# UD2.- PROGRAMACIÓN MULTIHILOS

Programación de Servicios y Procesos

# Índice


- 2.1. Introducción
- 2.2. Qué son los hilos
- 2.3. Clases para la creación de hilos
- 2.4. Estados de un hilo
- 2.5. Gestión de hilos
- 2.6. Gestión de prioridades
- 2.7. Comunicación y sincronización de hilos

# Índice


- **2.1. Introducción**
- 2.2. Qué son los hilos
- 2.3. Clases para la creación de hilos
- 2.4. Estados de un hilo
- 2.5. Gestión de hilos
- 2.6. Gestión de prioridades
- 2.7. Comunicación y sincronización de hilos

# 2.1. Introducción

- En el capítulo anterior se estudió la programación concurrente y cómo se podían realizar programas concurrentes.
- Se hizo una breve introducción al concepto de hilo y las diferencias entre estos y los procesos.
- Recordemos que los hilos **comparten el espacio de memoria** del usuario, es decir, corren dentro del contexto de otro programa.

# 2.1. Introducción

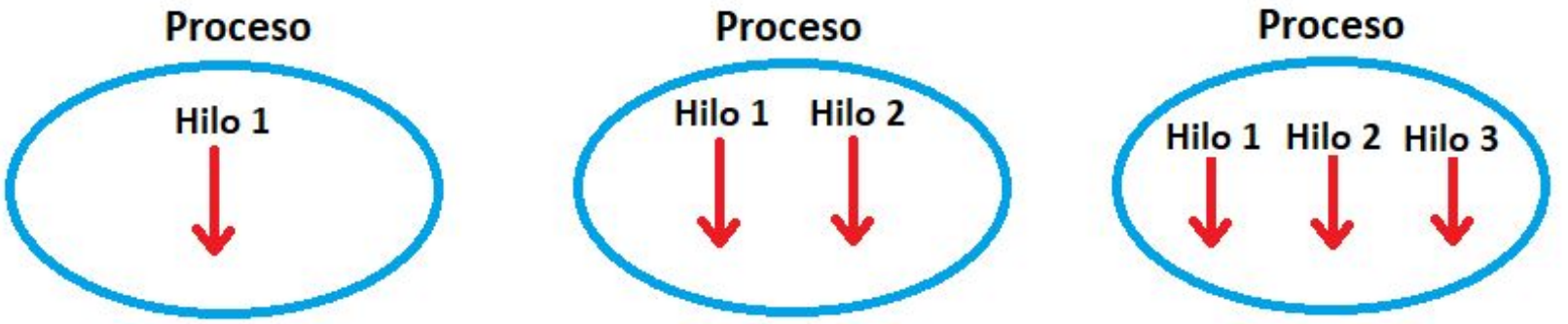
- Los procesos generalmente mantienen su propio **espacio de direcciones** y **entorno de operaciones.**
- Por ello, a los hilos se les conoce a menudo como **procesos ligeros.**
- Vamos a usar los hijos **en java** para realizar programas concurrentes.

# Índice


- 2.1. Introducción
- **2.2. Qué son los hilos**
- 2.3. Clases para la creación de hilos
- 2.4. Estados de un hilo
- 2.5. Gestión de hilos
- 2.6. Gestión de prioridades
- 2.7. Comunicación y sincronización de hilos

# 2.2. Qué son los hilos

- Un **hilo** (**thread** en inglés) es una secuencia de código en ejecución dentro del contexto de un proceso.
- Los hilos no pueden ejecutarse ellos solos, necesitan la supervisión de un proceso padre para ejecutarse.
- Dentro de cada proceso hay varios hilos ejecutándose.



# 2.2. Qué son los hilos

- Podemos usar los hilos para diferentes aplicaciones:
  - Programas que tengan que realizar varias tarea simultáneamente.
    - La ejecución de una parte requiera tiempo y no deba detener el resto del programa.
    - Ejemplos:
      - Un programa que controla sensores en una fábrica, cada sensor puede ser un hilo independiente y recoge un tipo de información (simultánea).
      - Un programa de impresión de documentos debe seguir funcionando, aunque se esté imprimiendo un documento, tarea que se lleva a cabo por medio de un hilo.
      - Un programa procesador de textos puede tener un hilo comprobando la gramática del texto mientras escribo y otro hilo guarda el texto cada cierto tiempo.
      - En un programa de bases de datos un hilo pinta la interfaz gráfica al usuario.
      - En un servidor web, un hilo puede atender las peticiones entrantes y crear un hilo por cada cliente que tenga que servir.

# Índice


- 2.1. Introducción
- 2.2. Qué son los hilos
- **2.3. Clases para la creación de hilos**
- 2.4. Estados de un hilo
- 2.5. Gestión de hilos
- 2.6. Gestión de prioridades
- 2.7. Comunicación y sincronización de hilos

# 2.3. Clases para la creación de hilos

- En java existe dos formas de crear hilos:
  - Extendiendo la clase **Thread.**
  - Implementando la interfaz **runeable.**
- Ambas, partes del paquete **java.lang.**

# 2.3. Clases para la creación de hilos

- 2.3.1. La clase THREAD
  - La forma más simple de añadir funcionalidad de hilo a una clase es extender la clase **Thread.**
  - Esta subclase debe sobrescribir el método **run()** con las acciones que el hilo debe desarrollar.
  - La clase **Thread** define también los métodos **start()** y **stop()** (actualmente en desuso) para iniciar y parar la ejecución del hilo.

# 2.3. Clases para la creación de hilos

- 2.3.1. La clase THREAD
  - La forma general de declarar un hilo extendiendo **Thread** es la siguiente:

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

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

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

  - Y para iniciar su ejecución utilizamos el método **start()**:

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

# 2.3. Clases para la creación de hilos

- 2.3.1. La clase THREAD
  - El siguiente ejemplo declara la clase *PrimerHilo* que extiende la clase **Thread.**
  - Desde el constructor se inicializa una variable numérica que se usará para pintar un número de veces un mensaje.
  - En el método **run()** se escribe la funcionalidad del hilo.
  - Se añade el método **main()** para crear el hilo e iniciar su ejecución con **start().**

# 2.3. Clases para la creación de hilos

- 2.3.1. La clase THREAD

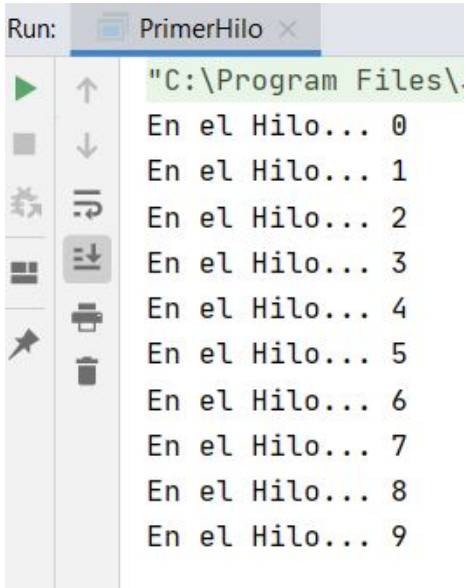
```
public class PrimerHilo extends Thread {
    private int x;

    PrimerHilo(int x) {
        this.x = x;
    }

    public void run() {
        for (int i = 0; i < x; i++)
            System.out.println("En el Hilo... " + i);
    }

    public static void main(String[] args) {
        PrimerHilo p = new PrimerHilo( x: 10);
        p.start();
    }

}
```



