U2Programacion-Multihilo.pdf

hugoomazariio_

Programación de servicios y procesos

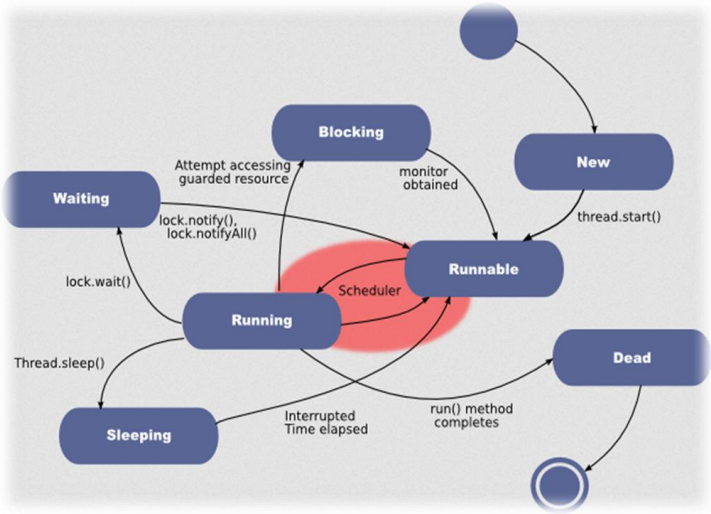
1º Desarrollo de Aplicaciones Multiplataforma

Estudios España

Accede al documento original

ÍNDICE

U2. PROGRAMACIÓN MULTIHILO



1. QUÉ SON LOS HILOS

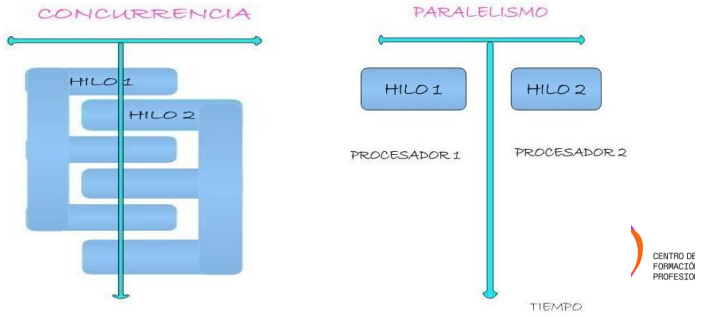
- ✓ **Programación Concurrente** → Creación de programas con varios procesos o hilos que colaboran para ejecutar un trabajo con el mayor rendimiento posible.

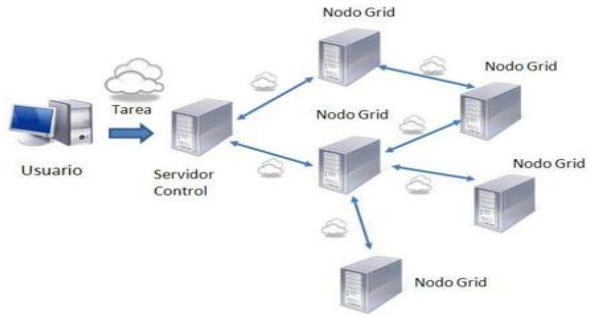
- ✓ **Programación Paralela** → Descomponer un problema complejo en partes dentro del mismo equipo que pueden ejecutarse simultáneamente. Compartición de memoria.

  - ✓ **Multiproceso** → Realizar programas que ejecuten varios procesos a la vez relacionados o no entre sí. Cada proceso es una aplicación independiente.

  - ✓ **Multihilo** → Realizar programas que ejecuten varias tareas dentro de la misma . Objetivo común, estén o no relacionados los hilos.

