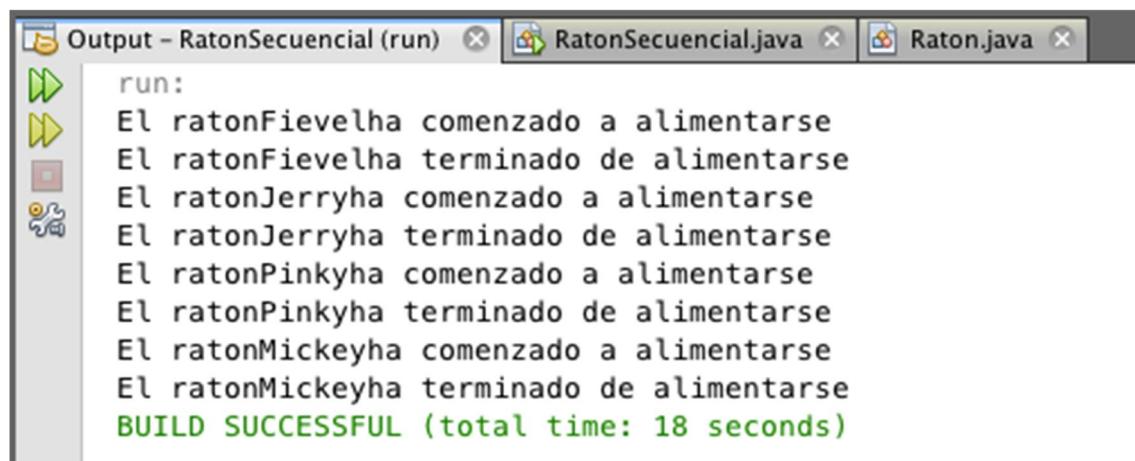
**DESARROLLO DE APLICACIONES MULTIPLATAFORMA**  
**PROGRAMACIÓN DE SERVICIOS Y PROCESOS**

```
package ratonsecuencial;

public class RatonSecuencial {

    public static void main(String[] args) {
        Raton fievel = new Raton("Fievel", 4);
        Raton jerry = new Raton("Jerry", 5);
        Raton pinky = new Raton("Pinky", 3);
        Raton mickey = new Raton("Mickey", 6);
        fievel.comer();
        jerry.comer();
        pinky.comer();
        mickey.comer();
    }
}
```

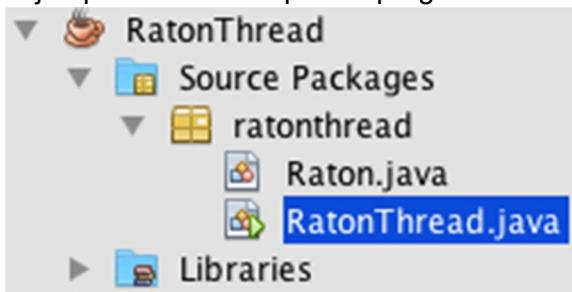
La salida producida por la ejecución es la siguiente:



```
Output - RatonSecuencial (run) × RatonSecuencial.java × Raton.java ×
run:
El ratonFievelha comenzado a alimentarse
El ratonFievelha terminado de alimentarse
El ratonJerryha comenzado a alimentarse
El ratonJerryha terminado de alimentarse
El ratonPinkyha comenzado a alimentarse
El ratonPinkyha terminado de alimentarse
El ratonMickeyha comenzado a alimentarse
El ratonMickeyha terminado de alimentarse
BUILD SUCCESSFUL (total time: 18 seconds)
```

## 2.2 Programación concurrente o multihilo

El ejemplo anterior se puede programar de manera concurrente de forma sencilla, ya



que no hay ningún recurso compartido. Para ello, **basta con convertir cada objeto de la clase Raton en un hilo (thread) y programar aquello que se quiere que ocurra concurrentemente dentro del método run**. Una vez creadas las instancias, se invoca al método start de cada una de ellas, lo que provoca la ejecución del contenido del método run en un hilo independiente.

```
package ratonthread;

public class Raton extends Thread{

    private String nombre;
    private int tiempoAlimentacion;

    public Raton(String nombre, int tiempoAlimentacion) {
        this.nombre=nombre;
        this.tiempoAlimentacion=tiempoAlimentacion;
    }

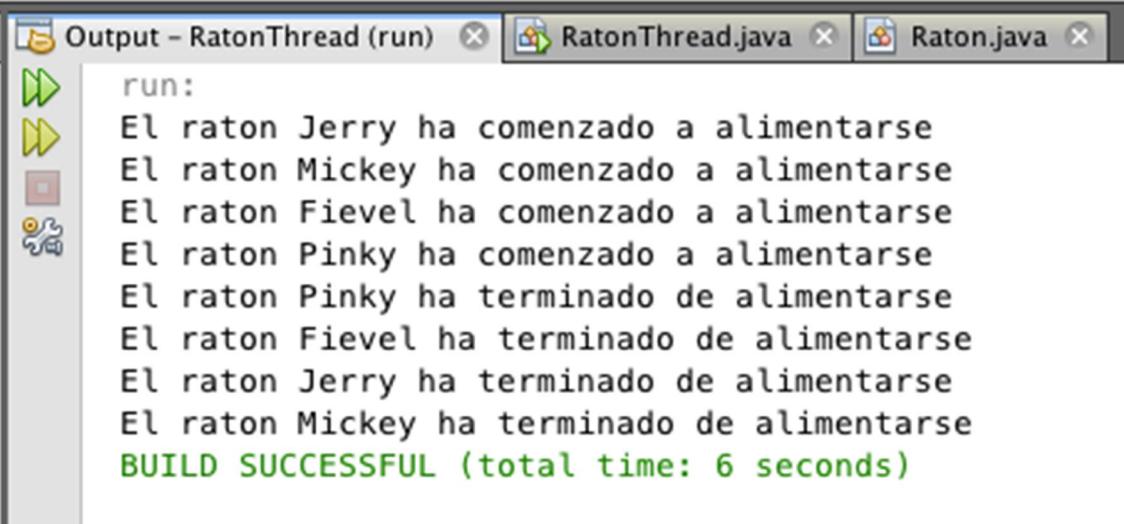
    public void comer(){
        try{
            System.out.println("El raton "+this.nombre+
                " ha comenzado a alimentarse");
            Thread.sleep(this.tiempoAlimentacion*1000);
            System.out.println("El raton "+this.nombre+
                " ha terminado de alimentarse");
        }
        catch(InterruptedException e){
            e.printStackTrace();
        }
    }
    public void run(){ this.comer(); }
}
```

```
package ratonthread;

public class RatonThread {

    public static void main(String[] args) {
        Raton fievel = new Raton("Fievel", 4);
        Raton jerry = new Raton("Jerry", 5);
        Raton pinky = new Raton("Pinky", 3);
        Raton mickey = new Raton("Mickey", 6);
        fievel.start();
        jerry.start();
        pinky.start();
        mickey.start();
    }
}
```

La salida producida por la ejecución es la siguiente:



```
Output - RatonThread (run) × RatonThread.java × Raton.java ×
run:
El raton Jerry ha comenzado a alimentarse
El raton Mickey ha comenzado a alimentarse
El raton Fievel ha comenzado a alimentarse
El raton Pinky ha comenzado a alimentarse
El raton Pinky ha terminado de alimentarse
El raton Fievel ha terminado de alimentarse
El raton Jerry ha terminado de alimentarse
El raton Mickey ha terminado de alimentarse
BUILD SUCCESSFUL (total time: 6 seconds)
```

Todos los ratones han comenzado a alimentarse de inmediato, sin esperar a que termine ninguno de los demás. El tiempo total del proceso será, aproximadamente, el tiempo del proceso más lento. La reducción del tiempo total de ejecución es evidente.

En java existen dos formas para la creación de hilos:

- Implementando la interface `java.lang.Runnable`
- Heredando de la clase `java.lang.Thread`

En Java, es generalmente preferible implementar la interfaz Runnable en lugar de extender la clase Thread. Runnable ofrece mayor **Flexibilidad** ya que puede extender otra clase si es necesario: recordar que Java no permite la herencia múltiple: si extiendes Thread, no podrás heredar de ninguna otra clase.

**DESARROLLO DE APLICACIONES MULTIPLATAFORMA**  
**PROGRAMACIÓN DE SERVICIOS Y PROCESOS**

Aquí tienes un ejemplo de cada enfoque:

**Ejemplo implementando la interface Runnable**

```
public class MiRunnable implements Runnable {  
    @Override  
    public void run() {  
        System.out.println("El hilo está corriendo");  
    }  
  
}  
  
  
public class Main {  
    public static void main(String[] args) {  
        MiRunnable miRunnable = new MiRunnable();  
        Thread hilo = new Thread(miRunnable);  
        hilo.start();  
    }  
}
```

**Ejemplo heredando de la clase Thread**

```
public class MiHilo extends Thread {  
    @Override  
    public void run() {  
        System.out.println("El hilo está corriendo");  
    }  
  
}  
  
  
public class Main {  
    public static void main(String[] args) {  
        MiHilo miHilo = new MiHilo();  
        miHilo.start();  
    }  
}
```

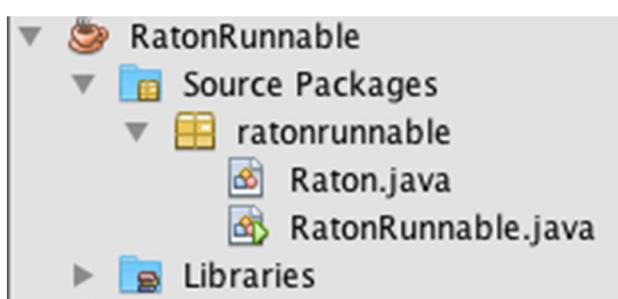
## DESARROLLO DE APLICACIONES MULTIPLATAFORMA

### PROGRAMACIÓN DE SERVICIOS Y PROCESOS

La clase Thread en sí no está deprecada en Java. Sin embargo, algunos métodos de la clase Thread sí han sido deprecados debido a problemas de seguridad y fiabilidad. Por ejemplo:

- Thread.stop(): Este método fue deprecado porque puede dejar los objetos en un estado inconsistente al liberar todos los monitores que el hilo ha bloqueado.
- Thread.suspend() y Thread.resume(): Estos métodos también fueron deprecados porque pueden causar deadlocks si el hilo suspendido mantiene un bloqueo sobre un recurso crítico.