# 2.3. Clases para la creación de hilos

## 2.3.1. La clase THREAD

### Ejemplo 2:

```
public class HiloEjemplo1 extends Thread {

    public HiloEjemplo1(String nombre) {
        super(nombre); //para acceder a este nombre desde la clase thread, de la que se hereda
        System.out.println("CREANDO HILO:" + getName());
    }


    public void run() {
        for (int i=0; i<5; i++)
            System.out.println("Hilo:" + getName() + " C = " + i);
    }


    public static void main(String[] args) {
        HiloEjemplo1 h1 = new HiloEjemplo1( nombre: "Hilo 1");
        HiloEjemplo1 h2 = new HiloEjemplo1( nombre: "Hilo 2");
        HiloEjemplo1 h3 = new HiloEjemplo1( nombre: "Hilo 3");

        h1.start();
        h2.start();
        h3.start();

        System.out.println("3 HILOS INICIADOS...");
    }

}
```

```
Run:  HiloEjemplo1
"C:\Program Files\Java\jdk1.8.0_24
CREANDO HILO:Hilo 1
CREANDO HILO:Hilo 2
CREANDO HILO:Hilo 3
3 HILOS INICIADOS...
Hilo:Hilo 2 C = 0
Hilo:Hilo 1 C = 0
Hilo:Hilo 2 C = 1
Hilo:Hilo 2 C = 2
Hilo:Hilo 2 C = 3
Hilo:Hilo 2 C = 4
Hilo:Hilo 3 C = 0
Hilo:Hilo 1 C = 1
Hilo:Hilo 1 C = 2
Hilo:Hilo 3 C = 1
Hilo:Hilo 3 C = 2
Hilo:Hilo 3 C = 3
Hilo:Hilo 3 C = 4
Hilo:Hilo 1 C = 3
Hilo:Hilo 1 C = 4

Process finished with exit code 0
```

# 2.3. Clases para la creación de hilos

- 2.3.1. La clase THREAD
  - Ejemplo 3, podemos definir por un lado la clase hilo y por otro la clase que usa hilo (el resultado de la compilación es el mismo).

```java
public class HiloEjemplo1_V2 extends Thread{
    // constructor
    public  HiloEjemplo1_V2(String nombre) {
        super(nombre);
        System.out.println("CREANDO HILO:" + getName());
    }

    // metodo run
    public void run() {
        for (int i=0; i<5; i++)
            System.out.println("Hilo:" + getName() + " C = " + i);
    }
}
```

```java
public class UsaHiloEjemplo1_V2 {
    public static void main(String[] args) {
        HiloEjemplo1_V2 h1 = new HiloEjemplo1_V2( nombre: "Hilo 1");
        HiloEjemplo1_V2 h2 = new HiloEjemplo1_V2( nombre: "Hilo 2");
        HiloEjemplo1_V2 h3 = new HiloEjemplo1_V2( nombre: "Hilo 3");

        h1.start();
        h2.start();
        h3.start();

        System.out.println("3 HILOS INICIADOS...");
    }
}
```

# 2.3. Clases para la creación de hilos

- 2.3.1. La clase THREAD

| MÉTODOS | MISIÓN |
|---|---|
| **start()** | Hace que el hilo comience la ejecución; la máquina virtual de Java llama al método **run()** de este hilo. |
| **boolean isAlive()** | Comprueba si el hilo está vivo |
| **sleep(long mils)** | Hace que el hilo actualmente en ejecución pase a dormir temporalmente durante el número de milisegundos especificado. Puede lanzar la excepción *InterruptedException*. |
| **run()** | Constituye el cuerpo del hilo. Es llamado por el método **start()** después de que el hilo apropiado del sistema se haya inicializado. Si el método **run()** devuelve el control, el hilo se detiene. Es el único método de la interfaz **Runnable**. |
| **String toString()** | Devuelve una representación en formato cadena de este hilo, incluyendo el nombre del hilo, la prioridad, y el grupo de hilos. Ejemplo: Thread[HILO1,2,main] |
| **long getId()** | Devuelve el identificador del hilo. |
| **void yield()** | Hace que el hilo actual de ejecución pare temporalmente y permita que otros hilos se ejecuten. |
| **String getName()** | Devuelve el nombre del hilo. |
| **setName(String name)** | Cambia el nombre de este hilo, asignándole el especificado como argumento. |

# 2.3. Clases para la creación de hilos

- 2.3.1. La clase THREAD

| MÉTODOS | MISIÓN |
|---|---|
| **int getPriority()** | Devuelve la prioridad del hilo. |
| **setPriority(int p)** | Cambia la prioridad del hilo al valor entero p. |
| **void interrupt()** | Interrumpe la ejecución del hilo |
| **boolean interrupted()** | Comprueba si el hilo actual ha sido interrumpido. |
| **Thread currentThread()** | Devuelve una referencia al objeto hilo que se está ejecutando actualmente. |
| **boolean isDaemon()** | Comprueba si el hilo es un hilo Daemon. Los hilos daemon o demonio son hilos con prioridad baja que normalmente se ejecutan en segundo plano. Un ejemplo de hilo demonio que está ejecutándose continuamente es el recolector de basura (*garbage collector*). |
| **setDaemon(boolean on)** | Establece este hilo como hilo Daemon, asignando el valor *true*, o como hilo de usuario, pasando el valor *false*. |
| **void stop()** | Detiene el hilo. Este método está en desuso. |
| **Thread currentThread()** | Devuelve una referencia al objeto hilo actualmente en ejecución. |
| **int activeCount()** | Este método devuelve el número de hilos activos en el grupo de hilos del hilo actual. |
| **Thread.State getState()** | Devuelve el estado del hilo: NEW, RUNNABLE, BLOCKED, WAITING, TIMED_WAITING, TERMINATED |

# 2.3. Clases para la creación de hilos

- 2.3.1. La clase THREAD
  - Para más información de sobre estos métodos, os dejo este link.
  - El siguiente ejemplo muestra el uso de algunos de los métodos anteriores:

# 2.3. Clases para la creación de hilos

- 2.3.1. La clase THREAD
  - **Actividad 1**.- El siguiente ejemplo muestra el uso de algunos de los métodos anteriores. Localiza cuáles y qué hacen junto al código. Prueba el código en tu PC.

# 2.3. Clases para la creación de hilos

## 2.3.1. La clase THREAD

```
public class HiloEjemplo2 extends Thread {

    public void run() {
        System.out.println(
                "Dentro del Hilo  : " + Thread.currentThread().getName() +
                        "\n\tPrioridad    : " + Thread.currentThread().getPriority() +
                        "\n\tID           : " + Thread.currentThread().getId() +
                        "\n\tHilos activos: " + Thread.currentThread().activeCount());
    }

    public static void main(String[] args) {
        Thread.currentThread().setName("Principal");
        System.out.println(Thread.currentThread().getName());
        System.out.println(Thread.currentThread().toString());


        HiloEjemplo2 h = null;

        for (int i = 0; i < 3; i++) {
            h = new HiloEjemplo2(); //crear hilo
            h.setName("HILO"+i);    //damos nombre al hilo
            h.setPriority(i+1);     //damos prioridad
            h.start();              //iniciar hilo

            System.out.println(
                    "Informacion del " + h.getName() + ": "+ h.toString());
        }
        System.out.println("3 HILOS CREADOS...");
        System.out.println("Hilos activos: " +Thread.activeCount());
    }
}
```

```
Run:  HiloEjemplo2
"C:\Program Files\Java\jdk1.8.0_241\bin\java.exe"
Principal
Thread[Principal,5,main]
Informacion del HILO0: Thread[HILO0,1,main]
Informacion del HILO1: Thread[HILO1,2,main]
Dentro del Hilo  : HILO0
    Prioridad    : 1
    ID           : 12
    Hilos activos: 5
Informacion del HILO2: Thread[HILO2,3,main]
3 HILOS CREADOS...
Hilos activos: 4
Dentro del Hilo  : HILO1
    Prioridad    : 2
    ID           : 13
    Hilos activos: 4
Dentro del Hilo  : HILO2
    Prioridad    : 3
    ID           : 14
    Hilos activos: 2

Process finished with exit code 0
```

# 2.3. Clases para la creación de hilos

- 2.3.1. La clase THREAD
  - La clase **ThreadGroup** se utiliza para manejar grupos de hilos en las aplicaciones java.
  - La clase **Thread** proporciona constructores en los que se puede especificar el grupo del hilo que se está creando en el mismo momento de instanciarlo.
  - Ejemplo para crear un grupo de hilos:

    Thread(grupo ThreadGroup, destino Runnable, nombre String)

