

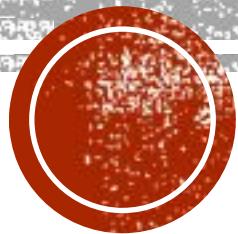
# **HILOS EN JAVA**

## **(TEMA 2)**

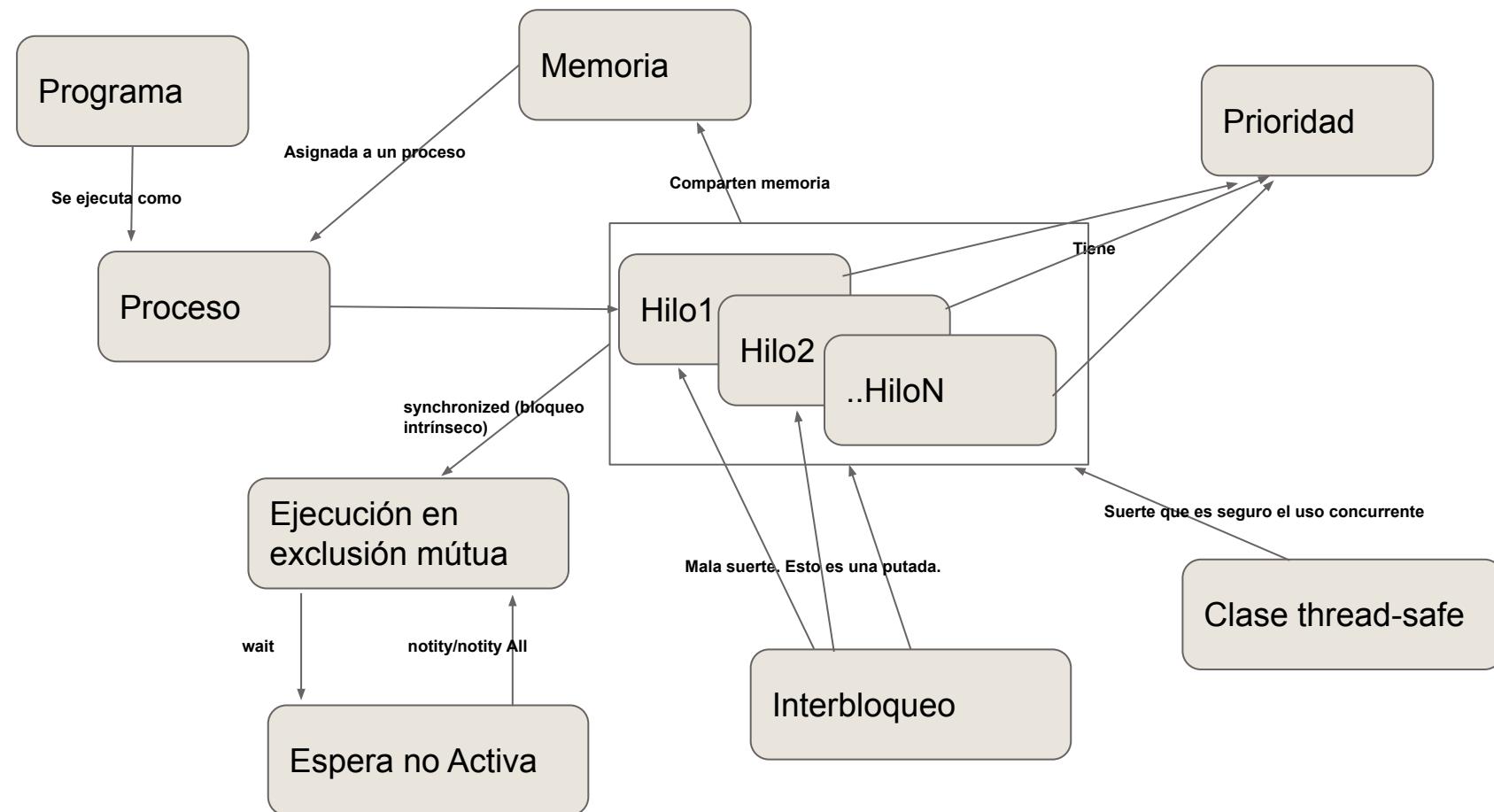
Santiago Rodenas Herráiz

IES VIRGEN DEL CARMEN (Departamento de Informática)

Curso 24/25

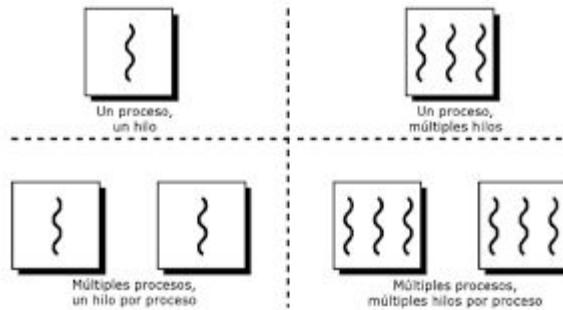


# Hilos (Mapa conceptual)



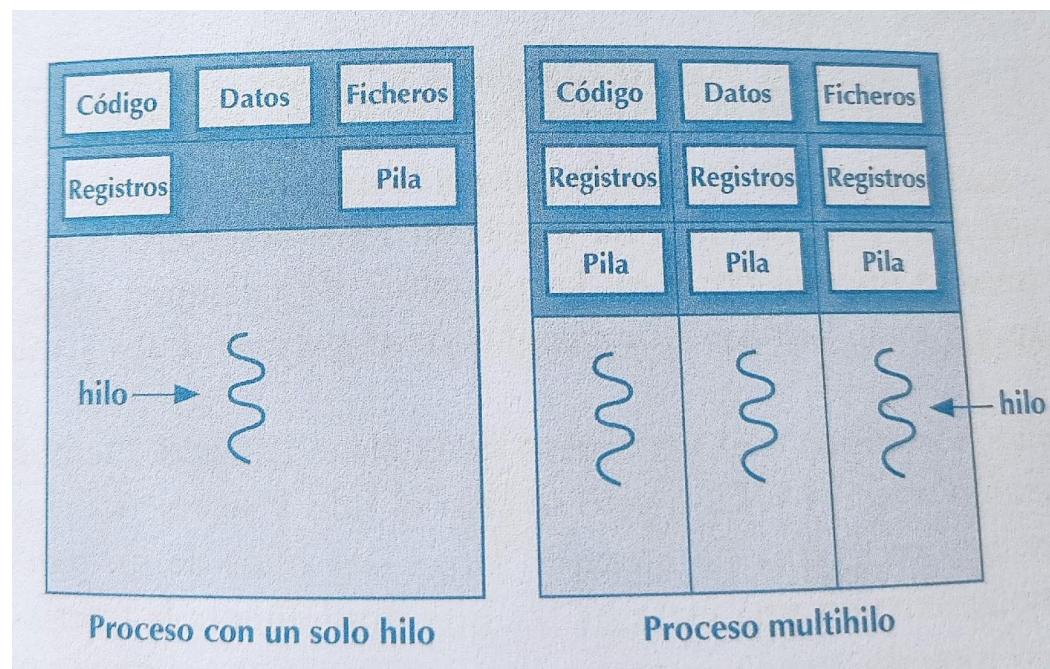
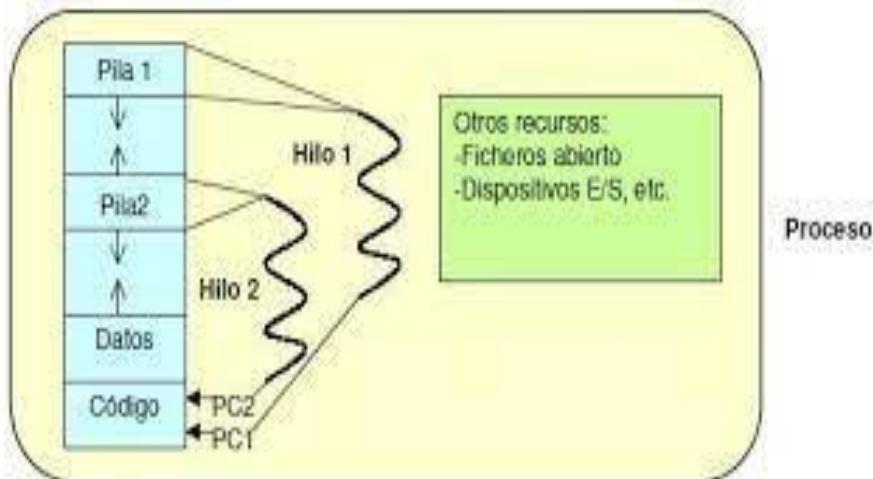
# Hilos

- Un **hilo** es una unidad de ejecución de un programa asociada a un proceso.
- Todos los hilos de un proceso, **comparten el mismo código y los datos**.
  - La información de acceso a ficheros.
  - Comparten aquellos datos que hay declarados en el proceso.
- Cada hilo tiene propio:
  - Los registros del procesador
  - El estado de su pila stack.
- Los cambios de contexto de un hilo por otro, es mucho más eficiente y rápido que con los procesos.



# Hilos

- Cada hilo posee su propia pila de ejecución, por tanto debe guardar el estado de los registros del procesador.
- La zona de código es totalmente compartida, por tanto es mucho más eficiente en ahorro de espacio en RAM
- Si un proceso abre un fichero, todos contienen el mismo descriptor de fichero.



# Hilos

- Ejemplos de hilos de ejecución los tenemos en:
  - **Servidores web** del tipo **Apache, Tomcat** que por cada petición **HTTP\_REQUEST**, éste crea un hilo de ejecución para atender la petición en forma de **HTTP\_RESPONSE**.
    - Desde el hilo principal asociado al proceso padre, se puede controlar perfectamente el número de peticiones.
  - **Juegos** en los que intervienen muchos elementos de interacción.
  - Programas **en tiempo real**. Existen multitud de sensores y cada hilo puede ocuparse de uno en particular. Por ejemplo, la alarma de una empresa en tanto:
    - Hilo por cada sensor de movimiento.
    - Hilo por cada cámara.
    - Hilo para la gestión y control.
    - Hilo para la conexión remota con centralita.
    - Hilo de actuación.