En lugar de estos métodos, se recomienda utilizar mecanismos de control más seguros, como la interrupción de hilos con Thread.interrupt() y el uso de variables de estado para gestionar la ejecución de los hilos.



```
package ratonrunnerable;

public class Raton implements Runnable{

    private String nombre;
    private int tiempoAlimentacion;

    public Raton(String nombre, int tiempoAlimentacion) {
        this.nombre=nombre;
        this.tiempoAlimentacion=tiempoAlimentacion;
    }

    public void comer(){
        try{
            System.out.println("El raton "+this.nombre+
                " ha comenzado a alimentarse");
            Thread.sleep(this.tiempoAlimentacion*1000);
            System.out.println("El raton "+this.nombre+
                " ha terminado de alimentarse");
        }
        catch(InterruptedException e){
            e.printStackTrace();
        }
    }

    public void run(){ this.comer(); }
}
```

Retomando el enunciado del ejemplo de los objetos de la clase Ratón, la solución mediante implementación de la interface Runnable tendría el código que se muestra a continuación:

```
package ratorunnable;

public class RatorRunnable {

    public static void main(String[] args) {
        Rator fievel = new Rator("Fievel", 4);
        Rator jerry = new Rator("Jerry", 5);
        Rator pinky = new Rator("Pinky", 3);
        Rator mickey = new Rator("Mickey", 6);
        new Thread(fievel).start();
        new Thread(jerry).start();
        new Thread(pinky).start();
        new Thread(mickey).start();
    }
}
```

La salida producida por la ejecución es la siguiente:

```
Output - RatorRunnable (run) × RatorRunnable.java × Rator.java ×
run:
El raton Jerry ha comenzado a alimentarse
El raton Fievel ha comenzado a alimentarse
El raton Pinky ha comenzado a alimentarse
El raton Mickey ha comenzado a alimentarse
El raton Pinky ha terminado de alimentarse
El raton Fievel ha terminado de alimentarse
El raton Jerry ha terminado de alimentarse
El raton Mickey ha terminado de alimentarse
BUILD SUCCESSFUL (total time: 6 seconds)
```

Es habitual, que en las primeras tomas de contacto con los hilos en Java, los programadores intenten ejecutar el método run en lugar de ejecutar el método start. Desde el punto de vista de la compilación no habrá problema: el programa se compilará sin errores. Desde la perspectiva de la programación concurrente el resultado no será adecuado, ya que los hilos se ejecutarán uno detrás de otro, de manera secuencial, no obteniendo ninguna mejora frente a la programación secuencial convencional.

### 3. Estados de un Hilo

---

Durante el ciclo de vida de los hilos, estos pasan por diversos estados. En java están recogidos dentro de la enumeración State contenida dentro de la clase java.lang.Thread.

El estado de un hilo se obtiene mediante el método getState() de la clase Thread, que devolverá algunos de los valores posibles recogidos en la enumeración indicada anteriormente.

Estados de hilos y su correspondencia en Java.

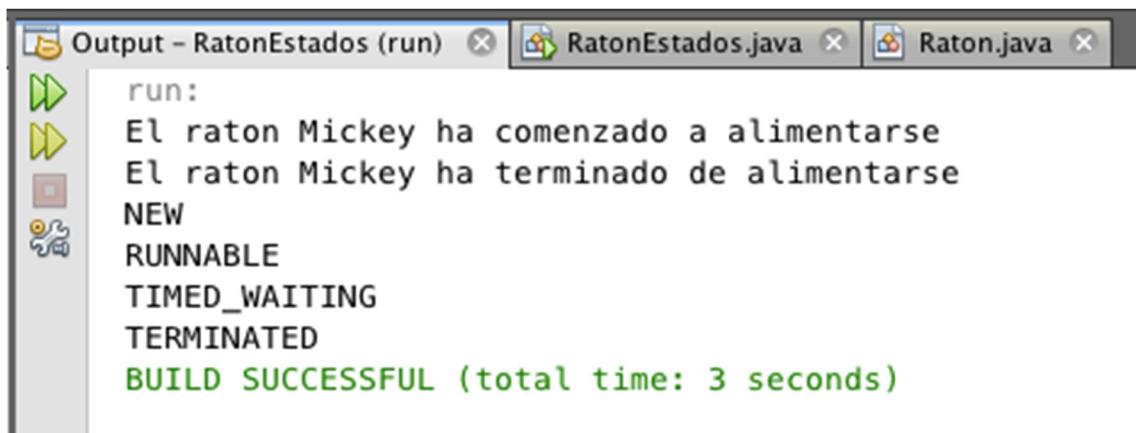
ESTADO	Thread.State	Descripción
Nuevo	NEW	El hilo está creado, pero aún no se ha arrancado.
Ejecutable	RUNNABLE	El hilo está arrancado y podría estar en ejecución o pendiente de ejecución. <b>Cuando un objeto llama al método start() comienza la ejecución del hilo; es decir, se empiezan a ejecutar las instrucciones que se encuentran en el método run().</b>
Bloqueado	BLOCKED	El hilo se ha parado pero puede volver a ejecutarse. Normalmente se bloquea con las siguientes acciones: <ul style="list-style-type: none"><li>• Se invoca al método <b>sleep()</b></li><li>• Queda a la espera de una operación de E/S</li><li>• Se invoca el método <b>wait()</b>. Cuando un hilo queda bloqueado por este método, otro hilo debe invocar el método <b>notify()</b> o <b>notifyAll()</b> para despertar un único hilo o todos aquellos que estén bloqueados, respectivamente.</li><li>• Otro hilo, externamente invoca el método <b>suspend()</b>. Cuando un hilo llama a este método, el hilo bloqueado debe recibir el mensaje <b>resume()</b> para poder volver a la ejecución.</li></ul>
Esperando	WAITING	El hilo está esperando a que otro hilo realice una acción determinada.
Esperando un tiempo	TIME_WAITING	El hilo está esperando a que otro hilo realice una acción determinada en un periodo de tiempo concreto.
Finalizado	TERMINATED	El hilo ha terminado su ejecución.

**DESARROLLO DE APLICACIONES MULTIPLATAFORMA**  
**PROGRAMACIÓN DE SERVICIOS Y PROCESOS**

En el siguiente código de ejemplo, se almacenan en un ArrayList los estados por los que pasa un hilo que contiene en su interior una llamada al método sleep. Se utiliza la clase Raton que implementa Runnable de los ejemplos anteriores.

```
package ratonestados;
import java.util.ArrayList;

public class RatonEstados {
    public static void main(String[] args) {
        Raton mickey = new Raton("Mickey", 3);
        ArrayList<Thread.State> estadosHilo = new ArrayList();
        Thread h = new Thread(mickey);
        estadosHilo.add(h.getState());
        h.start();
        while(h.getState()!=Thread.State.TERMINATED){
            if(!estadosHilo.contains(h.getState())){
                estadosHilo.add(h.getState());
            }
        }
        estadosHilo.add(h.getState());
        for(int i=0; i<estadosHilo.size(); i++)
            System.out.println(estadosHilo.get(i));
    }
}
```

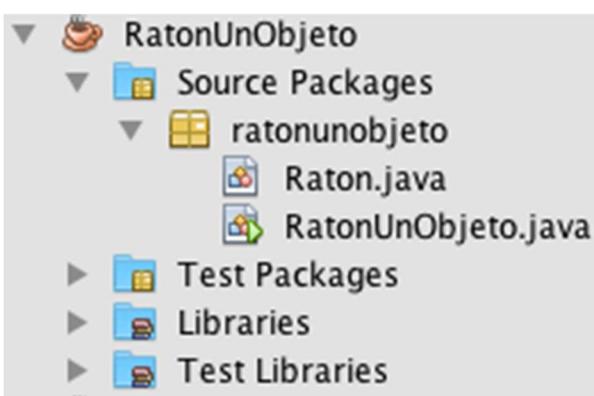


## 4. Problemas de Conurrencia

En programación concurrente las dificultades aparecen cuando los distintos hilos acceden a un recurso compartido limitado.

Un aspecto importante referente a los problemas de concurrencia es que **NO SIEMPRE PROVOCAN UN ERROR EN TIEMPO DE EJECUCIÓN**. Esto significa que en sucesivas ejecuciones del programa multihilo el error se producirá en algunas de ellas y en otras no. Frente al determinismo de los programas secuenciales, en el entorno multihilo los factores que determinan las condiciones de ejecución pueden tener su origen en elementos sobre los que el programador no tiene capacidad de influir, como el sistema operativo.

La dificultad en este tipo de programación, reside en el hecho de que el programador tiene que saber de forma anticipada que el error se puede producir en una sentencia o bloques de sentencias, por cómo se ejecuta y qué recursos utiliza. **No basta con realizar pruebas convencionales para determinar que el algoritmo está bien programado. Hay que SABER que está bien programado.**