# 2.3. Clases para la creación de hilos

## 2.3.1. La clase THREAD

- En el siguiente ejemplo se especifica el grupo de hilos y el nombre del hilo:

```
Run:  HiloEjemplo2Grupos
"C:\Program Files\Java\jdk1.8.0_241\bin\java.exe" ...
Principal
Thread[Principal,5,main]
3 HILOS CREADOS...
Hilos activos: 5
Informacion del hilo: Thread[Hilo 2,5,Grupo de hilos]
Informacion del hilo: Thread[Hilo 1,5,Grupo de hilos]
Hilo 2 Finalizando la ejecución.
Informacion del hilo: Thread[Hilo 3,5,Grupo de hilos]
Hilo 1 Finalizando la ejecución.
Hilo 3 Finalizando la ejecución.

Process finished with exit code 0
```

```java
public class HiloEjemplo2Grupos extends Thread {

    public void run() {
        System.out.println("Informacion del hilo: " + Thread.currentThread().toString());

        for (int i = 0; i < 1000; i++)
            i++;
        System.out.println(Thread.currentThread().getName() + " Finalizando la ejecución.");
    }

    public static void main(String[] args) {
        Thread.currentThread().setName("Principal");
        System.out.println(Thread.currentThread().getName());
        System.out.println(Thread.currentThread().toString());

        ThreadGroup grupo = new ThreadGroup( name: "Grupo de hilos");
        HiloEjemplo2Grupos h = new HiloEjemplo2Grupos();

        Thread h1 = new Thread(grupo, h,  name: "Hilo 1");
        Thread h2 = new Thread(grupo, h,  name: "Hilo 2");
        Thread h3 = new Thread(grupo, h,  name: "Hilo 3");

        h1.start();
        h2.start();
        h3.start();

        System.out.println("3 HILOS CREADOS...");
        System.out.println("Hilos activos: " + Thread.activeCount());
    }
}
```

# 2.3. Clases para la creación de hilos

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

# 2.3. Clases para la creación de hilos

- 2.3.2. La interfaz RUNNABLE
  - Para añadir la funcionalidad de hilo a una clase que deriva de otra clase (por ejemplo un applet), siendo esta distinta de **Thread**, Se utiliza la interfaz **Runnable**.
  - Esta interfaz añade la funcionalidad de hilo a una clase con solo implementarla.
  - Por ejemplo, para añadir la funcionalidad de hilo a un applet definimos la clase como:

    public class reloj **extends applet implements Runnable{}**
  - La interfaz **Runnable** proporciona un único método, el método **run()**. Este es ejecutado por el objeto hilo asociado.

# 2.3. Clases para la creación de hilos

- 2.3.2. La interfaz RUNNABLE
  - La forma general de declarar un hilo implementando la interfaz **Runnable** es la siguiente:

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

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

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

  - Y para iniciar su ejecución utilizaremos el método **start()**:

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

# 2.3. Clases para la creación de hilos

- 2.3.2. La interfaz RUNNABLE
  - O bien para lanzar el hilo escribimos lo anterior en dos pasos:

```
Thread h1 = new Thread(h);
h1.start();
```

  - O en un paso todo:

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

# 2.3. Clases para la creación de hilos

- 2.3.2. La interfaz RUNNABLE
  - Ejemplo:

```
public class PrimerHiloR implements Runnable {
    public void run() {
        System.out.println("Hola desde el Hilo! " +
                Thread.currentThread().getId());
    }
}
```

```
public class UsaPrimerHiloR {
    public static void main(String[] args) {
        //Primer hilo
        PrimerHiloR hilo1 = new PrimerHiloR();
        new Thread(hilo1).start();

        //Segundo hilo
        PrimerHiloR hilo2 = new PrimerHiloR();
        Thread hilo = new Thread(hilo2);
        hilo.start();

        //Tercer Hilo
        new Thread(new PrimerHiloR()).start();
    }
}
```

```
Run:  UsaPrimerHiloR
"C:\Program Files\Java\jdk1.8.0_241\
Hola desde el Hilo! 12
Hola desde el Hilo! 14
Hola desde el Hilo! 13

Process finished with exit code 0
```

# 2.3. Clases para la creación de hilos

- 2.3.2. La interfaz RUNNABLE
  - Vamos a ver cómo usar un hilo en un applet para realizar una tarea repetitiva.
  - Una applet es una aplicación Java que se puede insertar en una página web.
  - Cuando el navegador carga la página, el applet se carga y se ejecuta.

# 2.3. Clases para la creación de hilos

## 2.3.2. La interfaz RUNNABLE

- En un applet se definen varios métodos:
  - **init()**
    - Con instrucciones para inicializar el applet.
    - Es llamado una vez cuando se carga el applet.
  - **start()**
    - Parecido al **init()** pero con la diferencia de que es llamado cuando se reinicia el applet.
  - **paint()**
    - Muestra el contenido del applet.
    - Se ejecuta cada vez que hay que redibujar.
  - **stop()**
    - Invocado para ocultar el applet.
    - Se utiliza para detener hilos.

# 2.3. Clases para la creación de hilos

## 2.3.2. La interfaz RUNNABLE

- Para ejecutar un applet, al no tener método *main()*, se podrá realizar de dos formas:
  - Crear un fichero *nombre.html* con el siguiente contenido:

```
<html>
    <applet code="nombreClase.java" widt="200" height="100">
    </applet>
</html>
```

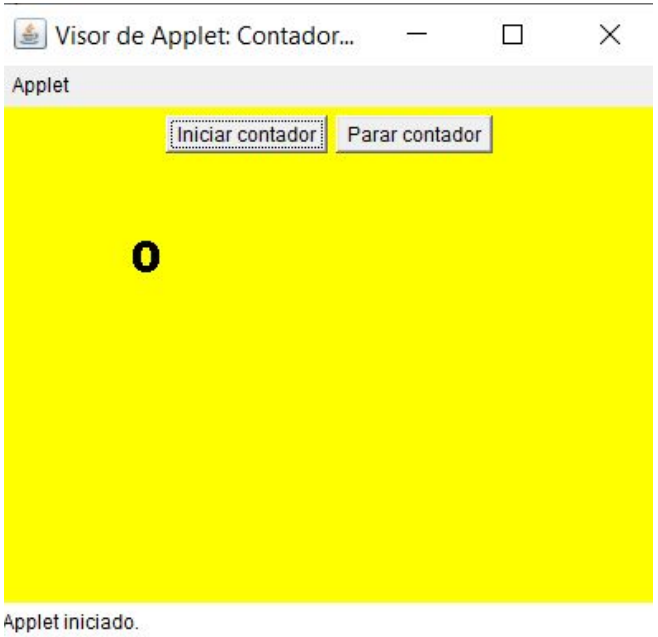
  - Desde un entorno gráfico (eclipse, IntellijIdea, etc), compilandolo desde la línea de comandos usando *appletviewer* o a través del *Applet run/debug configuratión*.
    - *Para ello debe instalar en IntellijIdea un plug-in, llamado **java Applets Support**.*
    - *Entra en **File>Settings** y dentro en **pluggins**.*
    - *Reinicia el programa y ejecuta tu código desde **run…**, seleccionando tu código.*

# 2.3. Clases para la creación de hilos

- 2.3.2. La interfaz RUNNABLE
  - Mostraremos la hora con los minutos y segundos: HH:MM:SS.
  - Para ello, se muestra el código de ejemplo en IntellijIdea.

# 2.3. Clases para la creación de hilos

- 2.3.2. La interfaz RUNNABLE
  - El siguiente ejemplo es un contador.
  - Para ello, se muestra el código de ejemplo en Intellijldea.