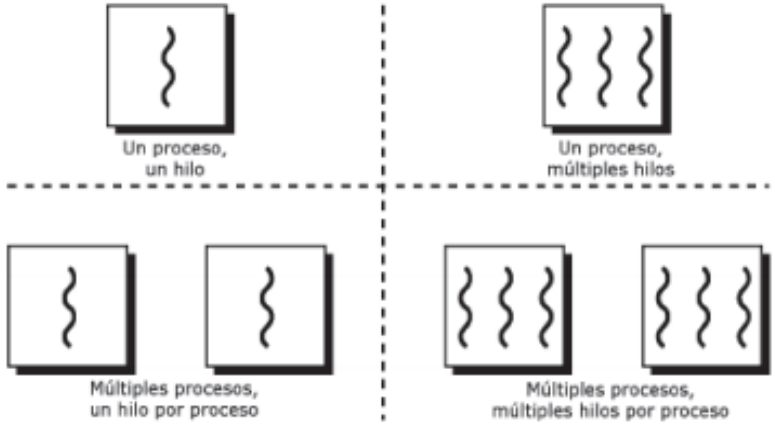
- ✓ **Programación Distribuida** → Programación de software desde ordenadores distintos comunicados entre sí por red, a través del envío de mensajes entre ellos.





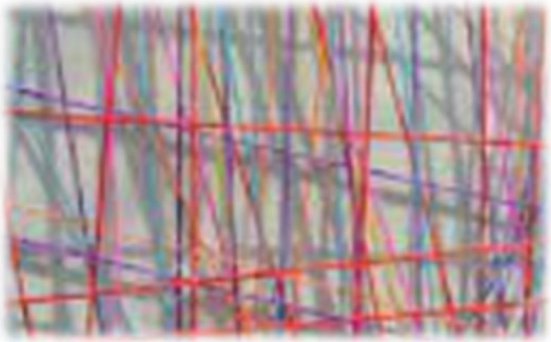
1. QUÉ SON LOS HILOS

- ✓ Los hilos son las tareas de ejecución más simples dentro de un proceso o aplicación suponiendo una secuencia de código dentro del programa.
- ✓ Cuando la ejecución de un programa es secuencial, usará un único hilo llamado main.
- ✓ Podemos tener:
  - ✓ Un proceso / Un hilo.
  - ✓ Un proceso / Varios hilos.
  - ✓ Varios procesos / Un hilo por proceso.
  - ✓ Varios procesos / Varios hilos.



1. QUÉ SON LOS HILOS

- ✓ Cada proceso necesita tener al menos un hilo
- ✓ Un proceso puede tener varios hilos. El proceso termina su ejecución cuando terminan de ejecutarse todos sus hilos.
- ✓ La creación de un hilo nuevo no requiere reservar espacio en la memoria
- ✓ Si un hilo hace un cambio en memoria los demás hilos pueden ver ese cambio
- ✓ Al tener la memoria compartida se necesitan mecanismos de sincronización para evitar conflictos.



msi

BLACK FRIDAY

Portátiles desde
499€

ENCUENTRA
EL TUYO

VER OFERTAS

1. QUÉ SON LOS HILOS

Ventajas sobre los procesos

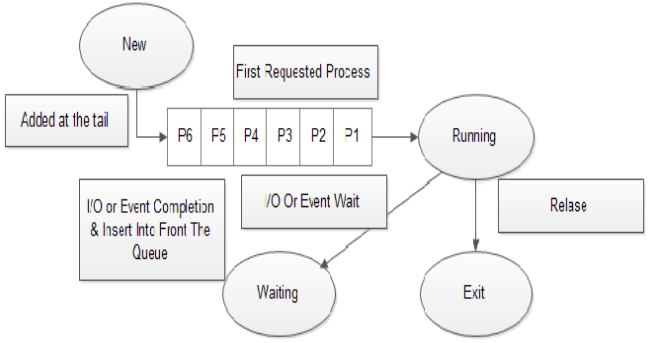
- ✓ Se tarda mucho menos tiempo en crear un hilo nuevo en un proceso existente que en crear un proceso (factor de 10).
- ✓ Se tarda mucho menos en terminar un hilo (contexto y pila) que un proceso (BCP).
- ✓ Se tarda mucho menos tiempo en cambiar entre dos hilos de un mismo proceso.
- ✓ Los hilos aumentan la eficiencia de la comunicación entre programas en ejecución.
- ✓ En la mayoría de los sistemas en la comunicación entre procesos debe intervenir el núcleo para ofrecer protección de los recursos y realizar la comunicación misma. En cambio, entre hilos pueden comunicarse entre sí sin la invocación al núcleo.