# Clase Thread

- Java incorpora una clase **thread-safe** (incorpora mecanismos de sincronización)
- Cuando se ejecuta una instancia de proceso en Java, viene asociado con un hilo de ejecución que se encarga de iniciar el **main**.
- Se pueden crear cualquier clase que implemente de la clase **abstracta Runnable**.
- La clase **Thread** acepta como parámetro un objeto que implemente de Runnable.
- Hay que sobreescribir el método **run**. `@override`
- Un hilo es lanzado con el método **start**.

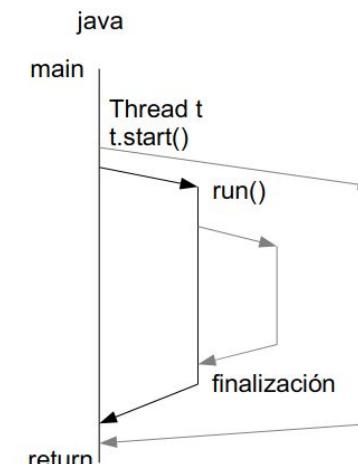


# Clase Thread

- Nuestras clases pueden heredar de **Thread** o implementar de **Runnable**.
- Clase que implementan de **Runnable**:
  - Hay que implementar el método **run()**.
  - Cuando creamos el hilo, debemos hacerlo utilizando la clase Thread cuyo constructor acepta un objeto de nuestra clase.

```
Thread hilo = new Thread (new ClaseQueImplementaRunnable(<arg>...));
```

- Clase que hereda de **Thread**:
  - Debemos de sobreescribir el método **run()**.



# Ejemplo con Runnable

```
package lanzahilos;

/*
 * La clase Hilo, debe implementar
 * de Runnable.
 */
class Hilo implements Runnable {

    private final String nombre;

    Hilo(String nombre) {
        this.nombre = nombre;
    }

    @Override
    public void run() {
        System.out.printf("Hola, soy el hilo: %s.\n", this.nombre);
        System.out.printf("Hilo %s terminado.\n", this.nombre);
    }
}

public class LanzaHilos {

    public static void main(String[] args) {

        Thread h1 = new Thread(new Hilo("H1 del profe Santi"));
        Thread h2 = new Thread(new Hilo("H2 del profe Gabriel"));
        h1.start();
        h2.start();
        System.out.println("Hilo principal terminado. Mis crios han terminado\n");
    }
}
```



# Actividades

- HACER EL EJERCICIO 1 DE LA PLATAFORMA
  - Clase Thread
  - Métodos:
    - isAlive()
    - currentThread()
    - getId()
    - getName()
    - getState()
- HACER EL EJERCICIO 2 DE LA PLATAFORMA



# Ejemplo con Thread

```
public class PingPong extends Thread
{
    private String word;
    public PingPong(String s) {word=s;}

    public void run()
    {
        for (int i=0;i<3000;i++)
        {System.out.print(word);
         System.out.flush();}
    }

    public static void main(String[] args)
    {Thread tP=new PingPong("P");
     Thread tp=new PingPong("p");
     //tP.setPriority(Thread.MAX_PRIORITY);
     //tp.setPriority(Thread.MIN_PRIORITY);
     tp.start();
     tP.start();
    }
}
```

¿QUÉ CREÉIS QUE SIGNIFICA **setPriority**?

- Probar a establecer prioridades diferentes a cada hilo y ejecutarlo.

# Clase Thread

- **Métodos:**

- **void run()**.- Se ejecuta cuando se lanza el hilo.
- **void start()**.- Lanza el método **run**. Hay que llamarlo desde el padre.
- **static void sleep(long ms)**.- Se duerme por un tiempo. (**bloqueo**)
- **void join()**.- Se espera a que acabe el hilo en cuestión.
- **void join(long ms, long ns)**.- Se espera a que acabe el hilo de ejecución por un máximo de tiempo.

Ejemplo  
HijoThread



# Lanza Hilos. Espera terminen

Pongamos un ejemplo de programa entre el hilo principal y otros **dos** que creemos nosotros mismos.

- Un programa formado por dos hilos. Cada uno de ellos hace pausas de duración aleatorias entre 10 y 500 ms, utilizando el método sleep de la clase Thread. El hilo principal utiliza el método join para esperar a que terminen los dos hilos lanzados, por lo que siempre terminará el último. Los dos métodos anteriores pausan la ejecución del hilo y durante ese periodo de tiempo se podría interrumpir. Si esto sucede, se lanzaría una **InterruptedException**, que se captura para mostrar un mensaje.

[Ejemplo LanzaHilosYEsperaQueTerminen](#)



# Lanza Hilos. Espera terminen

## Pasos:

1. Decidiremos que nuestra clase llamado **Hilo** implemente de **Runnable**, aunque podríamos crearnos una subclase directamente de **Thread**, pero por ahora lo haremos así:
2. Creamos un constructor, al que le pasaremos una cadena nombre para identificarlo.
3. Sobreescribimos nuestro método **run**.
  - a. El hilo debe hacer un número de pausas, para ello utilizamos el método estático de la clase **Thread**, al que le pasaremos una pausa aleatoria entre **10 y 500 ms**.
    - i. **pausa = 10 + new Random().nextInt(500 - 10);**
  - b. Claro está, esto lo hacemos un número de veces, por ejemplo **5**. Por tanto dentro de un bucle.
  - c. El hilo debe de dormirse con **Thread.sleep(pausa)** dentro de nuestro bloque try/catch. Hay que darse cuenta, que siempre que durmamos a un Hilo, éste puede no despertar y por tanto había que forzar una interrupción con **Control + C**.
4. Dentro del main, creamos ambos hilos con la clase Thread cuyo objeto que le pasamos es del tipo **Hilo**
5. Los lanzamos dentro de nuestro bloque try/catch y el hilo principal, debe esperar a la finalización de cada hilo



# Lanza Hilos. Espera terminen

- La solución del ejercicio, la tenéis en la plataforma.

Ejemplo LanzaHilosYEsperaQueTerminen



# Sincronización de hilos

- ¿CÓMO COMPARTEN DATOS LOS HILOS?

- Los hilos **comparten** los objetos y variables del proceso padre.
- Un hilo puede **modificar** el estado de un objeto del proceso y los demás lo ven.
- El problema está cuando un hilo que están modificando un objeto, **es interrumpido** por otro. Trataremos el problema de la **exclusión mutua**.
- Necesitamos **mecanismos de sincronización** que eviten un mal funcionamiento de nuestros programas.



# Cooperación entre hilos

- Ejemplo de acceso a un objeto común mediante la clase Contador.

```
package hilosincrementanvar;

/*
 * Clase Contador
 */
class Contador{

    private int conta = 0;

    public void incrementaContador(){this.conta++;}

    public int getContador(){ return this.conta; }

}
```



# Cooperación entre hilos

```
/*
 * Clase Hilo
 */
class Hilo implements Runnable{
    /*
     * Número de hilo, número de incrementos que debe llevar e
     * incrementos realizados
    */
    int numeroHilo;
    int cuantoIncremento;
    int numeroDeIncrementos=0;

    /*Es el contador compartido por cada hilo*/
    private Contador contador;

    /* Constructor */
    Hilo(int nHilo, int cIncremento, Contador conta){
        this.numeroHilo = nHilo;
        this.cuantoIncremento = cIncremento;
        this.contador = conta;
    }
    /* Devuelve el número de incrementos que ha hecho */
    public int getNumeroDeIncrementos(){
        return this.numeroDeIncrementos;
    }

    public void run(){
        for (int i=0; i<this.cuantoIncremento; i++){
            contador.incrementaContador();
            this.numeroDeIncrementos++;
        }
        System.out.printf("Hilo con id %s, ha hecho %d incrementos\n", this.numeroHilo, this.getNumeroDeIncrementos());
    }
}
```



# Cooperación entre hilos

```
public class HilosIncrementanVariable{
    private static final int    NUM_HILOS=10;
    private static final int    CUENTA_MAXIMA=100000;

    public static void main(String argv[]){
        Contador c = new Contador();
        Thread hilos [] = new Thread[NUM_HILOS];
        for (int i=0; i<NUM_HILOS; i++){
            hilos[i] = new Thread(new Hilo(i, CUENTA_MAXIMA/NUM_HILOS, c));
            hilos[i].start();
        }

        for (int i=0; i<NUM_HILOS; i++){
            try{
                hilos[i].join();
            }catch(InterruptedException e){
                System.out.println ("Se ha producido una interrupción no deseada\n");
            }
        }

        System.out.printf("Desde el hilo principal, se han producido un total de %d\n", c.getContador());
    }
}
```