En el siguiente ejemplo se implementa un hilo mediante la interface Runnable para crear múltiples hilos a partir de un único objeto. En el hilo, un atributo de instancia llamado alimentoConsumido se incrementa en 1 durante la ejecución del método comer, invocado en el método run. Se puede observar en el método main cómo se crea una única instancia de la clase RatonSimple y se crean cuatro hilos que la ejecutan.

```
package ratonunobjeto;

public class RatonUnObjeto {

    public static void main(String[] args) {
        Raton fievel = new Raton("Fievel", 4);
        new Thread(fievel).start();
        new Thread(fievel).start();
        new Thread(fievel).start();
        new Thread(fievel).start();
    }
}
```

```
package ratonunobjeto;

public class Raton implements Runnable{

    private String nombre;
    private int tiempoAlimentacion;
    private int alimentoConsumido;

    public Raton(String nombre, int tiempoAlimentacion) {
        this.nombre=nombre;
        this.tiempoAlimentacion=tiempoAlimentacion;
        this.alimentoConsumido=0; }

    public void comer(){
        try{
            System.out.println("El raton "+this.nombre+
                " ha comenzado a alimentarse");
            Thread.sleep(this.tiempoAlimentacion*1000);
            this.alimentoConsumido++;
            System.out.println("El raton "+this.nombre+
                " ha terminado de alimentarse");
            System.out.println("Lleva consumidos "+
                this.alimentoConsumido+" alimentos");

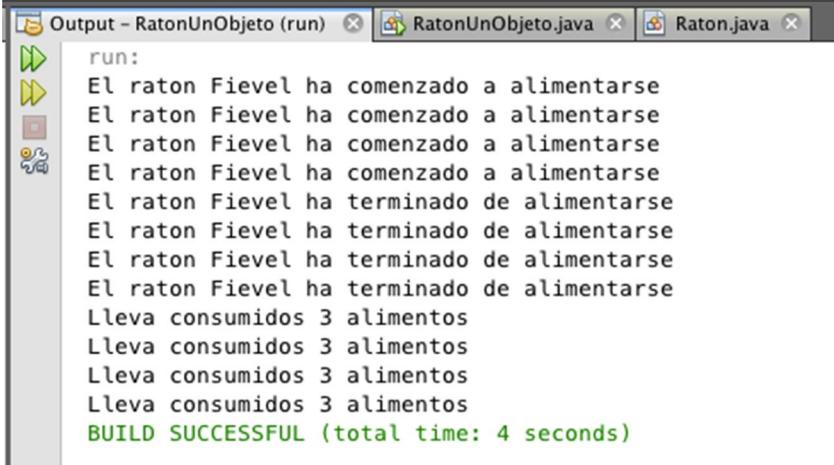
        }
        catch(InterruptedException e){
            e.printStackTrace(); }
    }

    public void run(){ this.comer(); }
}
```

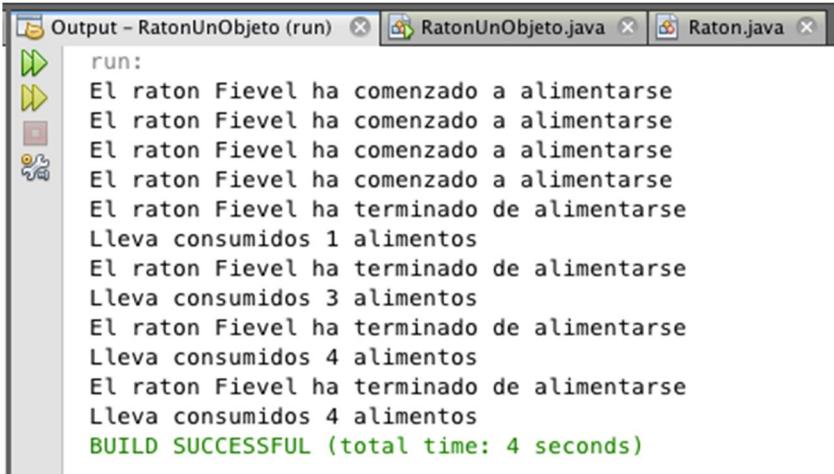
El resultado de la ejecución es el siguiente: cada hilo ha ejecutado el método run sobre los datos del mismo objeto. Es decir, se ha ejecutado simultáneamente cuatro veces un bloque de código de un único objeto, compartiendo sus atributos. De esta forma, en la salida se puede apreciar que el valor del atributo alimentoConsumido se ha incrementado varias veces durante la ejecución. El hecho de que algunos valores intermedios no aparezcan en la siguiente captura de la salida de la ejecución tiene que ver con la asincronía y el alto coste de ejecución que tienen las sentencias que escriben por pantalla.

## DESARROLLO DE APLICACIONES MULTIPLATAFORMA

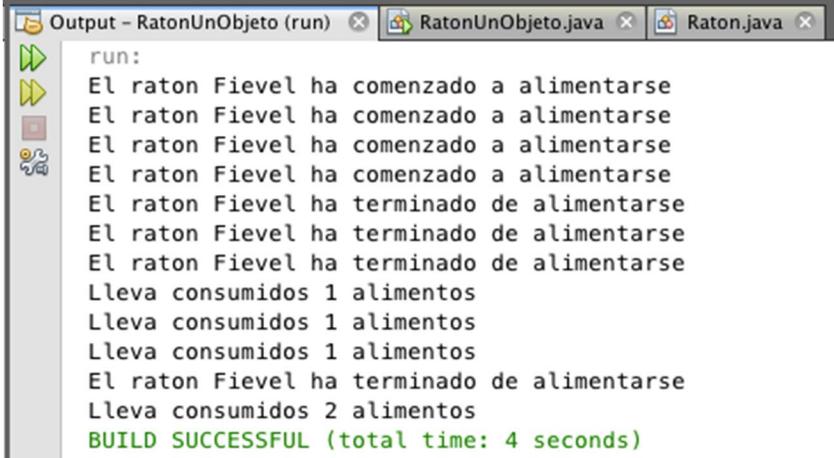
### PROGRAMACIÓN DE SERVICIOS Y PROCESOS



```
Output - RatonUnObjeto (run) RatonUnObjeto.java Raton.java
run:
El raton Fievel ha comenzado a alimentarse
El raton Fievel ha terminado de alimentarse
El raton Fievel ha terminado de alimentarse
El raton Fievel ha terminado de alimentarse
Lleva consumidos 3 alimentos
Lleva consumidos 3 alimentos
Lleva consumidos 3 alimentos
Lleva consumidos 3 alimentos
BUILD SUCCESSFUL (total time: 4 seconds)
```



```
Output - RatonUnObjeto (run) RatonUnObjeto.java Raton.java
run:
El raton Fievel ha comenzado a alimentarse
El raton Fievel ha terminado de alimentarse
Lleva consumidos 1 alimentos
El raton Fievel ha terminado de alimentarse
Lleva consumidos 3 alimentos
El raton Fievel ha terminado de alimentarse
Lleva consumidos 4 alimentos
El raton Fievel ha terminado de alimentarse
Lleva consumidos 4 alimentos
BUILD SUCCESSFUL (total time: 4 seconds)
```



```
Output - RatonUnObjeto (run) RatonUnObjeto.java Raton.java
run:
El raton Fievel ha comenzado a alimentarse
El raton Fievel ha terminado de alimentarse
El raton Fievel ha terminado de alimentarse
El raton Fievel ha terminado de alimentarse
Lleva consumidos 1 alimentos
Lleva consumidos 1 alimentos
Lleva consumidos 1 alimentos
El raton Fievel ha terminado de alimentarse
Lleva consumidos 2 alimentos
BUILD SUCCESSFUL (total time: 4 seconds)
```

Debido a la naturaleza de la programación multihilo, las salidas de las ejecuciones de los programas de ejemplo pueden no coincidir con las anteriores. De hecho, en distintas ejecuciones en la misma máquina los resultados pueden variar, tal como se observa en el ejemplo mostrado arriba.

## 4.1 PROBLEMAS DE SINCRONIZACIÓN

Cuando varios hilos comparten el mismo espacio de memoria es posible que aparezcan algunos problemas, denominados **PROBLEMAS DE SINCRONIZACIÓN**:

- **Condición de carrera:** se denomina condición de carrera a la ejecución de un programa en la que su salida depende de la secuencia de eventos que se produzcan.

*Imagina que 2 hilos están intentando incrementar a la vez una misma variable. El resultado dependerá de la secuencia en que se ejecute el programa:*

Variable vale 2 Hilo 1 lee variable: 2 Hilo 2 lee variable: 2 Hilo 1 suma 1 a variable: 3 Hilo 2 suma 1 a variable: 3 Variable vale 3	Variable vale 2 Hilo 1 lee variable: 2 Hilo 1 suma 1 a variable: 3 Hilo 2 lee variable: 3 Hilo 2 suma 1 a variable: 4 Variable vale 4
--	--

- **Inconsistencia de memoria:** es aquel problema en el que los hilos, que comparten un dato en memoria, ven diferentes valores para el mismo elemento.

*Imagina que tienes dos personas leyendo y escribiendo en un libro al mismo tiempo. Si una persona cambia una página mientras la otra está leyendo, la información que leen puede no ser correcta o estar desactualizada.*

*En programación, la inconsistencia de memoria ocurre cuando diferentes hilos o procesos tienen una visión diferente del estado de la memoria compartida. Esto puede suceder porque los cambios realizados por un hilo no son visibles inmediatamente para otros hilos debido a la forma en que la memoria se gestiona y se actualiza.*

*Por ejemplo, si un hilo actualiza una variable y otro hilo lee esa variable antes de que la actualización se haya propagado, el segundo hilo podría obtener un valor incorrecto o desactualizado. Esto puede llevar a comportamientos inesperados y errores en el programa.*

- **Inanición:** es uno de los problemas más graves. Consiste en que se deniegue siempre el acceso a un recurso compartido al mismo hilo, de forma que quede bloqueado a la espera del mismo.
- **Interbloqueo o abrazo mortal:** es el otro de los problemas más graves. Es aquel en que un hilo está esperando por un recurso compartido que está asociado a un hilo cuyo estado es bloqueado.

*Imagina dos personas que quieren cruzar un puente estrecho desde lados opuestos. Si ambas intentan cruzar al mismo tiempo y ninguna cede el paso, ambas quedan atrapadas en el puente sin poder avanzar ni retroceder. En programación, esto ocurre cuando los hilos o procesos están esperando recursos que están siendo retenidos por otros hilos o procesos, creando un ciclo de dependencia que no se puede resolver.*

*Para poder evitarlos en gran medida, es necesario implementar mecanismos de sincronización entre los hilos de un proceso. De esta forma, se consigue que un hilo que quiere acceder al mismo recurso que otro, se quede en espera hasta*

*que se liberan dichos recursos. Este tipo de mecanismos se denominan zona de exclusión mutua.*

## 4.2 MECANISMOS DE COMUNICACIÓN

Estos hilos deben saber quiénes de ellos deben esperar y cuáles continuar con su ejecución. Para ello existen diferentes **MECANISMOS DE COMUNICACIÓN ENTRE HILOS:**

- **Operaciones atómicas:** son aquellas que se realizan a la vez, es decir, que forman un pack. De esta forma se evita que los datos compartidos tengan distintos valores para el resto de hilos del proceso. *Puede utilizarse para solventar problemas de inconsistencia de memoria.*
- **Secciones críticas:** se estructura el código de la aplicación de tal forma que se accede de forma ordenada a aquellos datos compartidos.
- **Semáforos:** este mecanismo solo puede tomar valores 0 o 1. El hilo que accede al recurso inicializa el semáforo a 1 y tras su finalización el valor se queda a 0.
- **Tuberías:** todos los hilos se añaden a una cola que se prioriza por medio de un algoritmo FIFO, es decir, el primero en solicitar el acceso será asignado al recurso.
- **Monitores:** garantizan que solo un hilo accederá al recurso con el estado de ejecución. Esto se consigue por medio del envío de señales. El proceso que accede recibe el uso del “candado” y cuando finaliza devuelve este al monitor.
- **Paso de mensajes:** todos los hilos deben tener implementados los métodos para entender los mensajes. Esto supone un mayor coste, aunque si existe seguridad en el envío y recepción de un mensaje, se garantiza que solo un proceso accederá en el mismo momento a un recurso.