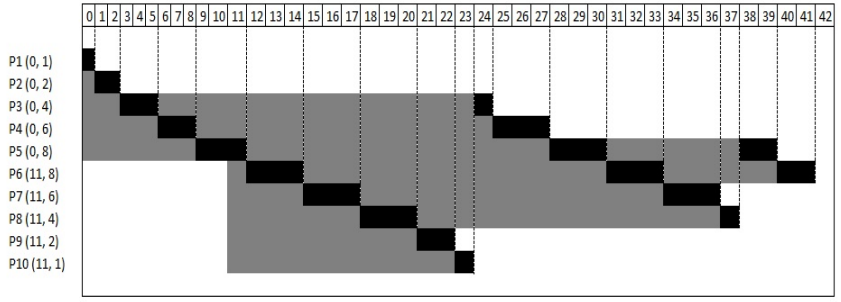
1. QUÉ SON LOS HILOS

Prioridad

La priorización de un hilo sobre otro de cara a su ejecución corre a cargo del SO según los Algoritmos de Planificación. Existen varias opciones:

- ✓ **FCFS (First Come First Served)** → El primer proceso que llegue al procesador se ejecuta en primer lugar y no deja de ejecutarse hasta que termina.
- ✓ **RR (Round Robin)** → Se asigna una cantidad corta de tiempo (quantum) a cada proceso y van rotando por turnos.

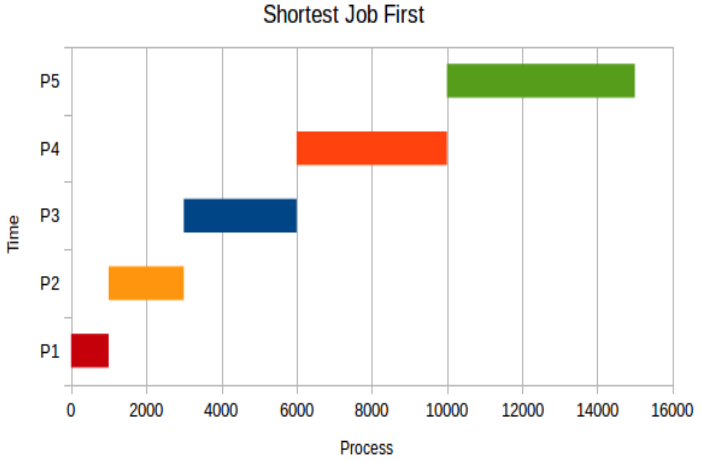


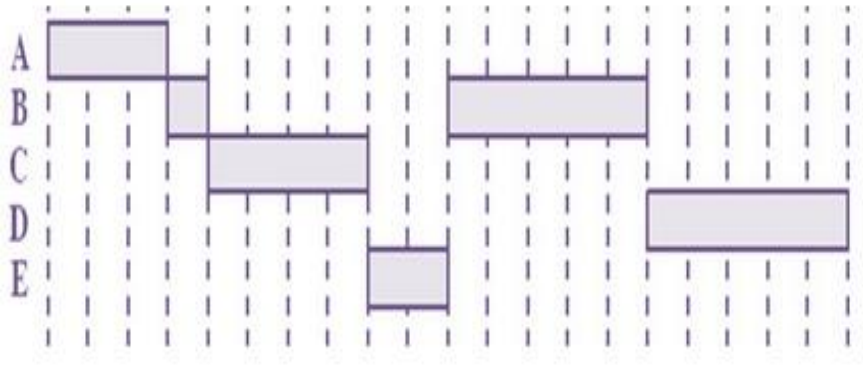


1. QUÉ SON LOS HILOS

Prioridad

- ✓ **SPF (Shortest Process First)** → Primero el proceso más corto de los listos para ejecutarse.
- ✓ **SRT (Shortest Remaining Time)** → De todos los procesos listos para ejecutarse, lo hace primero al que le quede menos tiempo.





2. CLASES PARA LA CREACIÓN DE HILOS

Clase Thread

Extender esta clase es la manera más sencilla de crear hilos.

-start() comienza el hilo.

-run() arranca su ejecución.

Se sobreescribe con lo que queremos que haga el hilo.

```
run:
En el Hilo...0
En el Hilo...1
En el Hilo...2
En el Hilo...3
En el Hilo...4
En el Hilo...5
En el Hilo...6
En el Hilo...7
BUILD SUCCESSFUL (total time: 0 seconds)
```

```java
package Hilos;

/**
 *
 * @author
 */
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(8);
        p.start();
    }
}
```

2. CLASES PARA LA CREACIÓN DE HILOS

Extender subclase Thread.