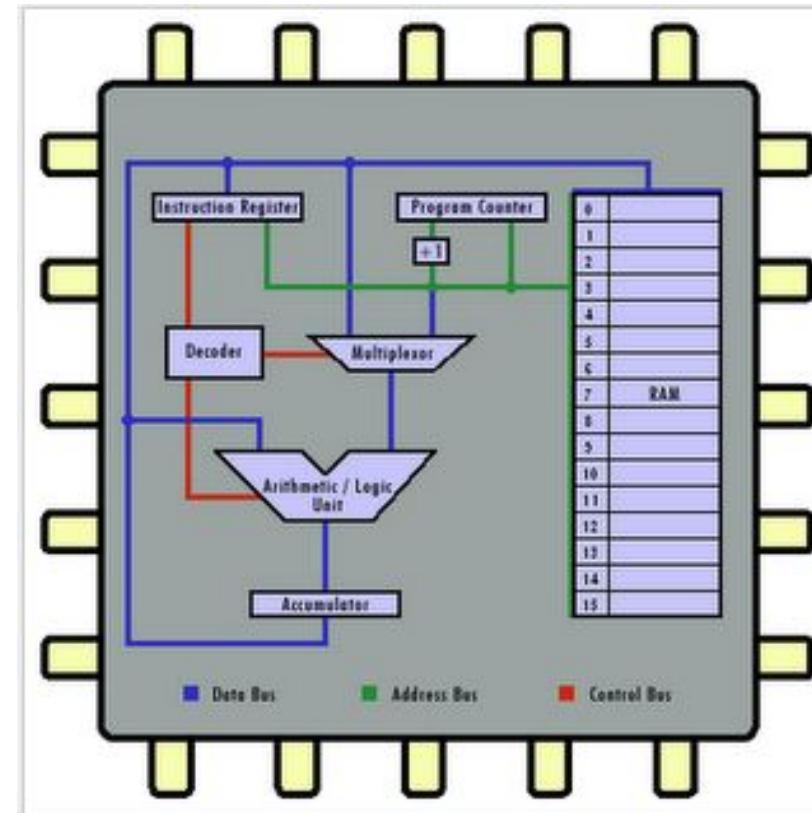
```
usuario@pc9-43:~/Descargas/PSP21_22-main/ejem_tema2/EjemploThreadHilosCooperantes/src$ java hilosincrementanvar.HilosIncrementanVariable
hilo con id 1, ha hecho 10000 incrementos
hilo con id 2, ha hecho 10000 incrementos
hilo con id 9, ha hecho 10000 incrementos
hilo con id 8, ha hecho 10000 incrementos
hilo con id 4, ha hecho 10000 incrementos
hilo con id 0, ha hecho 10000 incrementos
hilo con id 6, ha hecho 10000 incrementos
hilo con id 7, ha hecho 10000 incrementos
hilo con id 3, ha hecho 10000 incrementos
hilo con id 5, ha hecho 10000 incrementos
Desde el hilo principal, se han producido un total de 71117
```

# Cooperación entre hilos

- Para entender el comportamiento de este ejemplo, habría que repasar conceptos de arquitectura de una cpu y para ello pongo una muy sencilla que es la del 8085.
- Vamos allá.

Aconsejo ver presentación:

**Repaso general de computación**



# Cooperación entre hilos

- ¿Qué pasaría si el programa anterior no se llamara al método join?
- ¿Y si se hiciera en el mismo bucle que lanza los hilos con start?
- Vemos que existe un resultado erróneo. **¿Por qué?**
  - ¿Qué sucede si en el momento en el que el hilo1 debe incrementar su valor de contador en su registro del procesador, existe un cambio de contexto por otro hilo?
    - **Hilo1:** registro  $\leftarrow$  [cuenta]. Por ejemplo se almacena el valor 80
      - **<Cambio de contexto por Hilo2>**
    - **Hilo2:** registro  $\leftarrow$  [cuenta]. Almacena el mismo valor 80.
      - **<Cambio de contexto por Hilo1>**
    - **Hilo1:** registro  $\leftarrow$  registro + 1. Tendrá un valor de 81.
      - **<Cambio de contexto por Hilo2>**
    - **Hilo2:** registro  $\leftarrow$  registro + 1. Tendrá un valor de 81.
    - cuenta  $\leftarrow$  registro. El objeto cuenta tendrá un valor de 81 por Hilo2.
      - **<Cambio de contexto por Hilo1>**
    - cuenta  $\leftarrow$  registro. El objeto cuenta tendrá un valor de 81 por Hilo1.
  - **Conclusión:**
    - Con estos dos hilos, se ha perdido un incremento.



# Métodos de sincronización

- Para evitar este problema, tenemos los métodos sincronizados, **synchronized**.
- La idea es hacer de un método, una **operación atómica**.
- Cuando un hilo ejecuta un método sincronizado, el hilo que intente acceder a dicho recurso, se bloqueará hasta que éste no sea liberado.
- Es un método eficiente que incorpora la propia clase Thread para poder sincronizarse entre diferentes hilos de ejecución.

```
package hilosincrementar;  
  
class ContadorSincronizado{  
  
    private int conta = 0;  
  
    //serán métodos atómicos  
    synchronized public void incrementaContador(){  
        this.conta++;  
    }  
  
    synchronized public int getContador(){  
        return this.conta;  
    }  
}
```



# Métodos de sincronización

- Métodos de bloqueo a **nivel de objeto**.
  - El hilo que invoque un método sincronizado sobre un objeto, adquiere el bloqueo de dicho recurso y cuando otro hilo quiera invocar otro método sincronizado, sólo lo podrá hacer si son objetos diferentes.
- Métodos de bloqueo a **nivel de clase (métodos estáticos)**.
  - El hilo adquiere el bloqueo intrínseco para el objeto Class asociado con la clase.
  - Cuando un hilo adquiera bloqueo en cualquier objeto o instancia de la clase (recurso), cualquier hilo independiente de la la instancia de clase, no podrá ejecutar ningún método sincronizado.



# Bloqueo a nivel del objeto

## ■ Ejemplo de bloqueo de objeto.

- Crearemos un recurso donde sincronizaremos sólo un trozo de código.
- Cada hilo comparte el mismo recurso y al ejecutarse, accede directamente a la zona de exclusión mútua.
- Crearemos 4 hilos donde 3 de ellos comparten la misma instancia del recurso y otro hilo solicita él sólo una segunda instancia del recurso.
- Veremos cómo el hilo que adquiere el bloqueo del recurso que sólo comparte él, no tiene problema alguno.
- Veremos como los hilos que adquieren el bloqueo del recurso compartido, lo hacen por orden, por tanto existirá un bloqueo.



# Bloqueo a nivel del objeto

```
package metodosinterbloqueos;

class Recurso{
    public void bloquear(){
        System.out.println ("Soy el hilo cuyo nombre es "+ Thread.currentThread().getName() + "\n");
        synchronized(this){
            System.out.println (Thread.currentThread().getName() + " Accediendo al recurso y bloqueado \n");
            for (int i=0; i<100000000;i++);
            System.out.println (Thread.currentThread().getName() + " Deja y desbloquea el recurso \n");
        }
    }
}

class Hilo extends Thread{
    private Recurso objeto;
    Hilo (Recurso obj){
        this.objeto = obj;
    }
    public void run(){
        this.objeto.bloquear();
    }
}
```



# Bloqueo a nivel del objeto

```
public class MetodosDeInterbloqueos{  
    public static void main(String argv[]){  
  
        Recurso r1, r2, r3, r4;  
        Hilo hilo1, hilo2, hilo3, hilo4;  
  
        r1 = new Recurso();  
        r2 = new Recurso();  
  
        hilo1 = new Hilo(r1);  
        hilo2 = new Hilo(r1);  
        hilo3 = new Hilo(r2);  
        hilo4 = new Hilo(r1);  
  
        hilo1.setName("Hilo 1");  
        hilo2.setName("Hilo 2");  
        hilo3.setName("Hilo 3");  
        hilo4.setName("Hilo 4");  
  
        hilo1.start();  
        hilo2.start();  
        hilo3.start();  
        hilo4.start();  
  
    }  
}
```

```
usuario@pc9-43:~/Descargas/PSP21_22-main/ejem_tema2/EjemploMetodosInterbloqueos/src$ java metodosinterbloqueos.MetodosDeInterbloqueos  
Soy el hilo cuyo nombre es Hilo 1  
Soy el hilo cuyo nombre es Hilo 4  
Soy el hilo cuyo nombre es Hilo 3  
Soy el hilo cuyo nombre es Hilo 2  
Hilo 1 Accediendo al recurso y bloqueado  
Hilo 3 Accediendo al recurso y bloqueado  
Hilo 3 Deja y desbloquea el recurso  
Hilo 1 Deja y desbloquea el recurso  
Hilo 2 Accediendo al recurso y bloqueado  
Hilo 2 Deja y desbloquea el recurso  
Hilo 4 Accediendo al recurso y bloqueado  
Hilo 4 Deja y desbloquea el recurso
```



# Bloqueo a nivel de clase

## ■ Ejemplo de bloqueo de clase.

- Podemos sincronizar el código de datos con el nombre de la clase.class o incluso si el código hace referencia a métodos estáticos, donde no requiera un objeto this.
- En la ejecución, vemos cómo da igual que cada hilo adquiera un recurso independiente. Cómo es de clase, cada hilo se bloquea aun cuando otro hilo adquiera otro recurso diferente.



# Bloqueo a nivel de clase

- **Ejemplo de bloqueo de clase.**

```
package metodosinterbloqueos;

/*
 * También se consideraría con método estático
 */
class Recurso{

    public void bloquear(){
        System.out.println ("Soy el hilo cuyo nombre es " + Thread.currentThread().getName() + "\n");

        synchronized(Recurso.class){
            System.out.println (Thread.currentThread().getName() + " Accediendo al recurso y bloqueado \n");
            for (int i=0; i<1000000000;i++);
            System.out.println (Thread.currentThread().getName() + " Deja y desbloquea el recurso \n");
        }
    }
}
```



# Bloqueo a nivel de clase

- **Ejemplo de bloqueo de clase.**

```
public class MetodosDeInterbloqueosClase{  
    public static void main(String argv[]){  
  
        Recurso r1, r2, r3, r4;  
        Hilo hilo1, hilo2, hilo3, hilo4;  
  
        r1 = new Recurso();  
        r2 = new Recurso();  
        r3 = new Recurso();  
        r4 = new Recurso();  
  
        hilo1 = new Hilo(r1);  
        hilo2 = new Hilo(r2);  
        hilo3 = new Hilo(r3);  
        hilo4 = new Hilo(r4);  
  
        hilo1.setName("Hilo 1");  
        hilo2.setName("Hilo 2");  
        hilo3.setName("Hilo 3");  
        hilo4.setName("Hilo 4");  
  
        hilo1.start();  
        hilo2.start();  
        hilo3.start();|  
        hilo4.start();  
  
    }  
}
```