# 2.3. Clases para la creación de hilos

- 2.3.2. La interfaz RUNNABLE
  - **Actividad 3.-**
    - Partiendo del ejemplo anterior, separa el hilo en una clase aparte dentro del applet que extienda a **thread.** El applet ahora no impementará a **Runnable,** debe quedar así:
      - public class actividad3 extends Applet implements ActionListener{

```
class HiloContador extends Thread{
    //atributos y métodos
    …
    }//fin clase
//atributos y métodos
…
}//fin actividad3
```

continua…

# 2.3. Clases para la creación de hilos

- 2.3.2. La interfaz RUNNABLE
  - **Actividad 3.-**
    - Se debe crear un applet que lance dos hilos y muestre dos botones para finalizarslos.
    - Define en la clase *HiloContador* un constructor que reciba el valor inicial del contador a partir del cual empezará a contar.
    - El método *getContador()* que devuelva el valor actual del contador.
    - El applet debe crear e iniciar 2 hilos de esta clase, cada uno debe empezar con un valor.
    - Mostrará 2 botones, uno para detener el primer hilo y el otro.
    - Para detener los hilos usa el método **stop()**: *hilo.strop()*.
    - Cambia el texto de los botones cuando se pulsen, que muestre *Finalizado Hilo 1* o *2* dependiendo del botón pulsado.
    - *En el método* ***init()*** prapara la pantalla.
    - En el método **start()** inicia los dos hilos.
    - En el método **paint()** pinta la pantalla.
    - En el método **actionPerformed(ActionEvent e)** controla los botones.
    - En el método **stop()** finaliza los hilos asignándoles el valor *null*.

# Índice


- 2.1. Introducción
- 2.2. Qué son los hilos
- 2.3. Clases para la creación de hilos
- **2.4. Estados de un hilo**
- 2.5. Gestión de hilos
- 2.6. Gestión de prioridades
- 2.7. Comunicación y sincronización de hilos

# 2.4. Estados de un hilo

- Un hilo puede estar en uno de estos estados:
  - **New(Nuevo)**
    - Es el estado cuando se crea un objeto hilo con el operador *new Hilo()*.
    - En este estado el hilo no se ejecuta.
    - El programa no ha comenzado la ejecución del código del método **run()** del hilo.

# 2.4. Estados de un hilo

- Un hilo puede estar en uno de estos estados:
  - **Runnable (Ejecutable)**
    - Cuando se invoca al método **start()**, el hilo pasa a este estado.
    - El SO tiene que asignar tiempo de CPU al hilo para que se ejecute.
    - Por lo tanto, en este estado, el hilo puede estar o no en ejecución.

# 2.4. Estados de un hilo

- Un hilo puede estar en uno de estos estados:
  - **DEAD (Muerto),** un hilo muere por varias razones:
    - Porque el método **run()** finaliza con normalidad.
    - Por invocación del método **stop()**.
      - Este lanza una excepción *ThreadDeath* que mata al hilo.
      - Sin embargo, está en desuso porque cuando un hilo se detiene, inmediatamente **no libera los bloqueos.**
      - La manera segura, usando una variable.
        - Por ejemplo, una variable *stophilo* que se inicializa con valor *false* y que se utiliza dentro de **run()**.
        - Se llamaría al método *paraHilo()* que cambia el valor a *true*.

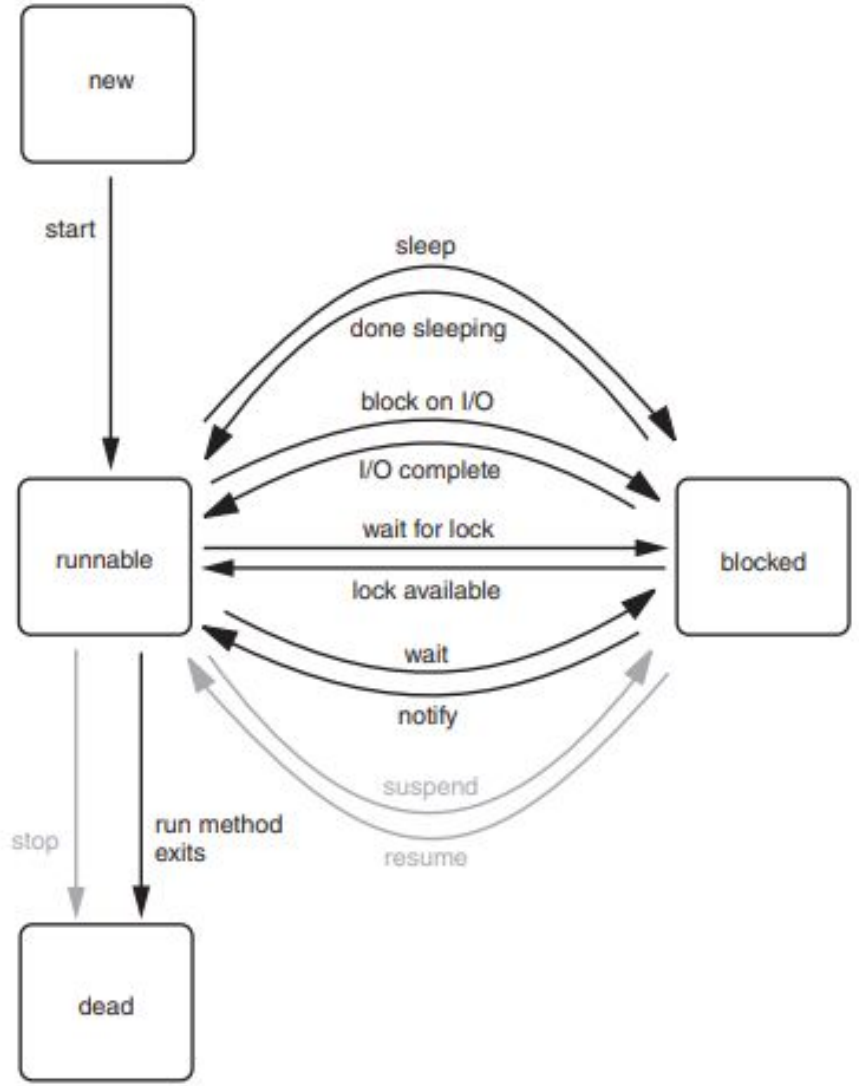
# 2.4. Estados de un hilo

```
1
2   public class HiloEjemploDead extends Thread {
3       private boolean stopHilo= false;
4       public void pararHilo()  {
5           stopHilo = true;
6       }
7       //metodo run
8       public void run() {
9           while (!stopHilo) {
10              System.out.println("En el Hilo");
11          }
12      }
13      public static void main(String[] args) {
14          HiloEjemploDead h = new HiloEjemploDead ();
15          h.start();
16          for(int i=0;i<100000; i++) ;//no hago nada
17
18          h.pararHilo();
19      }// main
20  }//fin clase hilo
```

# 2.4. Estados de un hilo

- Un hilo puede estar en uno de estos estados:
  - **Blocked (Bloqueado)**
    - Alguien llama al método **sleep()** del hilo.
    - El hilo está esperando a que se complete una operación de E/S.
    - El hilo llama al método **wait()**
      - Se quedará esperando hasta que reciba los mensajes **notify()** o **notifyAll()**.
    - El hilo intenta bloquear un objeto que está actualmente bloqueado por otro hilo.
    - Se llama a **suspend()** del hilo.
      - No se volverá a ejecutar hasta que reciba un mensaje **resume()**.

# 2.4. Estados de un hilo



# 2.4. Estados de un hilo

- El método **getState()** devuelve una constante que indica el estado del hilo.
- Los valores son los siguientes:
  - **NEW:** El hilo aún no se ha iniciado.
  - **RUNNABLE:** El hilo se está ejecutando.
  - **BLOCKED:** El hilo está bloqueado.
  - **WAITING:** El hilo está indefinidamente esperando hasta que otro realice una acción (**notify()**).
  - **TIMED_WAITING:** El hilo está esperando que otro hilo realice una acción.
  - **TERMINATED:** El hilo ha finalizado.

# Índice


- 2.1. Introducción
- 2.2. Qué son los hilos
- 2.3. Clases para la creación de hilos
- 2.4. Estados de un hilo
- **2.5. Gestión de hilos**
- 2.6. Gestión de prioridades
- 2.7. Comunicación y sincronización de hilos

# 2.5. Gestión de hilos

- 2.5.1.- Crear y arrancar hilos
  - Para crear un hilo extendemos la clase **Thread**.
  - También implementando la interfaz **Runnable**.
  - El siguiente ejemplo (bien por extensión de thread o implementación de runnable) crea un hilo donde se le pasa por constructor dos variables para inicializarlo:

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

# 2.5. Gestión de hilos

- 2.5.1.- Crear y arrancar hilos
  - Si extiende de **Thread**, se arrancará:

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

  - Si implementa **Runnable**, se arrancará:

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

  - Esto llamará a ejecutarse al método **run()**.