## 5. Actividades Guiadas

### 5.1 INTERRUPCIONES

Una interrupción es una suspensión temporal de la ejecución de un proceso o de un hilo de ejecución. Las interrupciones no suelen pertenecer a los programas, sino al sistema operativo, viniendo generadas por peticiones realizadas por los dispositivos periféricos.

En Java, una interrupción es una indicación a un hilo de que debe detener su ejecución para hacer otra cosa. Es responsabilidad del programador decidir qué quiere hacer ante una interrupción, siendo lo más habitual detener la ejecución del hilo.

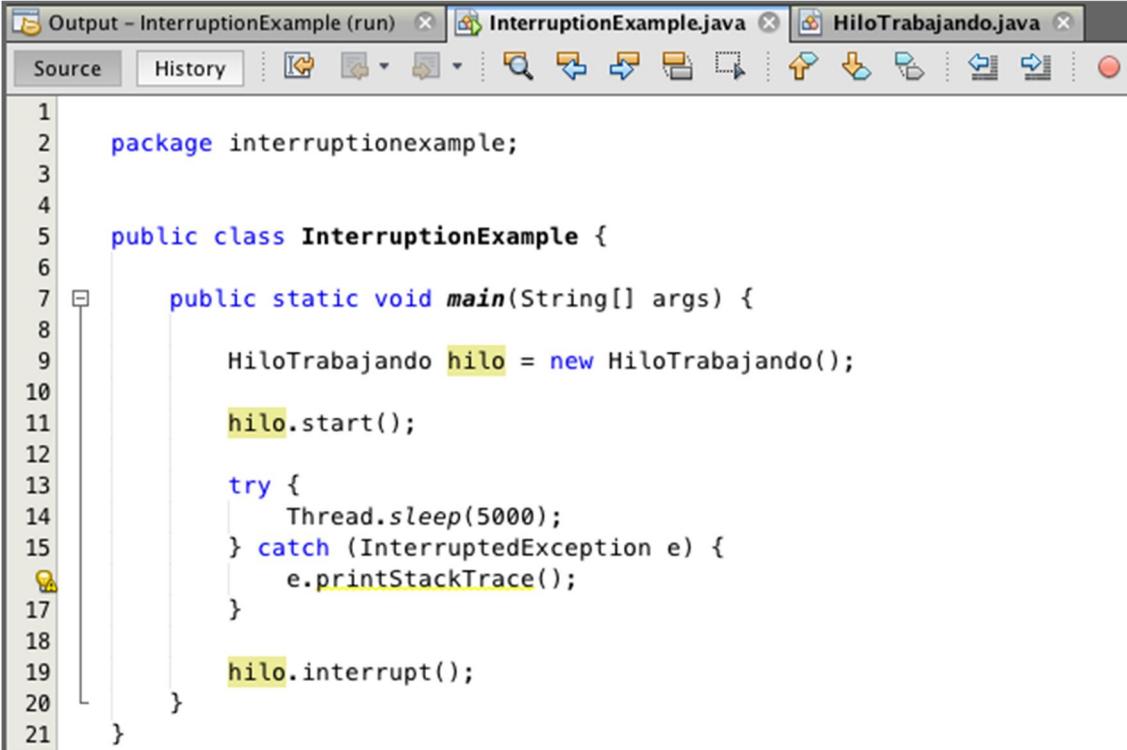
Las interrupciones se capturan mediante la excepción `InterruptedException`, provocadas por algunas operaciones como la invocación al método `sleep`. En caso de que la excepción no se tenga que controlar, se deberá invocar al método estático `interrupted` para gestionar la interrupción.

A continuación, vemos un ejemplo donde:

1. Creamos un hilo que ejecuta un bucle mientras no esté interrumpido.
2. Dentro del bucle, el hilo imprime un mensaje y duerme por 1 segundo para simular trabajo.
3. En el método main, dejamos que el hilo corra por 5 segundos.
4. Después de 5 segundos, interrumpimos el hilo usando `thread.interrupt()`.

Cuando el hilo es interrumpido, lanza una `InterruptedException`, y el mensaje "El hilo fue interrumpido." se imprime.

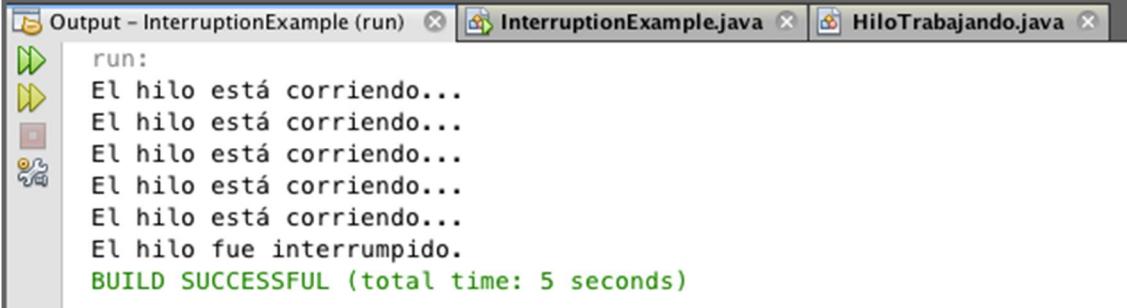
```
1 package interruptionexample;
2
3 public class HiloTrabajando extends Thread{
4
5     public void run() {
6         try {
7             while (true) {
8                 System.out.println("El hilo está corriendo...");
9                 Thread.sleep(1000); // Simula trabajo
10            }
11        }catch (InterruptedException e) {
12            System.out.println("El hilo fue interrumpido.");
13        }
14    }
15
16 }
```



The screenshot shows an IDE interface with three tabs at the top: "Output - InterruptionExample (run)", "InterruptionExample.java", and "HiloTrabajando.java". The "Source" tab is selected, displaying the following Java code:

```
1 package interruptionexample;
2
3
4
5 public class InterruptionExample {
6
7     public static void main(String[] args) {
8
9         HiloTrabajando hilo = new HiloTrabajando();
10
11         hilo.start();
12
13         try {
14             Thread.sleep(5000);
15         } catch (InterruptedException e) {
16             e.printStackTrace();
17         }
18
19         hilo.interrupt();
20     }
21 }
```

La salida es la siguiente:



The "Output" window shows the following text:

```
run:
El hilo está corriendo...
El hilo fue interrumpido.
BUILD SUCCESSFUL (total time: 5 seconds)
```

## 5.2 COMPARTICIÓN DE INFORMACIÓN

Si se desea que varios hilos comparten información existen varias alternativas:

- Utilizar **ATRIBUTOS ESTÁTICOS**. Los atributos estáticos son comunes a todas las instancias, por lo que independientemente de la manera de construir los hilos la información es compartida.
- Utilizando **REFERENCIAS A OBJETOS COMUNES** accesibles desde todos los hilos.

La sección crítica de un programa multihilo es el bloque de código que accede a recursos compartidos, por lo que solo debe ser accedido por un único hilo de ejecución. Determinar correctamente la sección crítica permite sincronizar correctamente el programa para evitar errores de concurrencia, así como hacerlo eficiente para poder aprovechar al máximo el paralelismo. Garantizar que a la sección crítica sólo acceda un hilo de ejecución es lo que se conoce como **EXCLUSIÓN MÚTUA**.

**ACTIVIDAD 1. DETERMINA LA SECCIÓN CRÍTICA DEL SIGUIENTE CÓDIGO.**

```
public class DeterminaSeccionCritica extends Thread{

    private int contador;
    private static int acumulador;
    private void ajustaContador(){this.contador-=500;}
    private void ajustaAcumulador(){this.acumulador-=500;}


    public void run(){
        for(int j=0; j<1000; j++)
        {
            contador++;
            acumulador++;
            if(contador==600) ajustaContador();
            if(acumulador==600) ajustaAcumulador();
        }
    }

    public static void main(String[] args) {
        for(int i=0; i<10; i++)
            new DeterminaSeccionCritica().start();
    }
}
```

La sección crítica es la parte donde se accede y modifica el recurso compartido acumulador (líneas `acumulador++` y `ajustaAcumulador()`) ya que es una variable estática y, como tal, está compartida entre todas las instancias de la clase `DeterminaSeccionCritica`. Esta situación puede ocasionar **condiciones de carrera**.

## 5.3 SINCRONIZACIÓN BÁSICA: WAIT, NOTIFY, NOTIFYALL

Los métodos `wait`, `notify` y `notifyAll` son propios de la clase `Object`, por lo que todas las clases en Java disponen de ellos.

Todos estos métodos se tienen que invocar desde segmentos de código de un hilo que disponga de un monitor, como por ejemplo, un bloque o segmento sincronizados, y obliga a capturar una excepción de tipo `InterruptedException`.

El método **wait detiene la ejecución del hilo y los métodos notify y notifyAll producen la reactivación de los hilos detenidos**. El método `notify` hace continuar a un único segmento al azar de los que están detenidos con `wait` mientras que `notifyAll` hace continuar a todos los segmentos detenidos con `wait`.

### ¿Qué quiere decir “disponer de un monitor”?

Imagina que el monitor de un objeto es una llave para una habitación. Solo una persona (hilo) puede tener la llave y entrar a la habitación a la vez. Mientras esa persona está dentro, nadie más puede entrar porque la puerta está cerrada con llave.