# Bloqueo a nivel de clase

## ■ Ejemplo de bloqueo de clase.

```
usuario@pc9-43:~/Descargas/PSP21_22-main/ejem_tema2/EjemploMetodosInterbloqueosClase/src$ java metodosinterbloqueos.MetodosDeInterbloqueos
Soy el hilo cuyo nombre es Hilo 2

Hilo 2 Accediendo al recurso y bloqueado

Soy el hilo cuyo nombre es Hilo 1

Soy el hilo cuyo nombre es Hilo 4

Soy el hilo cuyo nombre es Hilo 3

Hilo 2 Deja y desbloquea el recurso

Hilo 3 Accediendo al recurso y bloqueado

Hilo 3 Deja y desbloquea el recurso

Hilo 1 Accediendo al recurso y bloqueado

Hilo 1 Deja y desbloquea el recurso

Hilo 4 Accediendo al recurso y bloqueado

Hilo 4 Deja y desbloquea el recurso
```

Actividad propuesta 2.6



# Bloqueo a nivel de Objeto

## ■ Ejemplo de bloque sincronizado

- Una clase controla el acceso a **dos contadores**.
- Si dos hilos de ejecución, intentan acceder a contadores diferentes, aunque estén dentro del mismo objeto de tipo Contadores, no tendrán ningún bloqueo, porque la sincronización es sobre un contador en particular, pero....

```
Users > santi > Documents > trabajo_instituto > basura > Contadores.java > ↗
1  public class Contadores{
2
3      private long cont1 = 0;
4      private long cont2 = 0;
5      private final Object lock1 = new Object();
6      private final Object lock2 = new Object();
7
8      //método que sincroniza sólo sobre el Object lock1
9      public void incrementar1(){
10          synchronized (lock1){
11              cont1++;
12          }
13      }
14      //método que sincroniza sólo sobre el Object lock1
15      public long dameContador1(){
16          synchronized(lock1){
17              return cont1;
18          }
19      }
20
21      //método que sincroniza sólo sobre el Object lock2
22      public void incrementar2(){
23          synchronized (lock2){
24              cont2++;
25          }
26      }
27
28      //método que sincroniza sólo sobre el Object lock2
29      public long dameContador2(){
30          synchronized(lock2){
31              return cont2;
32          }
33      }
34
35  }
```

Opción 1

**Pregunta:** ¿Sería lo mismo Opción 1 y Opción 2, siempre que dos hilos ejecuten diferente método?

```
41  public class Contadores{
42
43      private long cont1 = 0;
44      private long cont2 = 0;
45      private final Object lock1 = new Object();
46      private final Object lock2 = new Object();
47
48
49      public synchronized void incrementar1(){
50          cont1++;
51      }
52
53      public synchronized long dameContador1(){
54          return cont1;
55      }
56
57      public synchronized void incrementar2(){
58          cont2++;
59      }
60
61      public synchronized long dameContador2(){
62          return cont2;
63      }
64
65  }
```

Opción 2

Y esto otro?????. Es absurdo el utilizar Object, porque bloqueamos el this

```
public class Contadores{
    private long cont1 = 0;
    private long cont2 = 0;
    private final Object lock1 = new Object();
    private final Object lock2 = new Object();

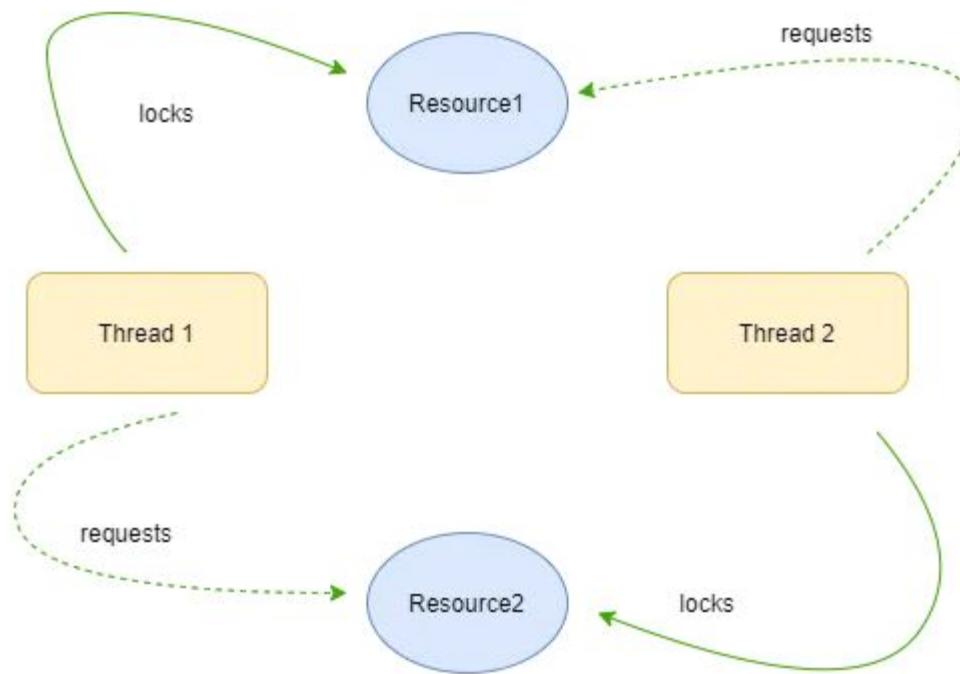
    //método que sincroniza sólo sobre el Object lock1
    public void incrementar1(){
        synchronized (this){
            cont1++;
        }
    }
    //método que sincroniza sólo sobre el Object lock1
    public long dameContador1(){
        synchronized (this){
            return cont1;
        }
    }

    //método que sincroniza sólo sobre el Object lock2
    public void incrementar2(){
        synchronized (this){
            cont2++;
        }
    }
    //método que sincroniza sólo sobre el Object lock2
    public long dameContador2(){
        synchronized (this){
            return cont2;
        }
    }
}
```

Opción 3

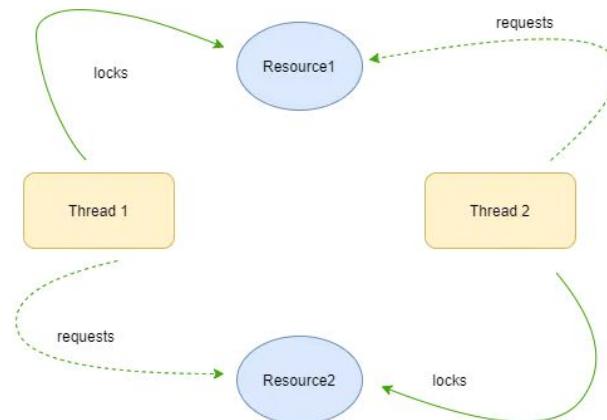
# INTERBLOQUEO THREAD

- Hay que tener cuidado a que el acceso a un recurso o varios, no quede en interbloqueo.
- La siguiente imagen, muestra la situación de dos hilos que se interbloquean a la espera de que alguno deje un recurso que el otro necesita.



# INTERBLOQUEO THREAD

- El Hilo1, requiere del Recursol, por tanto se bloquea hasta que quede libre por el Hilo2.
- El Hilo2, requiere del Recurso2, por tanto se bloquea hasta que quede libre por el Hilo1.
- Ambos quedan esperando a que cada uno libere su recurso y entran en una situación de interbloqueo mutuo.



# INTERBLOQUEO THREAD

- Pongamos un ejemplo sencillo en el que provocaremos una situación de **interbloqueo**:
  - Creamos una clase llamada **Interbloqueo** con dos atributos de tipo Object.
    - Crearemos dos métodos estableciendo un orden diferente de acceso a sus dos atributos, de modo que **accesoOrdenObjeto1Y2**, solicitará acceso al object1 y después al object2.
    - El método **accesoOrdenObjeto2Y1**, solicitará acceso al object2 y después al object1.
    - Ambos métodos se establecerá un acceso **synchronized**.
  - Crearemos una clase **Hilo** que acepte el objeto Interbloqueo y un número que establecerá si ejecuta el método **accesoOrdenObjeto1Y2** o el otro.
  - Desde la clase que implementa el main, creamos el objeto compartido y los dos hilos. El hilo1 llamará a un método y el hilo2 llamará al otro método establecido por el valor de su parámetro pasado en el constructor (1 ó 2).
  - Al ejecutarlo, veremos como se queda totalmente bloqueado.
  - Veremos una forma de mostrar información en el terminal. Antes veamos el código.



# INTERBLOQUEO THREAD

```
package interbloqueohilos;
/*
 * Ejemplo de un interbloqueo forzado
 * entre dos hilos.
 */
class Interbloqueo{
    private Object objeto1 = new Object();
    private Object objeto2 = new Object();

    public void accesoOrdenObjeto1Y2(){
        try{
            synchronized (objeto1){
                Thread.sleep(2000);
                synchronized (objeto2){
                    System.out.println("Acceso a recurso por orden 1 y 2, finalizado\n");
                }
            }
        }catch (InterruptedException e){
            e.printStackTrace();
        }
    }

    public void accesoOrdenObjeto2Y1(){
        try{
            synchronized (objeto2){
                Thread.sleep(2000);
                synchronized (objeto1){
                    Thread.sleep(2000);
                    System.out.println("Acceso a recurso por orden 2 y 1, finalizado\n");
                }
            }
        }catch(InterruptedException e){
            e.printStackTrace();
        }
    }
} //fin de clase Interbloqueo
```



# INTERBLOQUEO THREAD

```
class Hilo implements Runnable{
    private Interbloqueo bloqueo;
    private int numHilo;

    public Hilo(Interbloqueo b, int h){
        this.bloqueo = b;
        this.numHilo = h;
    }

    @Override
    public void run(){
        if (this.numHilo==1)
            bloqueo.accesoOrdenObjeto1Y2();
        else
            bloqueo.accesoOrdenObjeto2Y1();

    }

} //fin de Hilo
```