# 2.5. Gestión de hilos

## 2.5.2.- Suspensión de un hilo

- En ejercicios anteriores hemos utilizado el método **sleep()** para dejarlo 'dormido' un número de milisegundos que indiquemos.
- El método **suspend()** permite detener la actividad del hilo durante un intervalo de tiempo indeterminado.
  - Útil cuando se realizan applets con animaciones y, en algún momento, se decide parar la animación.
  - Para volver a activar el hilo se invoca al método **resume()**.
  - Está obsoleto porque puede producir interbloqueos (tanto *suspend()* como *resume()*).

# 2.5. Gestión de hilos

## 2.5.2.- Suspensión de un hilo

- Para suspender de forma segura el hilo se debe introducir una variable y **comprobar su valor dentro del run()**.
- Veamos el siguiente ejemplo:

# 2.5. Gestión de hilos

- 2.5.2.- Suspensión de un hilo

```
public class MyHilo extends Thread{
    private SolicitaSuspender suspender = new SolicitaSuspender();

    public void Suspende(){suspender.set(true);}
    public void Reanuda(){suspender.set(false);}

    public void run(){
        try{
            while(/*haya trabajo por hacer*/){
                suspender.esperandoParaReanudar(); //comprobar
            }
        }catch(InterruptedException exception){

        }
    }
}
```

# 2.5. Gestión de hilos

- 2.5.2.- Suspensión de un hilo

```
public class SolicitaSuspender {
    private boolean suspender;

    public synchronized void set(boolean b){
        suspender = b;
        notifyAll();
    }
    public synchronized void esperandoParaReanudar() throws InterruptedException {
            while(suspender){
                wait(); //espera a recibir notify() o notifyall()
        }
    }
}
```

# 2.5. Gestión de hilos

## 2.5.2.- Suspensión de un hilo

- El método **wait()** sólo puede ser llamado desde dentro de un método sincronizado (**synchronized**).
- Estos tres métodos **wait()**, **notify()**, **notifyAll()** se usan para sincronizar hilos y forman parte de la clase **Object.**
- Sin embargo, **sleep()**, **suspend()** y **resume()** forman parte de **Thread.**

# 2.5. Gestión de hilos

- **Actividad 4.-**
  - Partimos de las clases anteriores *MyHilo* y *SolicitaSuspender*.
  - Vamos a modificar la clase *MyHilo*.
    - Se define una variable contador y se inicia con valor 0.
    - En el método **run()** y dentro de un bucle que controle el fin del hilo mediante una variable, se incrementa en 1 el valor del contador y se visualiza su valor, seincluye también un **sleep()** para que podamos ver los números.
    - Haz una llamada al método *esperandoParaReanudar()* para suspender el hilo, el **sleep()** lo podemos hacer antes o después.
    - Crea en la clase un método que devuelva el valor del contador.
    - Al finalizar el bucle visualiza un mensaje.
  - Para probar las clases crea un método **main()** en el que introducirás una cadena por teclado en un proceso repetitivo hasta introducir un *.
  - Si la cadena introducida es S se supenderá el hilo, si la cadena es R se reanudará el hilo.
  - Al finalizar el proceso repetitivo visualizar el valor del contador, también al finalizar el hilo.
  - Comprueba que todos los mensajes se visualizan correctamente.

# 2.5. Gestión de hilos

- 2.5.3.- Parada de un hilo
  - El método **stop()** detiene la ejecución de un hilo de forma permanente y esta no se puede reanudar con el método **start()**:

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

  - ¿Qué significa el tachado?
    - deprecated (por interbloqueos).

# 2.5. Gestión de hilos

- 2.5.3.- Parada de un hilo
  - El método **interrupt()** envía una petición de interrupción a un hilo.
  - **isInterrupt()** devuelve *true* si ha sido interrumpido, en caso contratio devuelve *false*.
  - Si el hilo se encuentra bloqueado por una llamada a **sleep()** o **wait()** se lanza la excepción *InterruptedException*.
  - A continuación un ejemplo:

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

        System.out.println("FIN HILO");
    }//run


    public void interrumpir() {
        interrupt();
    }//interrumpir

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

```
"C:\Program Files\Java\jdk1.8.0_241\bin\java.exe" ...
En el Hilo
En el Hilo
HA OCURRIDO UNA EXCEPCIÓN
FIN HILO

Process finished with exit code 0
```

```
"C:\Program Files\Java\jdk1.8.0_241\bin\java.exe" ...
En el Hilo
HA OCURRIDO UNA EXCEPCIÓN
FIN HILO

Process finished with exit code 0
```

# 2.5. Gestión de hilos

- 2.5.3.- Parada de un hilo
  - El método **join()** provoca que el hilo que hace la llamada espere la finalización de otros hilos.
  - Ejemplo:

# 2.5. Gestión de hilos

- 2.5.3.- Parada de un hilo

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

```java
public class EjemploJoin {
    public static void main(String[] args)  {
        HiloJoin h1 = new HiloJoin( nom: "Hilo1", n: 2);
        HiloJoin h2 = new HiloJoin( nom: "Hilo2", n: 5);
        HiloJoin h3 = new HiloJoin( nom: "Hilo3", n: 7);

        h1.start();
        h2.start();
        h3.start();

    try {
       h1.join(); h2.join(); h3.join();
   } catch (InterruptedException e) { }

        System.out.println("FINAL DE PROGRAMA");
    }
}//
```

# 2.5. Gestión de hilos

- 2.5.3.- Parada de un hilo
  - Ejecución **con** y **sin join**

```
"C:\Program Files\Java\jdk1.8.0_241\bin\java.exe" ...
Hilo1: 1
Hilo3: 1
Hilo2: 1
Hilo2: 2
Hilo3: 2
Hilo1: 2
Hilo2: 3
Fin Bucle Hilo1
Hilo3: 3
Hilo3: 4
Hilo2: 4
Hilo3: 5
Hilo2: 5
Hilo3: 6
Fin Bucle Hilo2
Hilo3: 7
Fin Bucle Hilo3
FINAL DE PROGRAMA
```

```
"C:\Program Files\Java\jdk1.8.0_241\bin\java.exe" ...
FINAL DE PROGRAMA
Hilo1: 1
Hilo3: 1
Hilo2: 1
Hilo1: 2
Hilo2: 2
Hilo3: 2
Hilo2: 3
Hilo3: 3
Fin Bucle Hilo1
Hilo2: 4
Hilo3: 4
Hilo2: 5
Hilo3: 5
Fin Bucle Hilo2
Hilo3: 6
Hilo3: 7
Fin Bucle Hilo3
```

# 2.5. Gestión de hilos

- **Actividad 5.-**
  - Modifica el applet de la actividad 3 de manera que la finalización de los hilos no se realice con el método **stop()** sino que se realice de alguna de las formas vistas anteriormente (con interrupciones o variables para controlar el fin del hilo).

# Índice


- 2.1. Introducción
- 2.2. Qué son los hilos
- 2.3. Clases para la creación de hilos
- 2.4. Estados de un hilo
- 2.5. Gestión de hilos
- **2.6. Gestión de prioridades**
- 2.7. Comunicación y sincronización de hilos

# 2.6. Gestión de prioridades

- Cada hilo tiene una prioridad.
- Por defecto, un hilo hereda la prioridad del hilo padre que le crea.
- Los métodos **setPriority()** y **getPriority()** nor permite modificar y consultar las prioridades.
- La prioridad es un valor entre 1 y 10.
- 1 es la menor prioridad, **MIN_PRIORITY.**
- 10 es la máxima prioridad, **MAX_PRIORITY.**
- **NORM_PRIORITY** se define como 5.

