

# UD2.- Programación multihilo

---

## MANERAS DE EJECUTAR UN HILO

### extends Thread

```
public class UsaHilo {  
    public static void main(String[] args) {  
        HiloSimple hs = new HiloSimple();  
        hs.start();  
    }  
}  
public class HiloSimple extends Thread{  
    public void run() { }  
}
```

### implements Runnable

```
public class UsaHilo2 {  
    public static void main(String[] args) {  
        HiloSimple2 hs = new HiloSimple2();  
        Thread t = new Thread(hs);  
        t.start();  
        // new Thread(new HiloSimple2()).start();  
    }  
}  
public class HiloSimple2 implements Runnable {  
    public void run() { }  
}
```

## APPLET

### extends Thread

```
public class Actividad3 extends Applet implements ActionListener {  
  
    class HiloContador extends Thread {  
        public HiloContador () { }  
        public void run() {  
            while (true) {  
                //sleep  
                //repaint  
                //acciones  
            }  
        }  
    }  
}  
//Finaliza la clase HiloContador
```

```

//Atributos de la clase Actividad3
HiloContador hilo1;

//Se inician los dos hilos
public void start() {
    hilo1 = new HiloContador();
    hilo2 = new HiloContador();
    hilo1.start();
    hilo2.start();
}

// El init es llamado una vez cuando se carga el Applet
public void init() {
    setBackground(Color.pink);
    add(boton = new Button("Texto del botón"));
    boton.addActionListener(this);
    fuente = new Font("Calibri", Font.PLAIN, 26);
}

//Pinta la pantalla
public void paint(Graphics g) {
    g.clearRect(0, 0, 400, 400);
    g.setFont(fuente);
    g.drawString("Texto", 130, 100);
}

//Control de botones
public void actionPerformed(ActionEvent e) {
    if (e.getSource() == boton) {
        //acciones
    }
}
}

```

#### implements Runnable

```

public class ContadorApplet extends Applet implements Runnable {
    private Thread hilo;

    public void start() { // El start es llamado cuando se reinicia el Applet
        // está vacío porque se inicia en las acciones de botón
    }
    public void init() {}
    public void run() {}
    public void paint (Graphics g) {}

    public void actionPerformed(ActionEvent e) {
        if (e.getSource() == boton) {
            if (h != null && h.isAlive()) { // Si el hilo está corriendo no hago nada
            } else {
                hilo = new Thread(this);
                hilo.start();
            }
        }
    }
}
}

```

## MÉTODOS

MÉTODOS	MISIÓN
<b>start()</b>	Hace que el hilo comience la ejecución; la máquina virtual de Java llama al método <b>run()</b> de este hilo.
<b>boolean isAlive()</b>	Comprueba si el hilo está vivo
<b>sleep(long mills)</b>	Hace que el hilo actualmente en ejecución pase a dormir temporalmente durante el número de milisegundos especificado. Puede lanzar la excepción <i>InterruptedException</i> .
<b>run()</b>	Constituye el cuerpo del hilo. Es llamado por el método <b>start()</b> después de que el hilo apropiado del sistema se haya inicializado. Si el método <b>run()</b> devuelve el control, el hilo se detiene. Es el único método de la interfaz <b>Runnable</b> .
<b>String toString()</b>	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]
<b>long getId()</b>	Devuelve el identificador del hilo.
<b>void yield()</b>	Hace que el hilo actual de ejecución pare temporalmente y permita que otros hilos se ejecuten.
<b>String getName()</b>	Devuelve el nombre del hilo.
<b>setName(String name)</b>	Cambia el nombre de este hilo, asignándole el especificado como argumento.

MÉTODOS	MISIÓN
<b>int getPriority()</b>	Devuelve la prioridad del hilo.
<b>setPriority(int p)</b>	Cambia la prioridad del hilo al valor entero p.
<b>void interrupt()</b>	Interrumpe la ejecución del hilo
<b>boolean interrupted()</b>	Comprueba si el hilo actual ha sido interrumpido.
<b>Thread currentThread()</b>	Devuelve una referencia al objeto hilo que se está ejecutando actualmente.
<b>boolean isDaemon()</b>	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 ( <i>garbage collector</i> ).
<b>setDaemon(boolean on)</b>	Establece este hilo como hilo Daemon, asignando el valor <i>true</i> , o como hilo de usuario, pasando el valor <i>false</i> .
<b>void stop()</b>	Detiene el hilo. Este método está en desuso.
<b>Thread currentThread()</b>	Devuelve una referencia al objeto hilo actualmente en ejecución.
<b>int activeCount()</b>	Este método devuelve el número de hilos activos en el grupo de hilos del hilo actual.
<b>Thread.State getState()</b>	Devuelve el estado del hilo: NEW, RUNNABLE, BLOCKED, WAITING, TIMED_WAITING, TERMINATED

## ESTADOS DE UN HILO

El método **getState()** devuelve una constante que indica el estado del hilo.

**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.

## SUSPENSIÓN DE UN HILO

```
public class SolicitaSuspend () {  
    private boolean suspendido;  
    public synchronized void set (boolean b) {  
        suspendido = true;  
        notifyAll();  
    }  
    public synchronized void esperandoParaReanudar() throws InterruptedException {  
        While (suspendido) {  
            wait();  
        }  
    }  
}
```

## PARADA DE UN HILO

### Stop

El método **stop()** detiene la ejecución de un hilo de forma permanente y esta no se puede reanudar con el método **start()**. Está en desuso porque cuando un hilo se detiene, inmediatamente no libera los bloqueos. La manera segura sería hacerlo con booleanas.

### Interrupción

El método **interrupt()** envía una petición de interrupción a un hilo.