# INTERBLOQUEO THREAD

```
public class InterbloqueoHilos{
    public static void main (String argv[]){
        System.out.println ("Voy a bloquear a dos hilos al acceso a un recurso\n");
        Interbloqueo objetoQueBloquea = new Interbloqueo();
        Thread h1 = new Thread(new Hilo(objetoQueBloquea, 1));
        Thread h2 = new Thread(new Hilo(objetoQueBloquea, 2));
        h1.start();
        h2.start();
        try{
            h1.join();
            h2.join();
            System.out.println("Este mensaje nunca saldrá\n");
        }catch(InterruptedException e){
            e.printStackTrace();
        }
    }
} //fin de InterbloqueoHilos
```

```
usuario@pc9-43:~/Descargas/PSP21_22-main/ejem_tema2/EjemploInterbloqueoDeHilos/src$ java interbloqueohilos.InterbloqueoHilos
Voy a bloquear a dos hilos al acceso a un recurso
```



# INTERBLOQUEO THREAD

- Para ver que está pasando, sin cerrar la terminal miraremos el pid de nuestro programa java.

```
usuario@pc9-43:~$ jps
27346 Jps
27299 InterbloqueoHilos
23098 lecturaFlujoEntrada
```

- Después mostramos los hilos de dicho proceso en ejecución con su número de pid. Ejecutamos la orden **jstack 27299**

```
"Thread-0":
    at interbloqueohilos.Interbloqueo.accesoOrdenObjeto1Y2(InterbloqueoHilos
.java:18)
        - waiting to lock <0x000000009038cc38> (a java.lang.Object)
        - locked <0x000000009038cc28> (a java.lang.Object)
    at interbloqueohilos.Hilo.run(InterbloqueoHilos.java:61)
    at java.lang.Thread.run(java.base@14/Thread.java:832)

"Thread-1":
    at interbloqueohilos.Interbloqueo.accesoOrdenObjeto2Y1(InterbloqueoHilos
.java:35)
        - waiting to lock <0x000000009038cc28> (a java.lang.Object)
        - locked <0x000000009038cc38> (a java.lang.Object)
    at interbloqueohilos.Hilo.run(InterbloqueoHilos.java:63)
    at java.lang.Thread.run(java.base@14/Thread.java:832)
```

Found 1 deadlock.



# INTERBLOQUEO THREAD

- Es recomendable, que los hilos de ejecución soliciten los recursos en el mismo orden.
  - Esto quiere decir que si dos hilos desean acceder a r1 y r2, para evitar un interbloqueo, ambos solicitarán acceso en el mismo orden, de modo que h1 no podrá quedar bloqueado con r2 y h2 no podrá hacerlo a la espera de r1.
  - En el ejemplo de Acceso a cuentas bancarias por orden, podrá verse (ver **EjemploEvitaInterbloqueoPorOrden** en git).

SIN TERMINAR



# Bloqueo con estado

- El problema anterior, estimaba en el uso de los recursos del procesador, comprobando el bloqueo a los recursos.
- Los objetos, pueden tener una condición para que los hilos puedan acceder a la información compartida.
- Eso quiere decir, que un hilo, dependiendo de una condición podrá bloquearse con **wait()** y una vez dentro de la sección crítica deberá liberarlo con **notify()**
- *Imaginemos un semáforo* que levanta una barrera en un paso a nivel. Cuando llegue un hilo y se encuentre con el semáforo en verde, podrá entrar, pero inmediatamente bloqueará al/los que vayan detrás, por tanto pondrá el semáforo en rojo. Cuando finalice el paso a nivel, deberá desbloquear la barrera poniendo el semáforo en verde.
  - La condición es sólo uno al mismo tiempo, pero:
  - ¿Y si permitimos que pasen tres al mismo tiempo? **La condición habrá cambiado.**



# Bloqueo con estado

- De manera genérica para solicitar el acceso a la zona de exc. mutua:

```
synchronized (objetoBloqueado) {  
    while ( !condicionParaOperacion) {  
        try {  
            objetoBloqueado.wait();  
            .....  
        } catch (InterruptedException ex) {  
            .....  
        }  
    } //fin while  
  
    <operaciones a realizar dentro de la zona de exclusión mutua>  
    .....  
} //fin Acceso exclusión mutua.
```



# Bloqueo con estado

- De manera genérica para **liberar el acceso a la zona de exc. mutua**:

```
synchronized (objetoBloqueado) {  
  
    <operaciones a realizar dentro de la zona de exclusión mutua>  
    .....  
    objetoBloqueado.notify(); ó objetoBloqueado.notifyAll()  
} //fin Acceso exclusión mutua.
```



# Bloqueo con estado

- **Ejemplo sencillo:** Hilos que pueden incrementar o decrementar un contador con las siguientes condiciones:
  - Un contador no puede llegar a ser negativo.
  - Un contador no puede llegar a un valor máximo de 10.
- En nuestro ejemplo, tenemos dos condiciones previas que son:
  - Cuando un hilo decida **incrementar** el contador, **no podrá hacerlo** si tiene valor **igual a 10**, por tanto deberá **bloquearse** a la espera de que dicha condición se cumpla.
  - Cuando un hilo decida **decrementar** el contador, no podrá hacerlo si tiene un **valor igual a 0**, por tanto deberá **bloquearse** a la espera de que dicha condición se cumpla.
  - El objeto compartido, deberá ser una instancia de la clase **ContadorEstado**
  - El hilo al crearse, se le pasará un **valor aleatorio de 0 o 1**, de manera que si es **0, decrementara y si es 1, incrementará**.
  - Creamos un total de **100 hilos de ejecución** donde no sabemos qué harán.



```
class Contador{  
    private int cuenta = 0;  
  
    public Contador(int valorInicial) {  
        this.cuenta = valorInicial;  
    }  
  
    synchronized public int getCuenta() { return cuenta; }  
  
    synchronized public int incrementa(){  
        this.cuenta++;  
        return cuenta;  
    }  
  
    synchronized public int decrementa(){  
        this.cuenta--;  
        return cuenta;  
    }  
}  
  
class hiloIncr implements Runnable{  
    private final String id;  
    private final Contador cont;  
  
    hiloIncr (String id, Contador c){  
        this.id = id;  
        this.cont = c;  
    }  
}
```



```
@Override
public void run() {
    while (true){
        synchronized (this.cont){
            while (this.cont.getConta() > 9){
                System.out.printf("!!!Hilo %s no se puede incrementar, valor contador: %d\n", this.id, this.cont.getConta());
                try{
                    this.cont.wait();
                }catch (InterruptedException ex){}
            }

            this.cont.incrementa();
            this.cont.notify();
            System.out.printf("Hilo %s incrementa, valor contador: %d\n", this.id, this.cont.getConta());
        }
    }
}

class hiloDecr implements Runnable{
    private final String id;
    private final Contador cont;

    hiloDecr(String id, Contador cont) {
        this.id = id;
        this.cont = cont;
    }
}
```



```
public void run() {
    while (true){
        synchronized (this.cont){
            while (this.cont.getConta<span>n</span>(<span>this.id</span>)<1){
                System.out.printf("!!!Hilo %s no se puede decrementar, valor contador: %d\n", <span>this.id</span>, this.cont.getConta<span>n</span>());
                try {
                    this.cont.wait();
                }catch (InterruptedException ex){}
            }
            this.cont.decrementa();
            this.cont.notify();
            System.out.printf("Hilo %s decremente, valor contador: %d\n", <span>this.id</span>, this.cont.getConta<span>n</span>());
        }
    }
}
```

```
public class bloqueoConEstados {
    public static final int NUM_HILOS_INC = 10;
    public static final int NUM_HILOS_DEC = 10;

    public static void main(String[] args) {
        Contador c = new Contador( valorInicial: 0 );
        Thread[] hilosInc = new Thread[ NUM_HILOS_INC ];
        for (int i = 0; i < NUM_HILOS_INC; i++){
            Thread th = new Thread(new hiloIncr( id: "INC"+i,c));
            hilosInc[i] = th;
        }
        Thread[] hilosDec = new Thread[ NUM_HILOS_DEC ];
        for (int i = 0; i < NUM_HILOS_DEC; i++){
            Thread th = new Thread(new hiloDecr( id: "DEC"+i,c));
            hilosDec[i] = th;
        }
        for (int i = 0; i < NUM_HILOS_INC; i++){
            hilosInc[i].start();
        }
        for (int i = 0; i < NUM_HILOS_DEC; i++){
            hilosDec[i].start();
        }
    }
}
```

# Productor/Consumidor

- Estudiaremos un caso de ejemplo mediante dos hilos, uno que produce en una variable o estructura dinámica y otro que consume de dicha variable o estructura dinámica.
- **Crearemos** un recurso con métodos sincronizados que permitan sólo a un objeto el acceso a dicho método.
- **Crearemos** una clase **Hilo** que puede actuar de productor o consumidor.
- De manera **indefinida**, el productor producirá un valor aleatorio y sobreescibirá dicho recurso, siempre y cuando esté nulo (el consumidor lo ha leído y sobreescrito a null).
- Ninguno de los dos hilos, podrán acceder al recurso al mismo tiempo, por tanto al recurso compartido, viene asignada una cola de **hilos bloqueados**.
- Cuando el productor **no pueda sobreescribir** en dicho recurso, **se bloqueará**.
- De la misma forma, cuando el consumidor **no pueda consumir**, **se bloqueará**.
- Ambos hilos, después de acceder a la exclusión mútua, deberá desbloquear o liberar al primer hilo bloqueado en cola.