- **Poseer el monitor:** Significa que un hilo tiene la llave y está dentro de la habitación (la habitación representa el bloque o método sincronizado). Mientras esté dentro, otros hilos deben esperar fuera.
- **Liberar el monitor:** Ocurre cuando el hilo sale de la habitación y devuelve la llave, permitiendo que otro hilo pueda entrar.

*En términos de programación:*

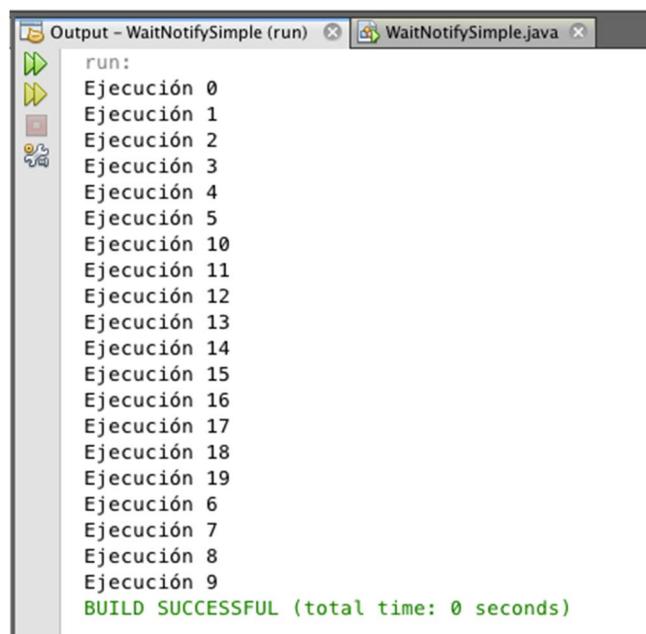
- Un hilo adquiere el monitor cuando entra en un bloque o método sincronizado,
- Mientras el hilo tiene el monitor, otros hilos **no** pueden ejecutar bloques o métodos sincronizados en el mismo objeto.
- Cuando el hilo sale del bloque o método sincronizado, libera el monitor, permitiendo que otros hilos puedan adquirirlo y entrar.

**Al limitar el acceso a un segmento de código mediante el uso de synchronized, se logra la exclusión mútua o mutex, permitiendo delimitar una sección crítica con precisión.**

En Java, la palabra clave `volatile` asegura que **cualquier cambio realizado en la variable es visible inmediatamente para todos los hilos**. Sin `volatile`, los hilos pueden almacenar en caché el valor de la variable, lo que puede llevar a inconsistencias. Las variables `volatile` son útiles en situaciones donde múltiples hilos necesitan leer y escribir en una variable compartida sin necesidad de sincronización completa. Sin embargo, `volatile` **no es un sustituto para la sincronización cuando se necesita garantizar la atomicidad de operaciones complejas**.

**DESARROLLO DE APLICACIONES MULTIPЛАTAFORMA**  
**PROGRAMACI覩N DE SERVICIOS Y PROCESOS**

```
1 package waitnotifysimple;
2
3 public class WaitNotifySimple implements Runnable{
4
5     private volatile boolean ejecutandoMetodo1=false;
6
7     public synchronized void metodo1() {
8         for(int i=0; i<10; i++){
9             System.out.println("Ejecuci覩n "+i);
10            if(i==5){
11                try {this.wait();}
12                catch (InterruptedException e){e.printStackTrace();}
13            }
14        }
15    }
16
17    public synchronized void metodo2() {
18        for (int i=10; i<20; i++)
19            System.out.println("Ejecuci覩n "+i);
20        this.notifyAll();
21    }
22
23    public void run() {
24        if(ejecutandoMetodo1==false) {
25            ejecutandoMetodo1=true;
26            metodo1();
27        }
28        else { metodo2(); }
29    }
30
31    public static void main(String[] args) {
32        WaitNotifySimple objetoComun = new WaitNotifySimple();
33        new Thread(objetoComun).start();
34        new Thread(objetoComun).start();
35    }
36}
37
```



The screenshot shows the Eclipse IDE's Output window titled "Output - WaitNotifySimple (run)". It displays the console output of the application, which consists of a sequence of "Ejecuci覩n" (Execution) messages from 0 to 19, followed by the message "BUILD SUCCESSFUL (total time: 0 seconds)".

```
run:
Ejecuci覩n 0
Ejecuci覩n 1
Ejecuci覩n 2
Ejecuci覩n 3
Ejecuci覩n 4
Ejecuci覩n 5
Ejecuci覩n 6
Ejecuci覩n 7
Ejecuci覩n 8
Ejecuci覩n 9
Ejecuci覩n 10
Ejecuci覩n 11
Ejecuci覩n 12
Ejecuci覩n 13
Ejecuci覩n 14
Ejecuci覩n 15
Ejecuci覩n 16
Ejecuci覩n 17
Ejecuci覩n 18
Ejecuci覩n 19
BUILD SUCCESSFUL (total time: 0 seconds)
```

## PRODUCTOR-CONSUMIDOR

El problema productor-consumidor es un clásico problema de sincronización en la programación de hilos. Se trata de coordinar dos tipos de hilos: productores y consumidores, que comparten un recurso común, como un buffer o una cola.

### Descripción del Problema:

- **Productor:** Genera datos y los coloca en el buffer.
- **Consumidor:** Toma datos del buffer y los procesa.

### Desafíos:

1. **Sincronización:** Asegurar que los productores no añadan datos al buffer cuando esté lleno y que los consumidores no intenten tomar datos cuando el buffer esté vacío.
2. **Evitar condiciones de carrera:** Prevenir que múltiples hilos accedan al buffer simultáneamente, lo que podría causar inconsistencias en los datos.
3. **Eficiencia:** Minimizar el tiempo que los hilos pasan esperando para acceder al buffer.

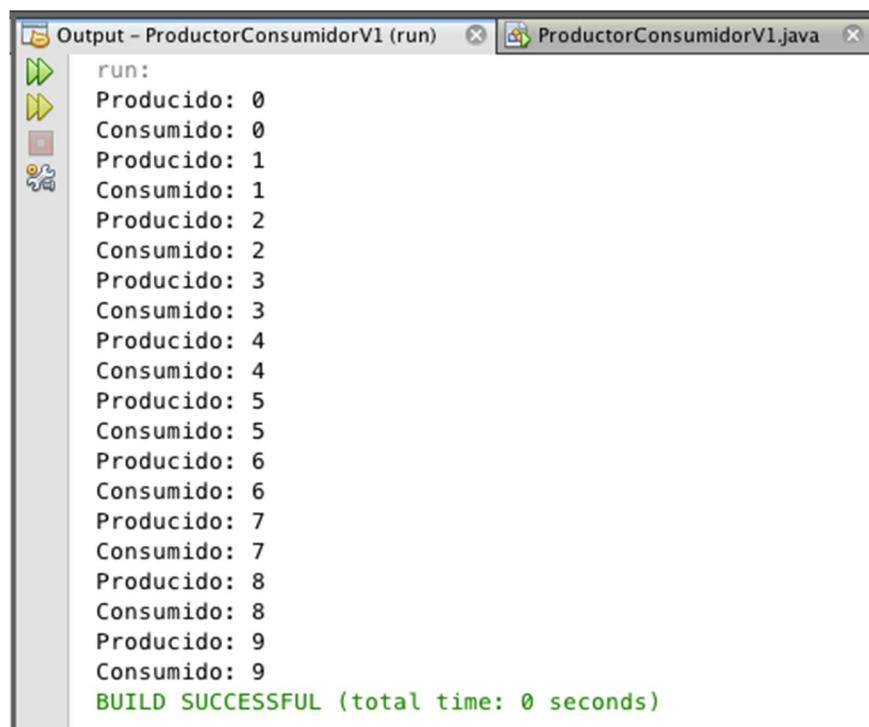
```
1 package productorconsumidorv1;
2
3 public class ProductorConsumidorV1 {
4
5     public static void main(String[] args) {
6         Buffer buffer = new Buffer();
7
8         Productor pr = new Productor(buffer);
9         Consumidor cs = new Consumidor(buffer);
10
11         pr.start();
12         cs.start();
13     }
14 }
```

```
1 package productorconsumidorv1;
2
3 public class Consumidor extends Thread {
4
5     private Buffer buffer;
6
7     public Consumidor(Buffer buffer){
8         this.buffer=buffer;
9     }
10
11     public void run() {
12         for (int i = 0; i < 10; i++)
13             buffer.consume();
14     }
15
16 }
```

```
1 package productorconsumidorv1;
2
3 public class Productor extends Thread {
4
5     private Buffer buffer;
6
7     public Productor(Buffer buffer){
8         this.buffer=buffer;
9     }
10
11     public void run() {
12         for (int i = 0; i < 10; i++)
13             buffer.produce(i);
14     }
15
16 }
17
```

**DESARROLLO DE APLICACIONES MULTIPLATAFORMA**  
**PROGRAMACIÓN DE SERVICIOS Y PROCESOS**

```
1 package productorconsumidorv1;
2
3 public class Buffer {
4
5     private int data;
6     private boolean empty = true;
7
8     public synchronized void produce(int value) {
9         while (!empty) {
10            try {
11                wait(); // Espera hasta que el buffer esté vacío
12            } catch (InterruptedException e) {
13                Thread.currentThread().interrupt();
14            }
15        }
16        data = value;
17        empty = false;
18        System.out.println("Producido: " + value);
19        notify(); // Notifica al consumidor que hay datos disponibles
20    }
21
22     public synchronized int consume() {
23         while (empty) {
24             try {
25                 wait(); // Espera hasta que haya datos disponibles
26             } catch (InterruptedException e) {
27                 Thread.currentThread().interrupt();
28             }
29         }
30         empty = true;
31         System.out.println("Consumido: " + data);
32         notify(); // Notifica al productor que el buffer está vacío
33         return data;
34     }
35 }
```



```
Output - ProductorConsumidorV1 (run) × ProductorConsumidorV1.java ×
run:
Producido: 0
Consumido: 0
Producido: 1
Consumido: 1
Producido: 2
Consumido: 2
Producido: 3
Consumido: 3
Producido: 4
Consumido: 4
Producido: 5
Consumido: 5
Producido: 6
Consumido: 6
Producido: 7
Consumido: 7
Producido: 8
Consumido: 8
Producido: 9
Consumido: 9
BUILD SUCCESSFUL (total time: 0 seconds)
```

**ACTIVIDAD 2. Resuelve el problema de sincronización del apartado 5.2**

**ACTIVIDAD 3.** Modifica el ejemplo Productor-Consumidor del apartado 5.3 para que la clase productor envíe las cadenas PING y PONG (de forma alternativa) y la clase consumidor recoja las cadenas de modo que la salida que se muestre al ejecutar el programa sea la siguiente: PING PONG PING PONG PING PONG...