- **getName()** → para obtener el nombre del hilo.

```
run:
CREANDO HILO: Hilo 1
CREANDO HILO: Hilo 2
CREANDO HILO: Hilo 3
3 HILOS INICIADOS...
Hilo: Hilo 1    C= 0
Hilo: Hilo 1    C= 1
Hilo: Hilo 2    C= 0
Hilo: Hilo 2    C= 1
Hilo: Hilo 2    C= 2
Hilo: Hilo 2    C= 3
Hilo: Hilo 2    C= 4
Hilo: Hilo 1    C= 2
Hilo: Hilo 3    C= 0
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
BUILD SUCCESSFUL (total time: 0 seconds)
```

```java
package Hilos;

/**
 *
 * @author
 */
public class HiloEjemplo1 extends Thread{
    public HiloEjemplo1(String nombre){
        super(nombre);
        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("Hilo 1");
    HiloEjemplo1 h2 = new HiloEjemplo1("Hilo 2");
    HiloEjemplo1 h3 = new HiloEjemplo1("Hilo 3");

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

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

2. CLASES PARA LA CREACIÓN DE HILOS

Métodos de 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.
`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.
`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 currenThread()`	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.
`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, **Thread.State getState()** WAITING, TIMED_WAITING, TERMINATED

2. CLASES PARA LA CREACIÓN DE HILOS

```
package Hilos;

/**...4 lines */
public class HiloEjemplo2 extends Thread{

    public void run(){
        System.out.println(
            "Dentro del Hilo: " + Thread.currentThread().getName()
                    + ", Prioridad: " + Thread.currentThread().getPriority()
                    + ", ID: " + Thread.currentThread().getId()
                    + ",  Hilos activos: " + Thread.currentThread().activeCount());

    }

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

        HiloEjemplo2 h = null;

        for (int i = 0; i < 3; i++){
            h = new HiloEjemplo2();
            h.setName("HILO_" + i);
            h.setPriority(i+1);
            h.start();

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

```
run:
current thread: Principal
toString de current thread: Thread[Principal,5,main]
Información del HILO_0: Thread[HILO_0,1,main]
Información del HILO_1: Thread[HILO_1,2,main]
Información del HILO_2: Thread[HILO_2,3,main]
3 HILOS CREADOS...
Hilos activos: 4
Dentro del Hilo: HILO_2, Prioridad: 3, ID: 12,  Hilos activos: 3
Dentro del Hilo: HILO_1, Prioridad: 2, ID: 11,  Hilos activos: 3
Dentro del Hilo: HILO_0, Prioridad: 1, ID: 10,  Hilos activos: 3
BUILD SUCCESSFUL (total time: 0 seconds)
```

2. CLASES PARA LA CREACIÓN DE HILOS

Crear Grupos de Hilos

-ThreadGroup(“nombre”);

Un hilo siempre pertenece a un grupo.

Si no lo tiene asignado será del grupo principal, main.

```
run:
Principal
Thread[Principal,5,main]
3 HILOS CREADOS...
Hilos activos: 4
Información del hilo: Thread[hilo 1,5,Grupo de hilos]
hilo 1 Finalizando la ejecución
Información del hilo: Thread[hilo 2,5,Grupo de hilos]
hilo 2 Finalizando la ejecución
Información del hilo: Thread[hilo 3,5,Grupo de hilos]
hilo 3 Finalizando la ejecución
BUILD SUCCESSFUL (total time: 0 seconds)
```

```
...5 lines
package Hilos;

/**...4 lines */
public class Grupo_Hilos extends Thread {

    public void run(){
        System.out.println("Información 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("Grupo de hilos");
        Grupo_Hilos h = new Grupo_Hilos();

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

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

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

CORE CENTRO DE FORMACIÓN PROFESIONAL

2. CLASES PARA LA CREACIÓN DE HILOS

Interfaz Runnable

Esta interfaz sólo tiene un método, el run(), donde se incluyen las tareas repetitivas.

Ejemplo con Jframe donde implementamos la interfaz Runnable para añadir funcionalidad de hilo sin usar directamente la clase Thread. Tiene los siguientes métodos:

- ✓ **run**() que hace que se ejecute en un hilo nuevo e incrementa el contador cada segundo.
- ✓ **paint**() se utiliza para actualizar la ventana con el nuevo valor del contador.
- ✓ **main**() crea una instancia de la clase ContadorHilo y la muestra en pantalla.

2. CLASES PARA LA CREACIÓN DE HILOS

Interfaz Runnable