# Productor/Consumidor

- Clase Recurso:
  - Método **get**
  - Método **set**
  - Método **datoDisponible** que devuelve true, si la variable valor, no es null.

```
package productorconsumidor;
import java.util.Random;

class Recurso<T>{
    private T valor;

    synchronized public T getValor(){
        T valorDevolver = this.valor;
        this.valor = null;
        return valorDevolver;
    }

    synchronized public void setValor(T v){
        this.valor = v;
    }

    synchronized public boolean datoDisponible(){
        return ( (this.valor!=null) ? true: false);
    }
}
```



# Productor/Consumidor

- Clase **Hilo**.
  - Recibe referencia del recurso compartido.
  - productor será **true** (**productor**)
  - productor será **false** (**consumidor**)

```
class Hilo extends Thread{  
    private String nombre;  
    private Recurso rec;  
    boolean productor;  
  
    Hilo(String n, Recurso rec, boolean p){  
        this.nombre = n;  
        this.rec = rec;  
        this.productor = p;  
    }  
}
```



# Productor/Consumidor

```
public void run(){

    while (true){
        if (this.productor){
            //soy productor
            synchronized (this.rec){
                try{
                    while (rec.datosDisponibles()){
                        rec.wait();
                    }
                    //aquí ponemos un valor aleatorio entre 10 y 50;
                    Random r = new Random();
                    int v = r.nextInt(50 - 10 +1 );
                    // Integer valor = v;
                    rec.setValor(v);
                    System.out.println ("Soy el productor con nombre " + this.nombre + " y he producido el valor: " + v);
                    //aquí liberamos a un consumidor.
                    rec.notify();
                }catch(InterruptedException e){
                    System.out.println("Se ha producido una excepcion");
                }
            }
        }
        else{
            //soy consumidor
        }
    }
}
```

# Productor/Consumidor

```
    }else{
        //soy consumidor

        synchronized (this.rec){
            while (!rec.datosDisponibles()){
                try{
                    rec.wait();
                }catch(InterruptedException e){
                    System.out.println("Se ha producido una excepcion");
                }
            } //fin while

            Integer valor = (Integer)rec.getValor();
            System.out.println ("Soy el consumidor con nombre " + this.nombre + " y he consumido el valor: " + valor);
            rec.notify(); //liberamos a un productor
        } //fin syn

    } //fin else

} //fin while
} //fin run
} //fin clase Hilo
```



# Productor/Consumidor

```
public class ProductorConsumidor {  
    Run | Debug  
    public static void main (String argv[]){  
        Recurso<Integer> recurso = new Recurso<Integer>();  
        Hilo productor1, productor2, consumidor1, consumidor2;  
        productor1 = new Hilo("Prod1", recurso, true);  
        consumidor1 = new Hilo("Cons1", recurso, false);  
        // productor2 = new Hilo("Prod2", recurso, true);  
        // consumidor2 = new Hilo("Cons2", recurso, false);  
  
        System.out.println("Lanzamos un productor y un consumidor que deben sincronizarse ");  
        productor1.start();  
        consumidor1.start();  
        // productor2.start();  
        // consumidor2.start();  
    }  
}
```



# Productor/Consumidor

- Al ejecutarlo:
  - Como hay dos hilos, uno que produce y otro que consume, existe un orden de bloqueo en el hilo correspondiente. Cuando uno produce, si viene el consumidor se bloquea hasta que el que está produciendo lo desbloquea. En caso contrario sucede lo mismo, por tanto:

```
Soy el consumidor con nombre Cons1 y he consumido el valor: 33
Soy el productor con nombre Prod1 y he productido el valor: 34
Soy el consumidor con nombre Cons1 y he consumido el valor: 34
Soy el productor con nombre Prod1 y he productido el valor: 23
Soy el consumidor con nombre Cons1 y he consumido el valor: 23
Soy el productor con nombre Prod1 y he productido el valor: 6
Soy el consumidor con nombre Cons1 y he consumido el valor: 6
Soy el productor con nombre Prod1 y he productido el valor: 30
Soy el consumidor con nombre Cons1 y he consumido el valor: 30
Soy el productor con nombre Prod1 y he productido el valor: 25
Soy el consumidor con nombre Cons1 y he consumido el valor: 25
```



# Productor/Consumidor

- ¡Qué sucede si metemos varios productores/consumidores?

```
Hilo productor1, productor2, consumidor1, consumidor2;
productor1 = new Hilo("Prod1", recurso, true);
consumidor1 = new Hilo("Cons1", recurso, false);
productor2 = new Hilo("Prod2", recurso, true);
consumidor2 = new Hilo("Cons2", recurso, false);

System.out.println("Lanzamos un productor y un consumidor que deben sincronizarse ");
productor1.start();
consumidor1.start();
productor2.start();
consumidor2.start();
```



# Productor/Consumidor

- En este caso, el programa se bloquea. No es un interbloqueo. El problema es que, cuando se hace el notify, sólo se notifica a un hilo y éste podría ser uno que no tiene ninguna posibilidad de continuar con su ejecución. Por ejemplo:
- Se lanza **3 productores y 2 consumidores**. P1, P2, P3, C1,C2.
  - Imaginar que entre **P1**, produce.  $x \leftarrow \text{valor}$ .
  - Ahora entra **P2** y no puede producir, por tanto **se bloquea**.
  - Ahora entra **P3** y no puede producir, por tanto **se bloquea**.
  - Ahora llega **C1** que consume pero notifica a uno bloqueado (**P2**)  $x \leftarrow \text{null}$
  - **P2** entra, produce y notifica a otro que en su caso es **P3**.  $x \leftarrow \text{valor}$ 
    - **P3 ya fue notificado y al no poder producir, ya no podrá hacerlo nunca.**
  - Entra **C2** y consume notificando a otro hilo que **NO** fue nunca notificado.  $x \leftarrow \text{null}$
  - Entra **C3** y quiere consumir, pero **se bloquea** porque no hay nada.  
**(Sin embargo, hay un productor que jamás podrá desbloquearse)**  $x \leftarrow \text{null}$
- Para solucionar este caso, lo podemos resolver sencillamente:
  - En vez de hacer un notify, hacer un notifyAll. Desbloquear a todos y el primero que entre en el método **sincronizado**, tendrá el acceso a la exclusión mútua.



# Productor/Consumidor

- Se puede plantear el ejercicio de un recurso que contenga una lista de valores con un máximo (pila), por ejemplo de 10.



# Productor/Consumidor

- Otra solución más elegante es liberar a los hilos del problema del bloqueo y desbloqueo.
- Para ello, el mismo recurso, deberá controlar el bloqueo y desbloqueo de los hilos que intentan acceder a sus métodos.
- Dicha clase recurso, deberá propagar las excepciones y desde los hilos, tratarlos.



# Clases Thread-safe (Prod-Cons)

- Clase **Contenedor<T>**
  - Clase segura para los hilos.
  - Los hilos se desocupan de la sincronización.
  - Es la **Clase Contenedor**, la que tiene que tener los métodos sincronizados.
  - Los hilos, utilizan los métodos seguros de la clase Contenedor<T>
  - Mirar la diferencia entre ArrayList <>y Vector<>.

```
/* Añadido 27-11-22*/
public class Contenedor<T>{

    private T dato;

    //comprueba cuando hay dato.
    public synchronized boolean hayDato(){
        return (dato!=null);
    }

    //Quiero devolver cuando hay dato
    public T get() throws InterruptedException{
        synchronized(this){
            while (!this.hayDato())
                wait();

            T aux = dato;
            dato = null;
            this.notifyAll();
            return aux;
        } //fin sincronización
    }

    //quiero escribir, cuando no hay dato
    public void set(T dat) throws InterruptedException{
        synchronized(this){
            while (this.hayDato())
                wait();
            dato = dat;
            this.notifyAll();
       } //fin sincronización
    }
}
```



# Clases Thread-safe (Prod-Cons)

```
class ProdCons extends Thread{
    private String nombre;
    private Contenedor<Integer> rec;
    boolean productor;

    ProdCons(String n, Contenedor<Integer> rec, boolean p){
        this.nombre = n;
        this.rec = rec;
        this.productor = p;
    }
}
```

J IdHilo.java > 📁 IdHilo

```
1 public class IdHilo {
2     static private long id = 0;
3     synchronized static public long getId(){
4         return id++;
5     }
6 }
7
```

```
public void run(){
    Random r = new Random();
    while (true){
        if (this.productor){
            //soy productor
            for (int i=0; ; i++){
                try{
                    Thread.sleep(10 + r.nextInt(50 - 10 + 1));
                    this.rec.set(i);
                    System.out.printf(format: "Hilo %d con nombre %s, produce valor %d\n", IdHilo.getId(), this.nombre, i);
                }catch (InterruptedException e){
                    System.out.println(x: "Hilo interrumpido");
                }
            }
        }else{
            //soy consumidor
            //soy productor
            Integer dato;
            for (int i=0; ; i++){
                try{
                    Thread.sleep(10 + r.nextInt(50 - 10 + 1));
                    dato = this.rec.get();
                    System.out.printf(format: "Hilo %d con nombre %s, consume valor %d\n", IdHilo.getId(), this.nombre, dato);
                }catch (InterruptedException e){
                    System.out.println(x: "Hilo interrumpido");
                }
            }
        }
    }
} //fin else
```



# Clases Thread-safe (Prod-Cons)

```
/*
 * Añadido. Santi 2022
 */
public class ProductorConsumidoThreadSafe {
    Run | Debug
    public static void main(String[] args) {
        Contenedor<Integer> almacen = new Contenedor<Integer>();
        ProdCons productor1 = new ProdCons(n: "P1", almacen, p: true);
        ProdCons productor2 = new ProdCons(n: "P2", almacen, p: true);
        ProdCons consumidor1 = new ProdCons(n: "C1", almacen, p: false);
        ProdCons consumidor2 = new ProdCons(n: "C2", almacen, p: false);

        productor1.start();
        productor2.start();
        consumidor1.start();
        consumidor2.start();
    }
}
```

```
Hilo 242 con nombre P2, produce valor 63
Hilo 243 con nombre C2, consume valor 63
Hilo 244 con nombre P1, produce valor 58
Hilo 245 con nombre C1, consume valor 58
Hilo 248 con nombre P2, produce valor 64
Hilo 247 con nombre C2, consume valor 59
Hilo 246 con nombre P1, produce valor 59
Hilo 249 con nombre C2, consume valor 64
Hilo 250 con nombre P2, produce valor 65
iMac-de-santiago:consumidor-productor-clase-thread-safe santi$
```