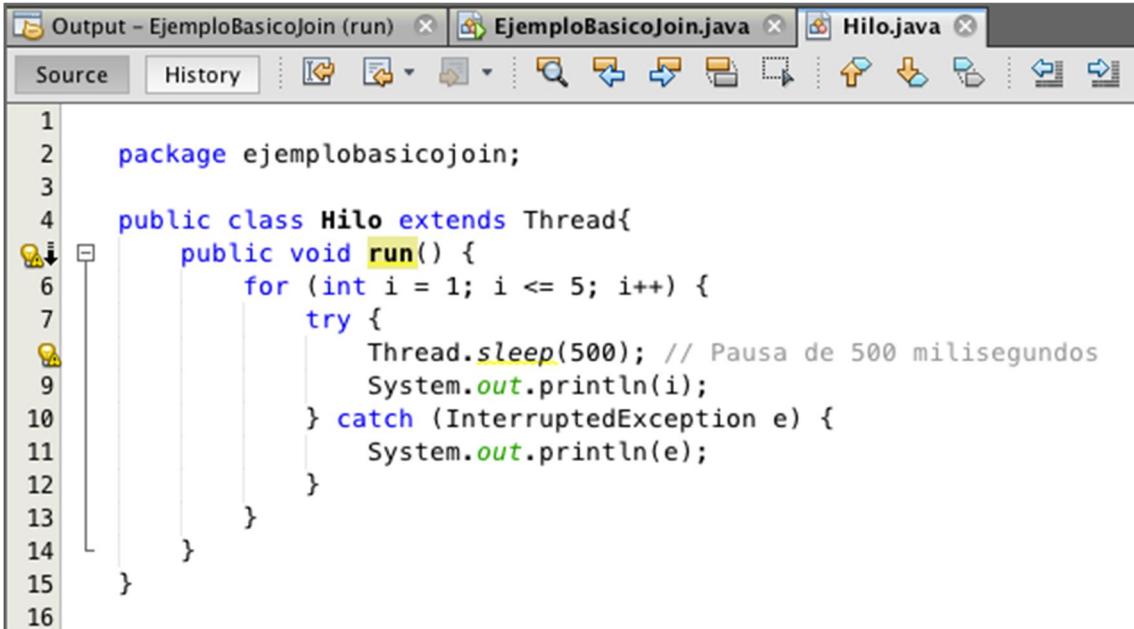
## 5.4 SINCRONIZACIÓN BÁSICA: MÉTODO JOIN

El método `join()` en Java y el método `wait()` en C tienen propósitos similares en cuanto a la sincronización. Ambos se utilizan para esperar que otra tarea termine con la salvedad que `join()` se aplica a hilos dentro de un mismo programa en Java, mientras que `wait()` en C se utiliza para procesos

**join() en Java:** se utiliza para hacer que un hilo espere a que otro hilo termine su ejecución antes de continuar. Por ejemplo, si el hilo principal llama a `join()` en un hilo secundario, el hilo principal se detendrá hasta que el hilo secundario haya terminado.

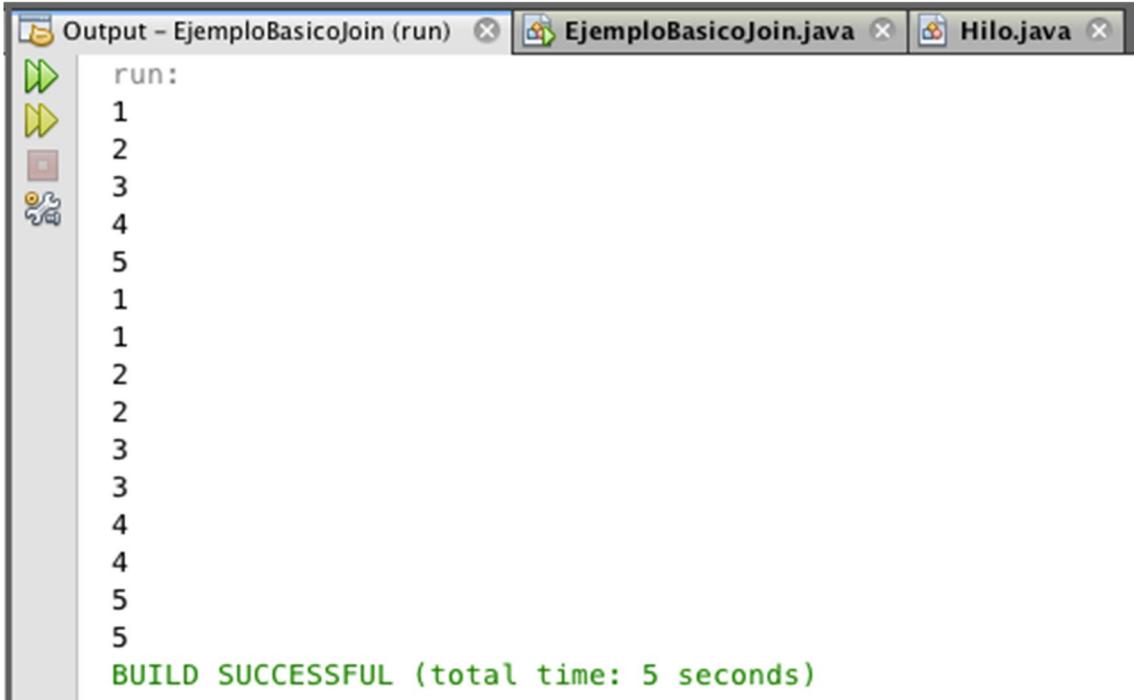
```
1 package ejemplobasicojoin;
2
3 public class EjemploBasicoJoin {
4
5     public static void main(String[] args) {
6         Hilo h1 = new Hilo();
7         Hilo h2 = new Hilo();
8         Hilo h3 = new Hilo();
9
10        h1.start();
11        try {
12            h1.join();
13        } catch (InterruptedException e) {
14            System.out.println(e);
15        }
16
17        h2.start();
18        h3.start();
19    }
20
21
22 }
```

**DESARROLLO DE APLICACIONES MULTIPLATAFORMA**  
**PROGRAMACIÓN DE SERVICIOS Y PROCESOS**



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

```
1 package ejemplobasicojoin;
2
3 public class Hilo extends Thread{
4     public void run() {
5         for (int i = 1; i <= 5; i++) {
6             try {
7                 Thread.sleep(500); // Pausa de 500 milisegundos
8                 System.out.println(i);
9             } catch (InterruptedException e) {
10                 System.out.println(e);
11             }
12         }
13     }
14 }
15
16 }
```



The screenshot shows the IDE's output window with the following text:

```
run:
1
2
3
4
5
1
1
2
2
3
3
4
4
5
5
5
```

BUILD SUCCESSFUL (total time: 5 seconds)

**ACTIVIDAD 4.** Utiliza join como método de sincronización para que un programa que arranca dos hilos indique al hilo2 que debe quedar suspendido hasta que termine la ejecución del hilo1. Ten en cuenta que ninguno de los dos hilos son el principal, con lo que es necesario que hilo2 tenga una referencia a hilo1 para invocar el método join.

## 5.5 SINCRONIZACIÓN AVANZADA: SEMÁFOROS

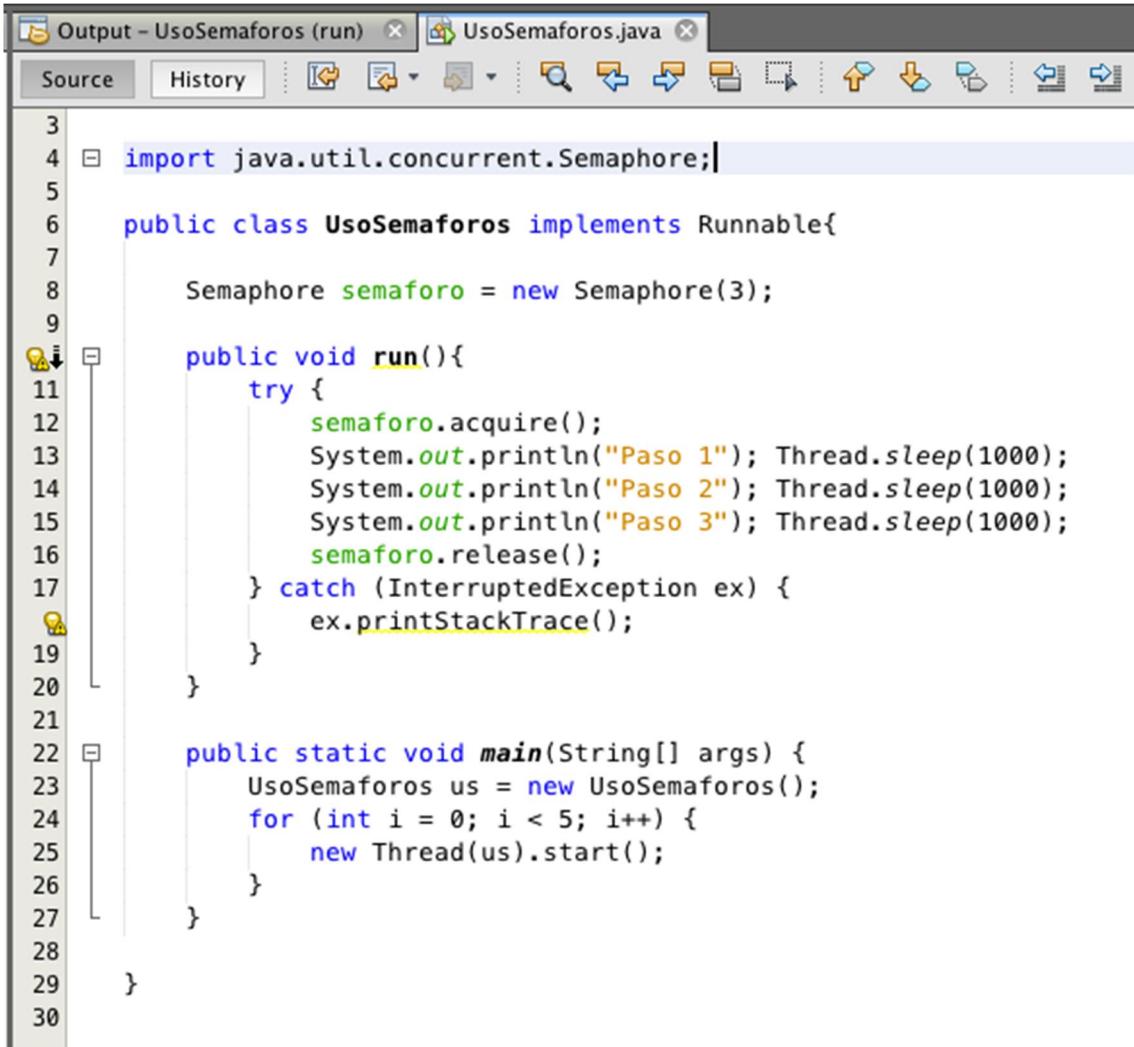
El uso de la palabra reservada `synchronized` permite establecer secciones críticas a las que únicamente tiene acceso un hilo en un momento determinado, pero existen otros escenarios en los que los recursos limitados permiten el acceso de más de un hilo.