```
public class ContadorHilo extends JFrame implements Runnable {

    private Thread hilo;
    private int cont = 0;
    private boolean parar;

    public ContadorHilo() {
        super(title: "Contador de hilos");
        setDefaultCloseOperation(operation: JFrame.EXIT_ON_CLOSE);
        setBounds(x: 100, y: 100, width: 300, height: 200);
        JButton botonIniciar = new JButton(text: "Iniciar");
        botonIniciar.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                if (hilo == null || !hilo.isAlive()) {
                    hilo = new Thread(task: ContadorHilo.this);
                    hilo.start();
                } else {
                    parar = false;
                }
            }
        });
        JButton botonDetener = new JButton(text: "Detener");
        botonDetener.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                parar = true;
            }
        });
        JLabel etiquetaContador = new JLabel("Contador: " + cont);
        // Colocamos los botones uno al lado del otro
        Box box = Box.createHorizontalBox();
        box.add(comp: botonIniciar);
        box.add(comp: botonDetener);
        add(comp: box);
        this.getContentPane().add(name: BorderLayout.NORTH, comp: box);
            // Colocamos la etiqueta debajo de los botones
        this.getContentPane().add(name: BorderLayout.CENTER, comp: etiquetaContador);
        setVisible(b: true);
    }
```

continúa

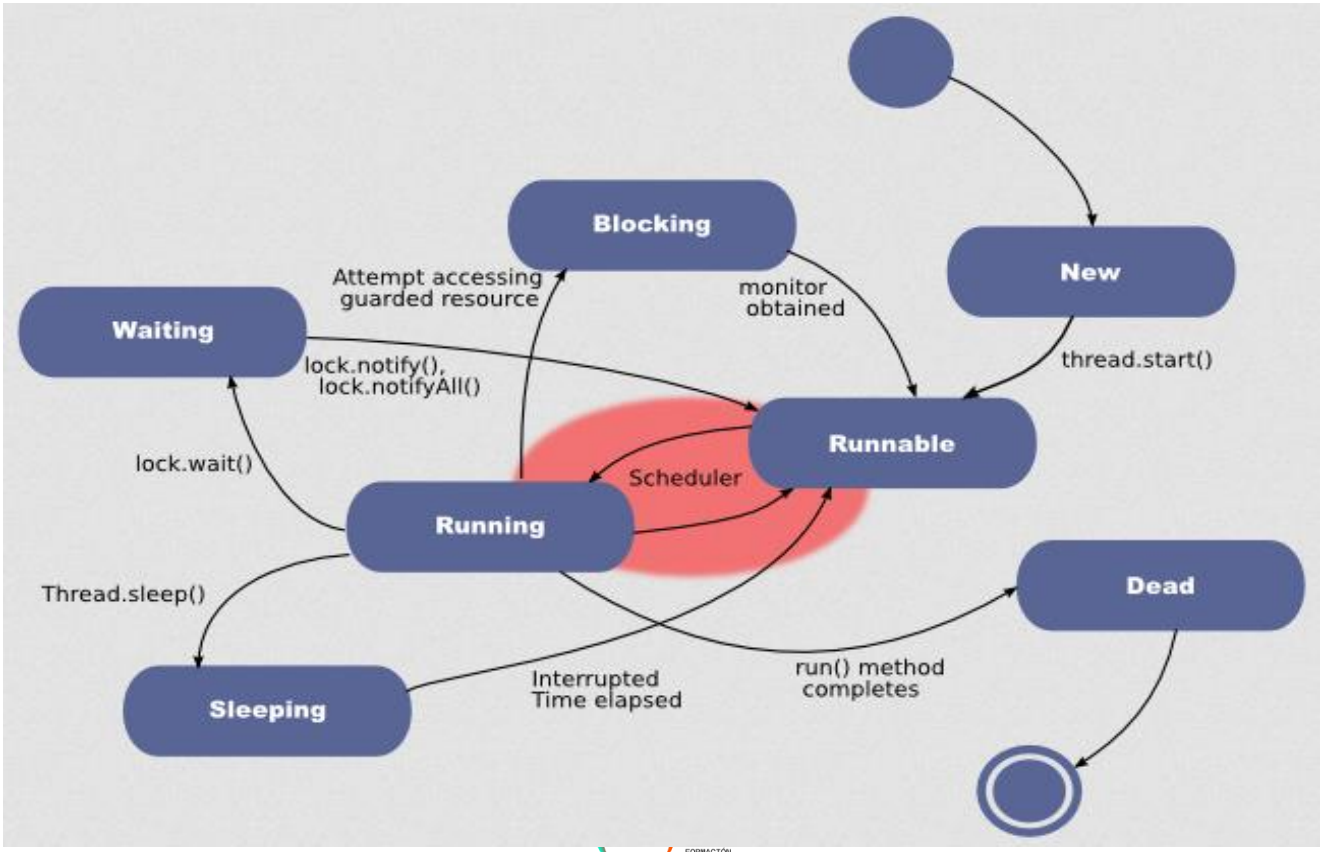
2. CLASES PARA LA CREACIÓN DE HILOS

Interfaz Runnable