# Clases Thread-safe (Prod-Cons)

Haremos la misma clase, pero de otra forma y quiero que indagueis en las diferencias con la anterior. Os lo dejo a vosotros.

```
Contenedor.java 6 X  
Contenedor.java > Contenedor<T> > get()  
1 //modificado 13/11/24  
2 public class Contenedor <T>{  
3     private T dato;  
4  
5     public boolean hayDatos() { return (dato!=null);}  
6     synchronized void metodo1(){  
7     synchronized void metodo2(){  
8  
9     public T get() throws InterruptedException {  
10         synchronized(this){  
11             while (!hayDatos())  
12                 wait();  
13         }  
14         //zona de ex. mutua  
15         T aux = dato;  
16         dato = null;  
17         metodo1();  
18  
19         synchronized(this){notifyAll();}  
20         return dato;  
21 }
```

```
22  
23     public void set(T dat) throws InterruptedException {  
24         synchronized(this){  
25             while (hayDatos())  
26                 wait();  
27         }  
28         //zona de ex. mutua  
29         dato = dat;  
30         metodo2();  
31         synchronized(this){notifyAll();}  
32     }  
33 }  
34  
35 } //fin clase|
```



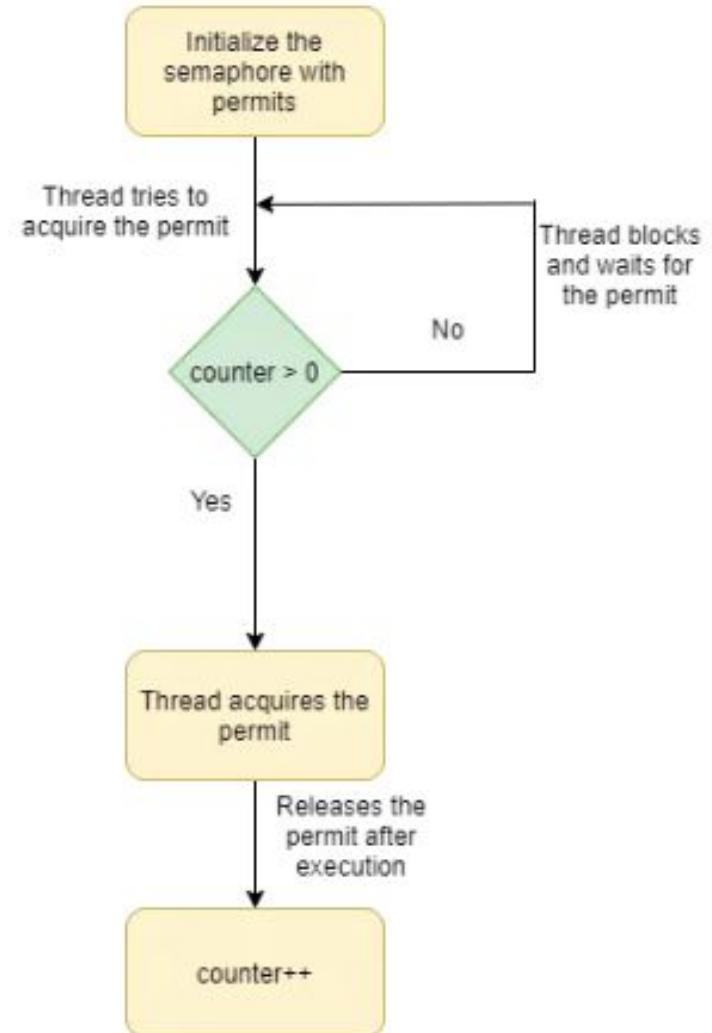
# Sincronización con semáforos

- A diferencia con los métodos anteriores, los hilos de ejecución implementan los mecanismos de sincronización a la exclusión mútua.
- Un **Semaphore** (semáforo) es una técnica de sincronización mediante una variable contador, que hace del acceso a la sección crítica, como si fuera una **operación atómica**.
- El semáforo se puede inicializar al valor que queramos y ello permite que puedan acceder varios hilos al mismo tiempo.
  - Sería el caso de un puente, donde sólo acepta un número máximo de coches al mismo tiempo.
  - Conforme los coches salgan, van desbloqueando a los hilos en espera tras su bloqueo porque el contador llegó a 0.
- **Semaphore consumidor = new Semaphore(MAX);**
- Veremos los métodos de la clase **Semaphore**:
  - **Semaphore.acquire()**.- Solicita el acceso a la exclusión mútua, decrementando el contador siempre que sea mayor que 0. En caso contrario, el hilo se bloqueará hasta que éste sea incrementado.
  - **Semaphore.release()**.- Incrementa el contador y libera a un hilo bloqueado.



# Sincronización con semáforos

- El semáforo lo debemos inicializar a un máximo.
- Si un hilo solicita el acceso a la sección crítica.
  - Podrá entrar si el contador es  $> 0$
  - Deberá bloquearse hasta que el contador sea mayor que 0.
- Los semáforos incorporan una cola de hilos bloqueados.



# Sincronización con semáforos

**Ejemplo:** Implementaremos el productor/consumidor mediante semáforos.

- Existirá un **máximo** de productos en nuestro **almacén**.
- Se podrá **consumir**, si existen productos.
- Tendremos una variable compartida que indicará las **existencias**.
- Necesitamos:
  - Un semáforo **productor** con un número **MAX a producir**.
  - Un semáforo **consumidor** para controlar las extracciones. Lo inicializaremos a 0, porque aún no hay nada en el almacén.
  - Un semáforo **mutex** inicializado a 1, para tener la cuenta de los productores (incrementando) y de los consumidores (decrementando). Queremos saber la disponibilidad de productos que hay en el almacén. Sólo 1 puede acceder al mismo tiempo a esa variable.
- Haremos tres clases, **Almacén**, **Productor**, y **Consumidor**.



# Sincronización con semáforos

- **Clase Almacén:**
  - **Método productir**
    - Un hilo deberá solicitar el acceso al semáforo producir.
    - Si no llega al máximo, podrá producir. Lo haremos informando sólo en pantalla que acaba de productir, pero podríamos utilizar cualquier colección como un arrayList <Producto>.
    - Para incrementar el número de elementos producidos, necesitamos nuestro mutex.
    - Cuando se produce, hay que incrementar el semáforo de consumidores.
  - **Método consumir**
    - Un hilo deberá solicitar el acceso al semáforo consumir.
    - Si es mayor que 0, podrá consumir. Lo que haremos es imprimir en pantalla que está consumiendo.
    - Volveremos a decrementar nuestro número de productos disponibles, accediendo al contador mediante nuestro mutex.
    - Cuando se consume, hay que incrementar el semáforo de productores porque pueden volver a almacenar 1 producto más en el almacén.



# Sincronización con semáforos

- **Clase Almacén:**

```
/*
 * @PSP2021/2022
 */
public class Almacen {

    private final int MAX_LIMITE = 20;
    private int producto = 0;
    private Semaphore productor = new Semaphore(MAX_LIMITE);
    private Semaphore consumidor = new Semaphore(0);
    private Semaphore mutex = new Semaphore(1);

    public void producir(String nombreProductor) {
        System.out.println(nombreProductor + " intentando almacenar un producto");
        try {

            productor.acquire();
            System.out.println(nombreProductor + " almacena un producto. ");

            mutex.acquire();
            producto++;

            mutex.release();
            Thread.sleep(500);

        } catch (InterruptedException ex) {
            Logger.getLogger(Almacen.class.getName()).log(Level.SEVERE, null, ex);
        } finally {
            consumidor.release();
        }
    }
}
```



# Sincronización con semáforos

- **Clase Almacén:**

```
public void consumir(String nombreConsumidor) {
    System.out.println(nombreConsumidor + " intentando retirar un producto");
    try {

        consumidor.acquire();
        System.out.println(nombreConsumidor + " retira un producto. ");

        mutex.acquire();
        producto--;
        mutex.release();

        Thread.sleep(500);
    } catch (InterruptedException ex) {
        Logger.getLogger(Almacen.class.getName()).log(Level.SEVERE, null, ex);
    } finally {
        productor.release();
    }
}
```



# Sincronización con semáforos

- **Clase Productor y Consumidor.**

```
public class Consumidor extends Thread{
    private Almacen almacen;

    public Consumidor(String name, Almacen almacen) {
        super(name);
        this.almacen = almacen;
    }

    @Override
    public void run() {
        while(true){
            almacen.consumir(this.getName());
        }
    }
}
```

```
package productorconsumidor;

public class Productor extends Thread {

    private Almacen almacen;

    public Productor(String name, Almacen almacen) {
        super(name);
        this.almacen = almacen;
    }

    @Override
    public void run() {
        while (true) {
            almacen.producir(this.getName());
        }
    }
}
```



# Sincronización con semáforos

- Clase para probarlo.

```
package productorconsumidor;

/**
 *
 * PSP2021/2022
 */
public class ProductorConsumidor {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        final int PRODUCTOR = 3;
        final int CONSUMIDOR = 10;

        Almacen almacen = new Almacen();

        for (int i = 0; i < PRODUCTOR; i++) {
            new Productor("Productor " + i, almacen).start();
        }

        for (int i = 0; i < CONSUMIDOR; i++) {
            new Consumidor("Consumidor " + i, almacen).start();
        }
    }
}
```



# Sincronización con semáforos

Os planteo este reto:

- **Implementar una simulación basada en productor/consumidor, en el que atendamos clientes de un supermercado sabiendo que:**
  1. Existirán un máximo de **100 clientes**, de los cuales tendrán un carro de la compra comprendido entre **1 y 10 productos**. Para la simulación de los productos, utilizaremos un array de doubles, comprendido entre **1 y 10** elementos y cada uno entre un valor de **0.1 y 50 euros**.
  2. Existirá una **única cola** de llegada de clientes.
  3. Existirán un total de **5 cajeros** que atenderán a la única cola de clientes.
  4. Queremos que cada cajero contabilice su caja y la caja común.
  5. Realizaremos la simulación con tiempos de llegada de clientes comprendido entre **10 y 30 msg**.
  6. Cada cajero atenderá a su cliente entre **500 y 1000 msg**.
  7. Al final, queremos saber cuánto recauda cada cajero y el total.