# 2.6. Gestión de prioridades

- El hilo de mayor prioridad sigue funcionando hasta que:
  - Cede el control llamando llamando al método **yield()** para que otros hilos de igual prioridad se ejecuten.
  - Deja de ser ejecutable.
  - Un hilo de mayor prioridad se convierte en ejecutable.
- El uso del método **yield()** devuelve automáticamente el control al planificador.
- Sin este método el mecanismo de multihilos sigue funcionando pero **algo más lento.**

# 2.6. Gestión de prioridades

```java
public class HiloPrioridad1 extends Thread {
    private int c = 0;
    private boolean stopHilo= false;

    public HiloPrioridad1(String nombre) {
        super(nombre);
    }
    public int getContador()  {
        return c;
    }
    public void pararHilo()  {
        stopHilo = true;
    }
    public void run() {
        while (!stopHilo) {
            try {
                Thread.sleep( millis: 2);
            } catch (Exception e) { }
            c++;
        }
        System.out.println("Fin hilo  "+this.getName());
    }
}
```

```java
public class EjemploHiloPrioridad1 {
    public static void main(String args[]) {
        HiloPrioridad1 h1 = new HiloPrioridad1( nombre: "Hilo1");
        HiloPrioridad1 h2 = new HiloPrioridad1( nombre: "Hilo2");
        HiloPrioridad1 h3 = new HiloPrioridad1( nombre: "Hilo3");

        h1.setPriority(Thread.NORM_PRIORITY);
        h2.setPriority(Thread.MAX_PRIORITY);
        h3.setPriority(Thread.MIN_PRIORITY);

        h1.start();
        h2.start();
        h3.start();

        try {
            Thread.sleep( millis: 10000);
        } catch (Exception e) { }


        h1.pararHilo() ;
        h2.pararHilo() ;
        h3.pararHilo() ;

        System.out.println("h2 (Prioridad Maxima): " + h2.getContador());
        System.out.println("h1 (Prioridad Normal): " + h1.getContador());
        System.out.println("h3 (Prioridad Minima): " + h3.getContador());
    }
}
```

# 2.6. Gestión de prioridades

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

```
"C:\Program Files\Java\jdk1.8.0_241\bin\java.exe" ...
h2 (Prioridad Maxima): 3637
h1 (Prioridad Normal): 3636
h3 (Prioridad Minima): 3634
Fin hilo  Hilo2
Fin hilo  Hilo1
Fin hilo  Hilo3
```

# 2.6. Gestión de prioridades

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

```
"C:\Program Files\Java\jdk1.8.0_241\bin\java.exe" ...
h2 (Prioridad Maxima): 3576
h1 (Prioridad Normal): 3574
h3 (Prioridad Minima): 3575
Fin hilo  Hilo1
Fin hilo  Hilo2
Fin hilo  Hilo3
```

```
"C:\Program Files\Java\jdk1.8.0_241\bin\java.exe"
h2 (Prioridad Maxima): 3616
h1 (Prioridad Normal): 3616
h3 (Prioridad Minima): 3615
Fin hilo  Hilo2
Fin hilo  Hilo3
Fin hilo  Hilo1
```

# 2.6. Gestión de prioridades

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

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

    public static void main(String[] args) {
        EjemploHiloPrioridad2 h1 = new EjemploHiloPrioridad2( nom: "Uno");
        EjemploHiloPrioridad2 h2 = new EjemploHiloPrioridad2( nom: "Dos");
        EjemploHiloPrioridad2 h3 = new EjemploHiloPrioridad2( nom: "Tres");
        EjemploHiloPrioridad2 h4 = new EjemploHiloPrioridad2( nom: "Cuatro");
        EjemploHiloPrioridad2 h5 = new EjemploHiloPrioridad2( nom: "Cinco");
        //asignamos prioridad
        h1.setPriority(Thread.MIN_PRIORITY);
        h2.setPriority(3);
        h3.setPriority(Thread.NORM_PRIORITY);
        h4.setPriority(7);
        h5.setPriority(Thread.MAX_PRIORITY);
        //se ejecutan los hilos
        h1.start();
        h2.start();
        h3.start();
        h4.start();
        h5.start();
    }
}
```

```
"C:\Program Files\Java\jdk1.8.0_241\bin\java.exe" ...
Ejecutando [Dos]
Ejecutando [Tres]
    (Tres: 1)
Ejecutando [Cinco]
    (Cinco: 1)
    (Cinco: 2)
    (Cinco: 3)
    (Cinco: 4)
    (Cinco: 5)
Ejecutando [Cuatro]
Ejecutando [Uno]
    (Uno: 1)
    (Uno: 2)
    (Uno: 3)
    (Uno: 4)
    (Cuatro: 1)
    (Tres: 2)
    (Tres: 3)
    (Tres: 4)
    (Tres: 5)
    (Dos: 1)
    (Dos: 2)
    (Cuatro: 2)
    (Cuatro: 3)
    (Cuatro: 4)
    (Cuatro: 5)
    (Uno: 5)
    (Dos: 3)
    (Dos: 4)
    (Dos: 5)
```

# 2.6. Gestión de prioridades

- Observar que no siempre el hilo con más prioridad es el que antes se ejecuta.
- A la hora de programar hilos con prioridades hemos de tener en cuenta que el comportamiento no está garantizado y dependerá de diferentes factores
  - La plataforma.
  - Aplicaciones en ejecución en ese mismo momento.

# 2.6. Gestión de prioridades

- **Actividad 6.-**
  - Prueba los ejemplos anteriores variando la prioridad y el orden de ejecución de cada hilo.
  - Comprueba los resultados para el primer y segundo ejemplo, coméntalos y analízalos.

# Índice


- 2.1. Introducción
- 2.2. Qué son los hilos
- 2.3. Clases para la creación de hilos
- 2.4. Estados de un hilo
- 2.5. Gestión de hilos
- 2.6. Gestión de prioridades
- **2.7. Comunicación y sincronización de hilos**

# 2.7. Comunicación y sincronización de hilos

- A menudo, los hilos necesitan comunicarse unos con otros.
- La forma de comunicarse consiste, usualmente, en **compartir un objeto**.
- En el siguiente ejemplo, dos hilos comparten un objeto de la clase *contador*.

```
public class Contador {
    private int c = 0;
    Contador(int c) {
        this.c = c;
    }
    public void incrementa() {
        c = c + 1;
    }

    public void decrementa() {
        c = c - 1;
    }
    public int getValor() {
        return c;
    }
}
```

# 2.7. Comunicación y sincronización de hilos

```
public class HiloA extends Thread {
    private Contador contador;
    public HiloA(String n, Contador c) {
        setName(n);
        contador = c;
    }
    public void run() {
        for (int j = 0; j < 300; j++) {
            contador.incrementa();
            try {
                sleep( millis: 100);
                /*para que un hilo se duerma mientras
                el otro hace una operación, así la CPU no realiza de
                una sola vez al completo un hillo y después otro y podemos
                observar mejor el efecto.*/
            } catch (InterruptedException e) {}
        }
        System.out.println(getName() + " contador vale " + contador.getValor());
    }
}
```

# 2.7. Comunicación y sincronización de hilos

```
public class HiloB extends Thread {
    private Contador contador;
    public HiloB(String n, Contador c) {
        setName(n);
        contador = c;
    }
    public void run() {
        for (int j = 0; j < 300; j++) {
            contador.decrementa();
            try {
                sleep( millis: 100);
            } catch (InterruptedException e) {}
        }
        System.out.println(getName() + " contador vale " + contador.getValor());
    }
}
```

# 2.7. Comunicación y sincronización de hilos

- Nos puede dar la impresión que al ejecutar los hilos el valor del contador en el hilo A debería ser 400, ya que empieza en 100 y se le suma 300
- Pero, la ejecución no siempre es lo esperado.