Cuando una sección crítica permite ser ejecutada por más de un hilo, pero el número de estos está limitado, se utiliza un elemento de programación concurrente conocido como semáforo e implementado en Java por la clase `Semaphore`.

Al construir el semáforo se proporciona, a través del constructor, la referencia al número de hilos que pueden estar ejecutando concurrentemente la sección crítica. Los principales métodos de la clase son:

- **adquiere**: el hilo accede a la sección crítica si el semáforo lo permite. En caso contrario, se queda a la espera.
- **release**: libera su permiso de acceso a la sección crítica para que otro hilo que esté esperando pueda entrar.

En el siguiente ejemplo se construye un semáforo con capacidad para tres hilos. A su vez, 5 hilos construidos a partir de la misma instancia `Runnable`, intentan acceder a la sección crítica.



The screenshot shows an IDE interface with the tab 'UsoSemaforos.java' selected. The code editor displays the following Java code:

```
3
4 import java.util.concurrent.Semaphore;
5
6 public class UsoSemaforos implements Runnable{
7
8     Semaphore semaforo = new Semaphore(3);
9
10    public void run(){
11        try {
12            semaforo.acquire();
13            System.out.println("Paso 1"); Thread.sleep(1000);
14            System.out.println("Paso 2"); Thread.sleep(1000);
15            System.out.println("Paso 3"); Thread.sleep(1000);
16            semaforo.release();
17        } catch (InterruptedException ex) {
18            ex.printStackTrace();
19        }
20    }
21
22    public static void main(String[] args) {
23        UsoSemaforos us = new UsoSemaforos();
24        for (int i = 0; i < 5; i++) {
25            new Thread(us).start();
26        }
27    }
28
29 }
30
```

**DESARROLLO DE APLICACIONES MULTIPLATAFORMA**  
PROGRAMACIÓN DE SERVICIOS Y PROCESOS

La salida muestra que los tres primeros hilos que han intentado acceder a la sección crítica a través del bloqueo del semáforo se han ejecutado concurrentemente, y el resto de hilos se han ejecutado cuando los primeros han terminado y liberado los bloqueos.

```
Output - UsoSemaforos (run)  X  UsoSemaforos.java  X
run:
Paso 1
Paso 1
Paso 1
Paso 2
Paso 2
Paso 2
Paso 3
Paso 3
Paso 3
Paso 1
Paso 1
Paso 2
Paso 2
Paso 3
Paso 3
BUILD SUCCESSFUL (total time: 6 seconds)
```

## 6. Actividades Propuestas

### ACTIVIDAD 5. PASO POR PUENTE. RESUELTA

Analiza, comprende y explica el funcionamiento del código.

¿QUÉ DIFERENCIA HAY ENTRE APlicar SYNCHRONIZED A UN MÉTODO O HACERLO A UN BLOQUE DE CÓDIGO SOBRE UN OBJETO?

En el siguiente programa de ejemplo se simula un sistema que controla el paso de personas por un puente, siempre en la misma dirección y cumpliendo las siguientes restricciones:

- No pueden pasar más de 3 personas a la vez.



- No puede haber más de 200kg de peso en ningún momento.
- Cada persona se representará mediante un hilo.
- El tiempo de llegada de dos personas es aleatorio entre 1 y 10s.
- Cada persona tarda en atravesar el puente un tiempo aleatorio entre 10 y 25s.
- Las personas tienen un peso aleatorio entre 40 y 120kg.

```
Output - pasoPorPuente (run)  PasoPorPuente.java  Persona.java  Puente.java
run:
Comienza la Simulacion
La siguiente persona llega en 6 segundos.
La siguiente persona llega en 8 segundos.
1 que pesa 51 quiere cruzar. En el puente hay 0 personas y en total pesan 0Kg.
1 que pesa 51 tiene autorizacion. Con ella, en el puente hay 1 personas y en total pesan 51Kg.
1 va a tardar 16 segundos en cruzar.
La siguiente persona llega en 7 segundos.
2 que pesa 116 quiere cruzar. En el puente hay 1 personas y en total pesan 51Kg.
2 que pesa 116 tiene autorizacion. Con ella, en el puente hay 2 personas y en total pesan 167Kg.
2 va a tardar 23 segundos en cruzar.
La siguiente persona llega en 7 segundos.
3 que pesa 110 quiere cruzar. En el puente hay 2 personas y en total pesan 167Kg.
3 NO tiene autorizacion para pasar.
1 que pesa 51 quiere salir del puente. En el puente hay 2 personas y en total pesan 167Kg.
1 Ha salido del puente. Ahora en el puente hay 1 personas y en total pesan 116Kg.
3 NO tiene autorizacion para pasar.
La siguiente persona llega en 9 segundos.
4 que pesa 58 quiere cruzar. En el puente hay 1 personas y en total pesan 116Kg.
4 que pesa 58 tiene autorizacion. Con ella, en el puente hay 2 personas y en total pesan 174Kg.
4 va a tardar 14 segundos en cruzar.
2 que pesa 116 quiere salir del puente. En el puente hay 2 personas y en total pesan 174Kg.
2 Ha salido del puente. Ahora en el puente hay 1 personas y en total pesan 58Kg.
3 que pesa 110 tiene autorizacion. Con ella, en el puente hay 2 personas y en total pesan 168Kg.
3 va a tardar 25 segundos en cruzar.
La siguiente persona llega en 2 segundos.
5 que pesa 41 quiere cruzar. En el puente hay 2 personas y en total pesan 168Kg.
5 NO tiene autorizacion para pasar.
La siguiente persona llega en 2 segundos.
6 que pesa 102 quiere cruzar. En el puente hay 2 personas y en total pesan 168Kg.
6 NO tiene autorizacion para pasar.
```

## DESARROLLO DE APLICACIONES MULTIPLATAFORMA

### PROGRAMACIÓN DE SERVICIOS Y PROCESOS

```
package pasoporpuente;

import java.util.Random;

public class Persona implements Runnable
{
    private final int dni;
    private final int peso;
    private final int tMinPaso, tMaxPaso;
    private final Puente puente;

    public Persona(Puente puente, int peso, int tMinPaso, int tMaxPaso, int dni)
    {
        this.dni=dni;
        this.peso=peso;
        this.tMinPaso=tMinPaso;
        this.tMaxPaso=tMaxPaso;
        this.puente=puente;
    }

    public int getDNI(){return this.dni;}
    public int getPeso(){return this.peso;}
    public int getTMinPaso(){return this.tMinPaso;}
    public int getTMaxPaso(){return this.tMaxPaso;}
    public Puente getPuente(){return this.puente;}

    @Override
    public void run()
    {
        System.out.println(this.dni+" que pesa "+this.peso+" quiere cruzar. "+
            "En el puente hay "+puente.getNumPersonas()+" personas y "+
            "en total pesan "+puente.getPeso()+"Kg.");

        //PEDIR PASO
        Boolean autorizado=false;
        while(!autorizado)
        {
            synchronized(this.puente){
                autorizado=this.puente.autorizacionPaso(this);
                if(autorizado==false)
                {
                    System.out.println(this.dni+" NO tiene autorizacion para pasar.");
                    try {
                        this.puente.wait();
                    } catch (InterruptedException ex) {
                        System.out.println(this.dni+" estaba esperando y recibe INTERRUPCION.");
                    }
                }
            }
        }
    }
}
```