# Sincronización con semáforos

## Idea del ejercicio

- **Cliente**
  - Tendrá un carro de compra que será un array de double
  - Tendrá un constructor, en el que genere el carro de compra. Generamos un aleatorio entre 1 y 10 productos, para crear el array de double. Después como valor de cada double, un aleatorio entre 0.1 y 50.0
- **RecaudaciónDiaria** (Recurso compartido)
  - Tendrá una cola de Clientes. Utilizar la estructura de datos **Queue<Cliente> cola = new LinkedList<Cliente>()**.
    - Métodos **add** (para encolar) y método **poll** para Devolver el elemento en cabeza de la cola.
  - **Semáforo** inicializado a **1**, para el acceso a la exc mútua de la caja común.
  - **Semáforo** inicializado a **0**, para el control de los clientes en la cola común. (Aún no han llegado)
  - **recaudaciónTotal** que sea un double. Caja común.



# Sincronización con semáforos

- **RecaudaciónDiaria** (Recurso compartido)
  - Tendrá una **cola** de Clientes. Utilizar la estructura de datos `Queue<Cliente> c = new LinkedList<Cliente>()`.
    - Métodos **add** (para encolar) y método **poll** para Devolver el elemento en cabeza de la cola.
  - **Semáforo** inicializado a **1**, para el **acceso a la exc mútua** de la caja común.
  - **Semáforo** inicializado a **0**, para el **control de los clientes** en la cola común. (Aún no han llegado)
  - **Atributo recaudaciónTotal** que sea un **double**. Caja común.
  - **Tendremos** una variable inicializada al número de **clientes máximos** a atender. Lo utilizaremos para que sepan los cajeros cuando finalizar. Podemos simularlo como el fin de la jornada. Podríamos simularlo mediante tiempo, pero así está bien.
  - **Método encolarCliente (Cliente)**
    - Que agrege un cliente a la cola. Por tanto, `cola.add(cli)`
    - Hacer un **release** del semáforo de **control de clientes**.
  - **Método atenderCliente → devuelve un cliente.**
    - Hacer un **acquire()** sobre el semáforo de **control de clientes**.
    - Después un **poll** de la cola de clientes.
    - Decrementar el valor de **clientes máximos a atender**.
  - **Método que actualiceRecaudaciónComún (cantidad).**
    - Hacemos un **adquire** del semáforo acc. exc. mutua. Después actualizamos la recaudación y volvemos a hacer un **release**.

# Sincronización con semáforos

- **Hilo GeneradorCientes (Actúa de productor)**
  - Contiene el **recurso compartido**
  - En el **run**
    - Para los 100 clientes (en un bucle for)
      - Creamos un Cliente
      - Lo agregamos al recurso compartido
      - Simulamos una espera entre 20 y 50 ms.
- **Hilo Cajero (Actúa como consumidor)**
  - Contiene el **recurso compartido**
  - En el **run**
    - Podemos simular un **bucle** que consulte del recurso compartido si ya llegó a 0 el número de clientes a atender.
      - Sacamos un cliente de la cola, **recurso.atenderCliente()** → Cliente
      - Obtiene el valor de compra de ese cliente y actualiza la caja común.
      - Se duerme por el tiempo de atención.
    - Fuera del bucle, el cajero ha finalizado y por tanto imprime su caja.



# Sincronización con semáforos

- **Clase Supermercado (Hilo principal)**
  - Inicializamos constantes.
  - Creamos el recurso compartido **RecaudacionDiaria** con 100 clientes.
  - Creamos al **productor** (GeneradorClientes) y los lanzamos
  - Creamos a los 5 **consumidores** (Cajeros) y los lanzamos.
  - Esperamos a la finalización de los Cajeros, para imprimir la caja común.



# Interrupción de hilos

- Una interrupción es una indicación a un hilo de que debe detener lo que está haciendo y hacer otra cosa. Depende del programador decidir exactamente cómo responde un hilo a una interrupción, pero es muy común que el hilo termine.
- **Thread.interrupt()**, interrumpe la ejecución de un hilo.

```
main(){  
    Thread a = new Thread();  
    a.start();  
    ...  
    a.interrupt(); //Interrumpimos al hilo a  
}
```

**“Cuando interrumpimos a un hilo, a éste se le activa un flag indicando que le han mandado una interrupción”**



# Interrupción de hilos

- Supondremos que la llamada **interrupt()** activa un flag en la tarea interrumpida de la forma:

```
private boolean flagI = false;  
flagI = true;
```

La tarea interrumpida puede hacer 2 cosas:

1. Capturar una **InterruptedException**
2. Chequear con **isInterrupted()**

- La primera opción es recibir una **InterruptedException**. Esto ocurre cuando la tarea está realizando alguna operación que la tiene en estado **WAITING**. Por ejemplo:

```
try {  
  
    Thread.sleep(s * 1000); //sólo se despertará si estaba WAITING con wait, sleep o join  
  
} catch (InterruptedException e) {  
  
    // habrá que hacer algo  
  
}
```

- Si el hilo **NO** está en estado de **WAITING**, habrá que consultar periódicamente si tiene alguna interrupción pendiente. Eso lo haremos con el método **isInterrupted**.



# Interrupción de hilos

- Si el thread no está esperando, entonces sigue funcionando normalmente, pero podemos comprobar de vez en cuando si hay una interrupción pendiente.
- **boolean interrupted()**
  - Es una llamada a un objeto Thread. **Devuelve TRUE** si el objeto tiene una interrupción pendiente.
  - Podemos utilizarla para **capturar** este tipo de interrupción.



# Interrupción de hilos

- **Ejemplo:**
  - Crearemos **dos hilos diferentes**, uno que puede ser interrumpido tras un estado **WAITING** y el otro que **capturará** la interrupción mirando **su flag de activación** con el método `isInterrupted()`.
  - Desde el **hilo principal**, interrumpimos explícitamente a ambos hilos y veremos su comportamiento. Lo haremos con el método **objetoHilo.interrupt();**
  - Veremos como en el primero se capturará la interrupción **InterruptedException**, mientras que en otro, sin utilizar un **catch(InterruptedException e)**, comprueba su flag e informa sin lanzar la interrupción. Una vez comprobado con **isInterrupted()**, si el **flags** está activo y se ha podido tratar, se resetea.



# Interrupción de hilos

- Ejemplo:

```
package interrupcionhilos;

class HiloInterrumpible extends Thread{
    String nombre;

    HiloInterrumpible(String n){
        this.nombre = n;
    }

    @Override
    public void run(){

        try{

            /*
            Me interrumpo por 1 segundo.
            Desde el main, seguro que me mandan una interrupción mientras duermo !!!!.
            */
            System.out.printf ("Soy el hilo %s%n", this.nombre);
            Thread.sleep(1000);
        }catch(InterruptedException e){
            System.out.println ("Acabo de interrumpir al hilo Interrumpible");
        }
    }
} //fin clase HiloInterrumpible
```



# Interrupción de hilos

- Ejemplo:

```
class HiloNoInterrumpible extends Thread {  
    String nombre;  
  
    HiloNoInterrumpible(String n){  
        this.nombre = n;  
    }  
  
    @Override  
    public void run (){  
        //cuando recibamos una señal de interrupción, la capturamos.  
        while (!Thread.currentThread().isInterrupted()){  
            for (long i=0; i< 1000000000L; i++){  
  
            }  
            System.out.printf ("Soy el hilo %s%n", this.nombre);  
        }  
        System.out.println ("Me acaban de interrumpir, la he capturado y me salgo del while, jaaaa");  
    }  
}
```



# Interrupción de hilos

- Ejemplo:

```
public class InterrupcionDeHilos {  
  
    public static void main(String[] args) {  
        try{  
            HiloInterrumpible h1 = new HiloInterrumpible("Hilo huevon");  
            HiloNoInterrumpible h2 = new HiloNoInterrumpible("Hilo guay");  
            h1.start();  
            h2.start();  
            Thread.sleep(300); //como hilo principal, me duermo 300 msg.  
            h1.interrupt();  
            h2.interrupt();  
  
        }catch(InterruptedException e){  
            System.out.println ("Soy el hilo principal y me acaban de interrumpir....");  
        }  
    }  
}
```



# Interrupción de hilos

- **Como ejercicio:** Imaginar un hilo que cada cierto tiempo, se despierta y hace cualquier tarea, pero a este se le interrumpe para que finalize y se vaya a su casa. Tenemos dos escenarios:
  - Qué pasa si le pilla la interrupción mientras está dormido.
  - Qué pasa si le pilla la interrupción mientras está haciendo la tarea.

**La respuesta es muy fácil:**

1. Capturo la interrupción InterruptedException y dentro finalizo.
2. Compruebo siempre en el while (true) si tengo el flag activo de la interrupted y si es así, finalizo.



# Interrupción de hilos

```
 HiloTrabajado.java > HiloTrabajado > run()
1  /**
2  *PSP 2024/25. Santi
3  */
4  public class HiloTrabajado extends Thread {
5      @Override
6      public void run() {
7          while (true) {
8              try {
9                  Thread.sleep(millis:1000); // ME DUERLO UN RATO. TOCA DESCANSO
10                 System.out.println(x:"Hilo trabajando en la vendimia de Socuelos");
11                 realizarTrabajo();
12
13                 if (Thread.currentThread().isInterrupted()) {
14                     System.out.println(x:"Qué bien, me voy a casa. Buff los riñones..")
15                     break;
16                 }
17
18             } catch (InterruptedException e) {
19                 System.out.println(x:"Que bién, me voy a casa, mientas descansaba..");
20                 System.out.println(x:"Pero también me duelen los riñones igual");
21                 break;
22             }
23         }
24     }
25
26     public void realizarTrabajo(){}
27
28 }
```



# PREGUNTAS???