```
public class CompartirInf1 {
    public static void main(String[] args) {
        Contador cont = new Contador( c: 100);
        HiloA a = new HiloA( n: "HiloA", cont);
        HiloB b = new HiloB( n: "HiloB", cont);
        a.start();
        b.start();
    }
}
```

# 2.7. Comunicación y sincronización de hilos

- Con Sleep:

```
"C:\Program Files\Java\jdk1.8.0_241\bin\java.exe" ...
HiloA contador vale 110
HiloB contador vale 110

Process finished with exit code 0
```

```
"C:\Program Files\Java\jdk1.8.0_241\bin\java.exe" ...
HiloA contador vale 126
HiloB contador vale 126

Process finished with exit code 0
```

```
"C:\Program Files\Java\jdk1.8.0_241\bin\java.exe" ...
HiloA contador vale 114
HiloB contador vale 114

Process finished with exit code 0
```

# 2.7. Comunicación y sincronización de hilos

- Sin Sleep:

```
"C:\Program Files\Java\jdk1.8.0_241\bin\java.exe" ...
HiloA contador vale 400
HiloB contador vale 100

Process finished with exit code 0

"C:\Program Files\Java\jdk1.8.0_241\bin\java.exe" ...
HiloA contador vale 398
HiloB contador vale 100

Process finished with exit code 0

"C:\Program Files\Java\jdk1.8.0_241\bin\java.exe" ...
HiloA contador vale 100
HiloB contador vale 100

Process finished with exit code 0
```

# 2.7. Comunicación y sincronización de hilos

- 2.7.1. Bloques sincronizados
  - Una forma de evitar que esto suceda es hacer que las operaciones de incremento y decremento del objeto contador se hagan de forma **atómica.**
  - Es decir, si estamos realizando la suma nos aseguramos que nadie realice la resta hasta que no terminemos la suma.
  - Esto se logra añadiendo **synchronized** a la parte del código que queramos que se ejecute de manera atómica.
  - Java utiliza los **bloques synchronized** para implementar las **regiones críticas**.
  - Veamos un ejemplo:

# 2.7. Comunicación y sincronización de hilos

- 2.7.1. Bloques sincronizados

```
public class Contador {
    private int c = 0;
    Contador(int c) {
        this.c = c;
    }
    public void incrementa() {
        c = c + 1;
    }

    public void decrementa() {
        c = c - 1;
    }
    public int getValor() {
        return c;
    }
}
```

# 2.7. Comunicación y sincronización de hilos

- 2.7.1. Bloques sincronizados

```
public class HiloA extends Thread {
    private Contador contador;

    public HiloA(String n, Contador c) {
        setName(n);
        contador = c;
    }
    public void run() {
        synchronized (contador) {
            for (int j = 0; j < 300; j++) {
                contador.incrementa();
                try {
                    sleep( millis: 100);
                } catch (InterruptedException e) {      }

            }
            System.out.println(getName() + " contador vale "
                    + contador.getValor());
        }
    }
}
```

# 2.7. Comunicación y sincronización de hilos

- 2.7.1. Bloques sincronizados

```
public class HiloB extends Thread {
    private Contador contador;

    public HiloB(String n, Contador c) {
        setName(n);
        contador = c;
    }
    public void run() {
        synchronized (contador) {
            for (int j = 0; j < 300; j++) {
                contador.decrementa();
                try {
                    sleep( millis: 100);
                } catch (InterruptedException e) {      }

            }
            System.out.println(getName() + " contador vale "
                    + contador.getValor());
        }
    }
}
```

# 2.7. Comunicación y sincronización de hilos

- 2.7.1. Bloques sincronizados

```
public class CompartirInf2 {
    public static void main(String[] args) {
        Contador cont = new Contador( c: 100);
        HiloA a = new HiloA( n: "HiloA", cont);
        HiloB b = new HiloB( n: "HiloB", cont);
        a.start();
        b.start();
    }
}
```

```
"C:\Program Files\Java\jdk1.8.0_241\bin\java.exe" ...
HiloA contador vale 400
HiloB contador vale 100

Process finished with exit code 0
```

# 2.7. Comunicación y sincronización de hilos

- 2.7.1. Bloques sincronizados
  - **Actividad 7.-** Crea un programa en Java que lance cinco hilos, cada uno incrementará una variable contador tipo entero, compartida por todos, 5000 veces y luego saldrá. Comprobar el resultado final de la variable. ¿Se obtiene el resultado correcto? Ahora sincroniza el acceso a dicha variable. Lanza los hilos primero mediante la clase **Thread** y luego mediante el interfaz **Runnable**. Comprueba el resultado.

# 2.7. Comunicación y sincronización de hilos

- 2.7.2. Métodos sincronizados
  - Se debe evitar la sincronización de **bloques de código** y sustituirlas siempre que sea posible por la **sincronización de métodos.**
  - **Exclusión mutua** de los procesos respecto a la variable compartida.
  - Por ejemplo,
    - Imaginemos la situación que dos personas comparten una cuenta y pueden sacar dinero de ella en cualquier momento; antes de retirar dinero se comprueba siempre si existe saldo.
    - La cuenta tiene 50€, una de las personas quiere retirar 40 y la otra 30. La primera llega al cajero, revisa el saldo, comprueba que hay dinero y se prepara para retirar el dinero, pero antes de retirarlo llega la otra persona a otro cajero, comprueba el saldo que todavía muestra los 50€ y también se dispone a retirar el dinero.
    - Las dos personas retiran el dinero, pero entonces el saldo actual será ahora de -20€.

# 2.7. Comunicación y sincronización de hilos

- 2.7.2. Métodos sincronizados
  - Para sincronizar un método, añadimos **synchronized** a su declaración, por ejemplo:

```
public class ContadorSincronizado {
    private int c = 0;
    ContadorSincronizado(int c) {
        this.c = c;
    }
    public synchronized void incrementa() {
        c = c + 1;
    }

    public synchronized void decrementa() {
        c = c - 1;
    }
    public synchronized int getValor() {
        return c;
    }
}
```

# 2.7. Comunicación y sincronización de hilos

- 2.7.2. Métodos sincronizados
  - El uso de métodos sincronizados implica que **no es posible invocar dos métodos sincronizados del mismo objeto a la vez.**
  - Cuando un hilo está ejecutando un método sincronizado de un objeto, **los demás hilos** que invoquen a métodos sincronizados para el mismo objeto **se bloquean** hasta que el primer hilo termine con la ejecución del método.
  - Veamos un ejemplo **sin usar métodos sincronizados:**

# 2.7. Comunicación y sincronización de hilos

- 2.7.2. Métodos sincronizados

```
public class Cuenta {
    private int saldo ;
    Cuenta (int s) {saldo = s;}

    int getSaldo() {return saldo;}
    void restar(int cantidad){saldo=saldo-cantidad;}

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

            restar(cant);

            System.out.println("\t"+nom+ " retira =>"+cant + " ACTUAL("+getSaldo()+")" );
        } else {
            System.out.println(nom+ " No puede retirar dinero, NO HAY SALDO("+getSaldo()+")" );
        }
        if (getSaldo() < 0) {
            System.out.println("SALDO NEGATIVO => "+getSaldo());
        }
    }
}
```

# 2.7. Comunicación y sincronización de hilos

- 2.7.2. Métodos sincronizados

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

# 2.7. Comunicación y sincronización de hilos

- 2.7.2. Métodos sincronizados

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

        h1.start();
        h2.start();
    }
}
```

# 2.7. Comunicación y sincronización de hilos

- 2.7.2. Métodos sincronizados

```
"C:\Program Files\Java\jdk1.8.0_241\bin\java.exe" ...
Ana: SE VA A RETIRAR SALDO (ACTUAL ES: 40)
Juan: SE VA A RETIRAR SALDO (ACTUAL ES: 40)
    Juan retira =>10 ACTUAL(30)
Juan: SE VA A RETIRAR SALDO (ACTUAL ES: 30)
    Ana retira =>10 ACTUAL(30)
Ana: SE VA A RETIRAR SALDO (ACTUAL ES: 30)
    Juan retira =>10 ACTUAL(20)
Juan: SE VA A RETIRAR SALDO (ACTUAL ES: 10)
    Ana retira =>10 ACTUAL(10)
Ana: SE VA A RETIRAR SALDO (ACTUAL ES: 10)
    Juan retira =>10 ACTUAL(0)
SALDO NEGATIVO => -10
Juan No puede retirar dinero, NO HAY SALDO(-10)
SALDO NEGATIVO => -10
    Ana retira =>10 ACTUAL(-10)
SALDO NEGATIVO => -10
Ana No puede retirar dinero, NO HAY SALDO(-10)
SALDO NEGATIVO => -10

Process finished with exit code 0
```