## DESARROLLO DE APLICACIONES MULTIPЛАTAFORMA

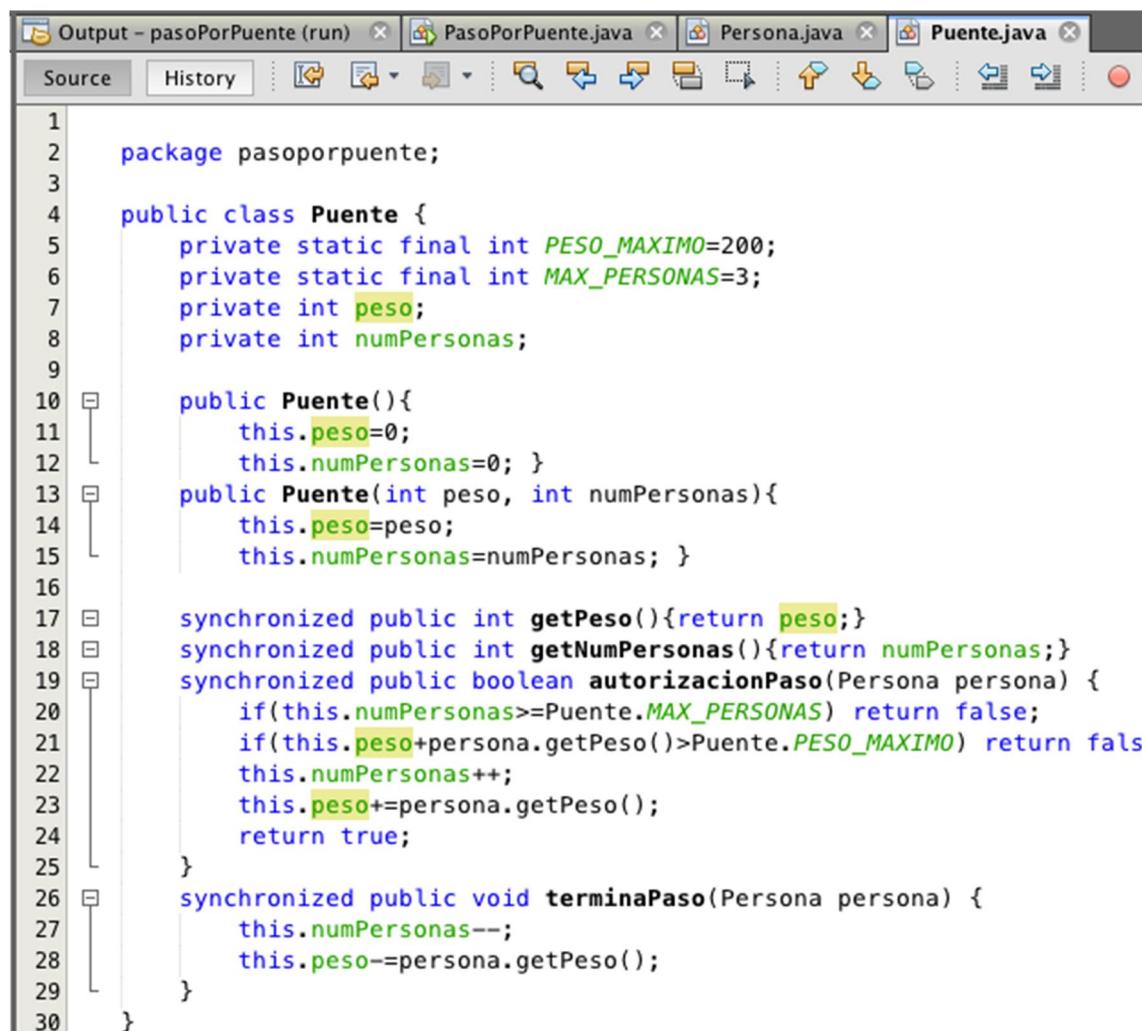
### PROGRAMACIОN DE SERVICIOS Y PROCESOS

```

//CRUZAR
System.out.println(this.dni+" que pesa "+this.peso+" tiene autorizacion. "+
    "Con ella, en el puente hay "+puente.getNumPersonas()+" personas y "+
    "en total pesan "+puente.getPeso()+"Kg.");
Random r = new Random();
int tEnCruzar = this.tMinPaso+r.nextInt(this.tMaxPaso-this.tMinPaso+1);
System.out.println(this.dni+" va a tardar "+tEnCruzar+" segundos en cruzar.");
try {
    Thread.sleep(tEnCruzar*1000);
} catch (InterruptedException ex) {
    System.out.println("Interrupcion mientras "+this.dni+" est谩 cruzando.");
}

//SALIR DEL PUENTE
synchronized(this.puente){
    System.out.println(this.dni+" que pesa "+this.peso+" quiere salir del puente. "+
        "En el puente hay "+puente.getNumPersonas()+" personas y "+
        "en total pesan "+puente.getPeso()+"Kg.");
    this.puente.terminaPaso(this);
    System.out.println(this.dni+" Ha salido del puente. "+
        "Ahora en el puente hay "+puente.getNumPersonas()+" personas y "+
        "en total pesan "+puente.getPeso()+"Kg.");
    puente.notifyAll();
}
}
}

```



```

1 package pasoporpuente;
2
3 public class Puente {
4     private static final int PESO_MAXIMO=200;
5     private static final int MAX_PERSONAS=3;
6     private int peso;
7     private int numPersonas;
8
9
10    public Puente(){
11        this.peso=0;
12        this.numPersonas=0; }
13    public Puente(int peso, int numPersonas){
14        this.peso=peso;
15        this.numPersonas=numPersonas; }
16
17    synchronized public int getPeso(){return peso;}
18    synchronized public int getNumPersonas(){return numPersonas;}
19    synchronized public boolean autorizacionPaso(Persona persona) {
20        if(this.numPersonas>=Puente.MAX_PERSONAS) return false;
21        if(this.peso+persona.getPeso()>Puente.PESO_MAXIMO) return false;
22        this.numPersonas++;
23        this.peso+=persona.getPeso();
24        return true;
25    }
26    synchronized public void terminaPaso(Persona persona) {
27        this.numPersonas--;
28        this.peso-=persona.getPeso();
29    }
30 }

```

## DESARROLLO DE APLICACIONES MULTIPLATAFORMA

### PROGRAMACIÓN DE SERVICIOS Y PROCESOS

```
package pasoporpuente;

import java.util.Random;

public class PasoPorPuente {

    public static void main(String[] args)
    {

        final Puente puente = new Puente();
        int tMinLlegada = 1;
        int tMaxLlegada = 10;
        int tMinPaso = 10;
        int tMaxPaso = 25;
        int minPesoPersona = 40;
        int maxPesoPersona = 120;
        int dniP = 1;
        int tLlegadaP;
        int pesoP;

        System.out.println("Comienza la Simulacion");
        Random r = new Random();

        while(true)
        {
            tLlegadaP = tMinLlegada + r.nextInt(tMaxLlegada-tMinLlegada+1);
            pesoP = minPesoPersona + r.nextInt(maxPesoPersona-minPesoPersona+1);
            System.out.println("La siguiente persona llega en "+tLlegadaP+" segundos.");
            try {
                Thread.sleep(tLlegadaP*1000);
                Persona p = new Persona(puente, pesoP, tMinPaso, tMaxPaso, dniP);
                Thread hilo = new Thread(p);
                hilo.start();
                dniP++;
            } catch (InterruptedException ex) {
                System.out.println("Interrupcion mientras esperaba siguiente peaton");
            }
        }
    }
}
```

### **ACTIVIDAD 6. COLECTA.**



Varios hilos (por ejemplo, cuatro) realizan una colecta. En un tiempo aleatorio entre 10 y 200ms, consiguen una cantidad entre 4 y 25. La colecta termina cuando se llega a una cantidad de 2000. Utilizar una clase Colecta que almacene y permita consultar la cantidad recogida hasta el momento. El hilo principal crea un objeto de tipo Colecta, crea los hilos y les pasa este objeto, que es compartido entre todos y, por tanto, sus métodos deben protegerse de manera apropiada.

### **ACTIVIDAD 7. DEPÓSITOS DE PINTURA.**

Se tienen 3 depósitos de pintura, cada uno de un color primario, a saber: cian, magenta y amarillo. Se tienen 3 personas que intentan preparar, cada una, pintura de un color secundario distinto. Estos son rojo (mezcla de amarillo y magenta), azul (mezcla de amarillo y cian) y verde (mezcla de cian y magenta). Para preparar un color secundario, una persona debe tener acceso en exclusiva a los dos colores primarios necesarios. Cada persona debe implementarse mediante un hilo que, en un bucle sin fin, haga lo siguiente: obtener acceso en exclusiva al depósito de los dos colores primarios que necesita, mantenerlo durante un tiempo aleatorio (por ejemplo entre 100 y 500ms) y esperar un tiempo aleatorio antes de volver a intentar preparar un nuevo color, que podría estar, por ejemplo entre 1 y 2 s. Hay que tener especial cuidado en realizar el bloqueo de los depósitos de pintura (como se ha dicho, uno por cada color primario), de manera que no se pueda nunca producir un interbloqueo.



### **ACTIVIDAD 8. COMIDA DE FILÓSOFOS.**

Implementa una solución al problema de la comida de los 5 filósofos de Dijkstra que evite el interbloqueo. Cinco filósofos pasan la vida pensando en una mesa redonda y,



solo dejan de pensar de vez en cuando para comer de un cuenco de espaguetis. Para ello, deben tomar los palillos que tienen a ambos lados, primero uno y luego el otro. Por supuesto, no pueden coger un palillo si lo está utilizando otro filósofo. Entonces, deben esperar a que termine y lo suelte. Una vez tomado un palillo, no lo sueltan hasta que no consiguen el otro palillo. Entonces, comen y, cuando han terminado de comer, dejan ambos palillos de nuevo en la mesa. Se recomienda numerar a los filósofos de 0 a 4. El filósofo  $i$  come con los palillos  $i$  (a su izquierda) e  $(i+1)$  (a su derecha).

Debe tenerse especial cuidado para que la solución implementada elimine toda posibilidad de interbloqueo. ¿cómo lo has hecho? ¿sería posible, con la solución que has implementado, que algún filósofo no consiguiera comer nunca, por improbable que fuera? Si es así, explica el escenario en que esto sucedería.

**ACTIVIDAD 9. PRODUCTOR-CONSUMIDOR.**

Escribe un programa para el problema clásico de los procesos productores y consumidores, pero utilizando un contenedor. Un proceso productor almacena datos en un contenedor y un proceso consumidor obtiene los datos del mismo contenedor. Los datos deben consumirse en el mismo orden en que se producen, según el modelo de cola FIFO. Los datos producidos deben ser números secuenciales empezando por 1 e incrementando de uno en uno. El contenedor debe ser un ArrayList.



**ACTIVIDAD 10. APARCAMIENTO.**

Se tiene un aparcamiento con 50 plazas, numeradas de 1 a 50, al que intentan acceder coches continuamente. La entrada de coches se puede simular con el programa principal que, cada cierto tiempo (aleatorio entre 500 y 3000ms) crea un hilo que simula un coche. Cada coche se crea con una matrícula distinta. Cada coche intenta conseguir acceso al aparcamiento, indicando su matrícula para ello, que queda registrada para la plaza de aparcamiento libre, si la hay, que se le asigna. Pasado un tiempo aleatorio (por ejemplo 10 a 20s), el coche abandona el aparcamiento.

Desarrolla un programa que simule el parking.