```
@Override
public void run() {
    parar = false;
    while (!parar) {
        cont++;
        repaint();
        try {
            Thread.sleep(millis: 1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

public void paint(Graphics g) {
    super.paint(g);
    g.drawString(str:String.valueOf(i: cont), x: 100, y: 100);
}

public static void main(String[] args) {
    new ContadorHilo();
}
```

3. ESTADOS DE UN HILO

Un hilo puede estar en diferentes estados en su ciclo de vida.



3. ESTADOS DE UN HILO

Un hilo puede estar en una de los siguientes estados :

- ✓ **New** (nuevo) : Cuando se crea el hilo con el operador new. Todavía no ha comenzado a ejecutarse su método run().
- ✓ **Runnable** (ejecutable) : Cuando se invoca al método **start**(). El hilo puede estar ejecutándose, si la CPU le asigna tiempo, o no.
- ✓ **Dead** (muerto) : El hilo ha finalizado totalmente porque su método run ha terminado su ejecución, normalmente o porque se ha producido una excepción.
- ✓ **Blocked** (bloqueado) : Hay un motivo que evita la ejecución del hilo:
  - ✓ Otro hilo ha llamado al método **sleep**() del hilo.
  - ✓ Está esperando a que se realice una operación de E/S.
  - ✓ El hilo ejecuta el método **wait**(). Debe recibir un **notify**() para volver a ejecutarse.
  - ✓ El hilo trata de bloquear un objeto que está bloqueado por otro hilo.
  - ✓ Otro hilo ha llamado al método **suspend**() del hilo. Debe recibir un resume() para volver a ejecutarse.

3. ESTADOS DE UN HILO

El método **getState()** devuelve el estado de un hilo:

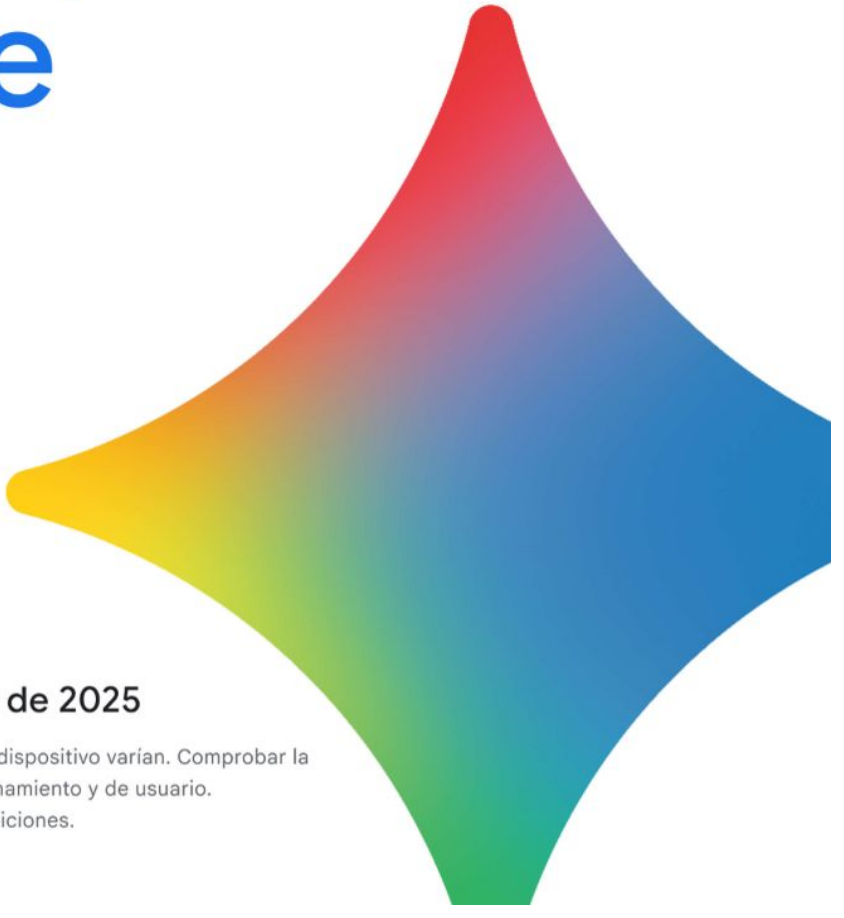
- ✓ **New** : El hilo aun no ha comenzado su ejecución.
- ✓ **Runnable** : El hilo está en posición de ejecutarse.
- ✓ **Blocked** : El hilo está bloqueado esperando tomar el bloqueo de un objeto.
- ✓ **Waiting** : El hilo está esperando indefinidamente hasta que otro hilo realice una acción. Si el hilo ejecuta el método **wait**() de un objeto queda a la espera de que otro hilo ejecute el método **notify**() sobre le mismo objeto.
- ✓ **Timed waiting** : El hilo está esperando durante un tiempo especificado hasta que otro hilo realice una acción.
- ✓ **Terminated** : El hilo ha finalizado.

Google Gemini: Plan Pro a 0€ durante 1 año.

Tu ventaja por ser estudiante

Entra en wlh.es/estudiacongemini

Consigue la oferta



Oferta válida hasta el 9 de diciembre de 2025

Después, 21,99€/mes. 18+. Los resultados/la compatibilidad del dispositivo varían. Comprobar la exactitud de las respuestas. Se aplican restricciones de almacenamiento y de usuario. Se requiere una cuenta de Google. Consulta los términos y condiciones.

4. GESTIÓN DE HILOS

Crear y arrancar un hilo

- ✓ Con subclase **Thread()** → En el constructor de la clase. Hiloclase h = **new** Hiloclase();
- ✓ Implementando la interfaz **Runnable** → En el **start()** o cuando se ejecuta un evento.