# 2.7. Comunicación y sincronización de hilos

- 2.7.2. Métodos sincronizados
  - Para solucionar esto, basta con utilizar **synchronized** en el método *RetirarDinero()*:

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

```
"C:\Program Files\Java\jdk1.8.0_241\bin\java.exe" ...
Ana: SE VA A RETIRAR SALDO (ACTUAL ES: 40)
    Ana retira =>10 ACTUAL(30)
Ana: SE VA A RETIRAR SALDO (ACTUAL ES: 30)
    Ana retira =>10 ACTUAL(20)
Ana: SE VA A RETIRAR SALDO (ACTUAL ES: 20)
    Ana retira =>10 ACTUAL(10)
Ana: SE VA A RETIRAR SALDO (ACTUAL ES: 10)
    Ana retira =>10 ACTUAL(0)
Juan No puede retirar dinero, NO HAY SALDO(0)
Juan No puede retirar dinero, NO HAY SALDO(0)
Juan No puede retirar dinero, NO HAY SALDO(0)
Juan No puede retirar dinero, NO HAY SALDO(0)
```

# 2.7. Comunicación y sincronización de hilos

## 2.7.2. Métodos sincronizados

**Actividad 8.-**

- Crea una clase de nombre *Saldo,* con un atributo que nos indica el saldo, el constructor que da un valor inicial al saldo. Crea varios métodos uno para obtener el saldo y otro para dar valor al saldo, en estos dos métodos añade un **sleep()** aleatorio. Y otro método que reciba una cantidad y se la añada al saldo, este método debe informar de quién añade cantidad al saldo, la cantidad que añade, el estado inicial del saldo (antes de añadir la cantidad) y el estado final del saldo después de añadir la cantidad. Define los parámetros necesarios que debe de recibir este método y defínelo como **synchronized.**
- Crea una clase que extienda de **Thread**, desde el método **run()** hemos de usar el método de la clase *Saldo* que añade la cantidad al saldo. Averigua los parámetros que se necesita en el constructor. No debe visualizar nada en la pantalla.
- Crea en el método *main()* un bojeto *Saldo* asignándole un valor inicial. Visualiza el saldo inicial. Crea varios hilos que compartan ese objeto *Saldo*. A cada hilo le damos un nombre y le asignamos una cantidad. Lanzamos los hilos y esperamos a quefinalicen para visualizar el saldo final del objeto *Saldo*. Comprueba los resultados quitando **synchronized** del método de la clase *Saldo* que reciba la cantidad y se la añada al saldo.

# 2.7. Comunicación y sincronización de hilos

- 2.7.2. Bloqueo de Hilos
  - Para mantener cierta coordinación entre los dos hilos, se usan los métodos **wait(), notify()** y **notifyAll()**
    - **wait():** un hilo llama a al método **wait()** de un cierto objeto queda suspendido hasta que otro hilo llame al método **notify()** o **notifyAll()** del mismo objeto.
    - **notify():** despierta sólo a uno de los hilos que realizó una llamada a **wait()** sobre el mismo objeto notificándole que ha habido un cambio de estado sobre el objeto. **Si varios están esperando, solo uno es despertado arbitrariamente.**
    - **notifyAll():** despierta todos los hilos que están esperando al objeto.

# 2.7. Comunicación y sincronización de hilos

- 2.7.2. Bloqueo de Hilos
  - Ejemplo:

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

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

# 2.7. Comunicación y sincronización de hilos

- 2.7.2. Bloqueo de Hilos
  - Ejemplo:

(Sin wait-notify)

```
public class HiloCadena extends Thread {
    private ObjetoCompartido objeto;
    String cad;
    public HiloCadena (ObjetoCompartido c, String s) {
        this.objeto = c;
        this.cad=s;
    }
    public void run() {
        synchronized (objeto) {
            for (int j = 0; j < 10; j++) {
                objeto.PintaCadena(cad);
                /*objeto.notify(); //aviso que ya he usado el objeto
                try {
                    objeto.wait();//esperar a que llegue un notify
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }*/
            }//for
            objeto.notify(); //despertar a todos los wait sobre el objeto
        }//fin bloque synchronized

        System.out.print("\n"+cad + " finalizado");
    }
}
```

# 2.7. Comunicación y sincronización de hilos

- 2.7.2. Bloqueo de Hilos
  - Ejemplo:

(Sin wait-notify)

```
"C:\Program Files\Java\jdk1.8.0_241\bin\java.exe" ...
 A  A  A  A  A  A  A  A  A  A  B  B  B  B
 A  finalizado B  B  B  B  B  B
 B  finalizado

Process finished with exit code 0
```

# 2.7. Comunicación y sincronización de hilos

- 2.7.2. Bloqueo de Hilos
  - Ejemplo:

(con wait-notify)

```
public class HiloCadena extends Thread {
    private ObjetoCompartido objeto;
    String cad;
    public HiloCadena (ObjetoCompartido c, String s) {
        this.objeto = c;
        this.cad=s;
    }
    public void run() {
        synchronized (objeto) {
            for (int j = 0; j < 10; j++) {
                objeto.PintaCadena(cad);
                objeto.notify(); //aviso que ya he usado el objeto
                try {
                    objeto.wait();//esperar a que llegue un notify
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }//for
            objeto.notify(); //despertar a todos los wait sobre el objeto
        }//fin bloque synchronized

        System.out.print("\n"+cad + " finalizado");
    }
}
```

# 2.7. Comunicación y sincronización de hilos

- 2.7.2. Bloqueo de Hilos
  - Ejemplo:

(con wait-notify)

```
"C:\Program Files\Java\jdk1.8.0_241\bin\java.exe" ...
 A  B  A  B  A  B  A  B  A  B  A  B  A  B  A  B  A  B  A  B
 A  finalizado
 B  finalizado
Process finished with exit code 0
```

# 2.7. Comunicación y sincronización de hilos

- 2.7.2. El modelo productor-consumidor
  - Un problema típico de sincronización es el que representa el modelo **Productor-Consumidor.**
  - Se produce cuando uno o más **hilos producen datos** a procesar y **otros** hilos **los cosumen.**
  - El problema surge cuando el **productor** produce datos **más rápido** que el **consumidor** los consuma, dando lugar
    - El consumidor se salte algún dato.
  - Si es el **consumidor** el que consume **más rápido** que el **productor** produce,
    - Puede recoger varias veces el mismo dato
    - Puede no tener datos para recoger
    - Puede detenerse
    - etc

# 2.7. Comunicación y sincronización de hilos

- 2.7.2. El modelo productor-consumidor
  - La solución pasa por utilizar **synchronized, wait(), notify() y notifyAll().**
  - Ejemplo en IntellijIdea.

# 2.7. Comunicación y sincronización de hilos

- 2.7.2. El modelo productor-consumidor
  - **Actividad 9.-** Prueba el ejemplo visto en clase usando 2 consumidores y un productor. La salida debe ser parecida a esta, en el productor se producen todas las iteraciones, en los consumidores no, ya que solo se producen números en 5 iteraciones:

0=> Productor: 1, produce 0

0=> Consumidor: 1, consume 0

0=> Consumidor: 2, consume 1

1=> Productor: 1, produce 1

1=> consumidor: 1, consume 2

2=> Productor: 1, produce 2

1=> consumidor: 2, consume 3

3=> Productor: 1, produce 3

2=> consumidor: 1, consume 4

4=> Productor: 1, produce 4

Fin productor…

  - Modifica la clase *Producto* para que envíe las cadenas PING y PONG de forma alternativa a la cola. La clase consumidor la toma de la cola. La saluda tiene que mostrar algo como lo siguiente: PING PONG PING PONG PING PONG PING PONG PING PONG …