**isInterrupted()** devuelve true si ha sido interrumpido, sino devuelve false.

Si el hilo se encuentra bloqueado por una llamada a **sleep()** o **wait()** se lanza la excepción **InterruptedException**.

## Join

El método **join()** provocar que el hilo que hace la llamada espere la finalización de otros hilos

```
Public class Ejemplo Join {
    Public static void main (String [] args) {
        // creo los 3 hilos
        // start() a cada uno

        Try {
            h1.join(); h2.join(); h3.join();
        } catch (InterruptedException e) {
            // nada
        }

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

## GESTIÓN DE PRIORIDADES

Los métodos **setPriority()** y **getPriority()** nos permiten modificar y consultar las prioridades. 1 es la menor prioridad (MIN\_PRIORITY), 5 (NORM\_PRIORITY) y 10 (MAX\_PRIORITY).

```
h1.setPriority(Thread.NORM_PRIORITY);
```

No siempre el hilo con más prioridad es el que antes se ejecuta. Tenemos que tener en cuenta que el comportamiento no está garantizado y dependerá de diferentes factores como la plataforma o las aplicaciones en ejecución que haya en ese momento.

## COMUNICACIÓN Y SINCRONIZACIÓN DE HILOS

### 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;
    }
}
```

```

public class HiloA extends Thread {
    private Contador contador;

    public HiloA (String n, Contador c) {
        setName(n);
        contador = c;
    }

    public void run() {
        synchronized (contador) {
            for () {
                // llama a incrementa o decrementa
                // sleep()
            }
        }
    }
}

```

### Métodos sincronizados

```

public class Cuenta {
    float saldo;
    float valorMaximo;

    public Cuenta (float saldo, float valorMaximo) {
        this.saldo = saldo;
        this.valorMaximo = valorMaximo;
    }

    public float getSaldo() {
        return saldo;
    }

    public synchronized void ingreso(float cantidadSuma, String nombre) {
        // acciones de ingreso
    }

    public synchronized void reintegro (float cantidadResta, String nombre) {
        // acciones de reintegro
    }
}

```

```

public class Persona extends Thread {
    // atributos nombre y cuenta

    public Persona (String nombre, Cuenta cuenta) {
        // los atributos obtienen su valor
    }

    public void run() {
        // hace un ingreso y un reintegro alternativamente
    }
}

```

## Bloqueo de hilos

Para mantener cierta coordinación entre los dos hilos, se usan los métodos:

**wait()** - el hilo queda suspendido hasta que otro llame al método **notify()** o **notifyAll()** del mismo objeto

**notify()** – despierta sólo a uno de los hilos que hizo un **wait()** sobre el mismo objeto. Si hay varios esperando, sólo uno se despierta de manera arbitraria

**notifyAll()** – despierta todos los hilos que están esperando al objeto

### Modelo Productor – Consumidor

```
public class Productor extends Thread {  
    // atributos cola y nombre  
  
    public Productor(Cola c, int n) {  
        // los atributos obtienen su valor  
    }  
  
    public void run() {  
        bucle {  
            cola.put(cadena)  
            // sleep() al final  
        }  
    }  
}
```

```
public class Consumidor extends Thread {  
    // atributos cola y nombre  
  
    public Consumidor(Cola c, int n) {  
        // los atributos obtienen su valor  
    }  
  
    public void run() {  
        String cadena;  
        bucle {  
            cadena = cola.get();  
        }  
    }  
}
```

```

public class Cola {
    private String cadenaEnCola;
    private boolean disponibleParaConsumir = false; // inicialmente cola vacía

    public synchronized String get() {
        while (!disponibleParaConsumir) {
            // wait()
        }
        disponibleParaConsumir = false;
        notify();
        return cadenaEnCola;
    }

    public synchronized void put(String cadena) {
        while (disponibleParaConsumir) {
            // wait()
        }
        cadenaEnCola = cadena;
        disponibleParaConsumir = true;
        notify();
    }
}

```

## FICHEROS

```

public void leerFichero(String fichero) {

    int contador = 0;
    String linea;

    File archivo = new File(fichero);
    FileReader lector = new FileReader(archivo);
    BufferedReader lectorBuffer = new BufferedReader(lector);

    // cuenta palabras
    while ((linea = lectorBuffer.readLine()) != null) {
        if (!linea.equals("")) { // Si la línea no está vacía
            for (int i = 0; i < linea.length(); i++) {
                if (linea.charAt(i) == 32) {
                    contador++;
                }
            }
            contador++;
        }
    }
}

```

```

// cuenta caracteres
while ((linea = lectorBuffer.readLine()) != null) {
    for (int i = 0; i < linea.length(); i++) {
        contador++;
    }
}

lectorBuffer.close();
lector.close();
}

```

## EXTRAS

```

public static void main(String[] args) throws InterruptedException {
    long tiempoComienzo = System.currentTimeMillis();
    // acciones
    sleep(1000); // sleep que luego le restamos
    long tiempoFin = System.currentTimeMillis(); // tiempo hasta que finaliza
    long tiempoTotal = tiempoFin - tiempoComienzo - 1000; // tiempo total del proceso
}

```

```

public int numeroRandom() {
    int num = (int) Math.floor(Math.random() * maximo + minimo);
    return num;
}

```

```

ChangeListener changeListener = new ChangeListener() {
    public void stateChanged(ChangeEvent event) {
        //acciones
    }
};

```

```

// Se añade el evento a los sliders
sliderHilo1.addChangeListener(changeListener);

```

```

private void cambiarLookAndFeel () {
    try {
        UIManager.setLookAndFeel(FlatDarculaLaf.class.getCanonicalName());
        SwingUtilities.updateComponentTreeUI(this);
    } catch (Throwable e) {
        JOptionPane.showConfirmDialog(this, "Error estableciendo LookAndFeel");
    }
}

```