```
PrimerHiloR hilo1 = new PrimerHiloR();
New Thread(hilo1).start();
```

Suspensión de un hilo

- ✓ **sleep(long)** → el hilo se queda dormido por un tiempo determinado.
- ✓ **suspend()** → Suspende la ejecución de un hilo hasta que aparece un resume(). Está obsoleto.
- ✓ **wait()** → Mantiene al hilo en espera que aparezca:
  - ✓ **notity()** → Libera a un hilo de los que están bloqueados por el wait().
  - ✓ **notifyAll()** → Libera a todos los hilos bloqueados por el wait().

Parada de un hilo

- ✓ **stop()** → ejecuta la excepción **Deadthread(),** pero ya no se usa.
- ✓ **interrupt()** → pide la interrupción de un hilo. **IsInterrupted()** indica si un hilo esta interrumpido o no (true o false).

Los métodos stop y suspend están obsoletos. Se usa interrupt o se programa de otro modo.

4. GESTIÓN DE HILOS

Para parar un proceso sin el uso de interrupt (), se recomienda usar una variable booleana.

```
run:
En el Hilo
En el Hilo
En el Hilo
En el Hilo
En el Hilo
En el Hilo
En el Hilo
En el Hilo
En el Hilo
En el Hilo
En el Hilo
En el Hilo
En el Hilo
En el Hilo
En el Hilo
En el Hilo
En el Hilo
En el Hilo
En el Hilo
En el Hilo
En el Hilo

------
Hilo Parado
-------

BUILD SUCCESSFUL (total time: 0 seconds)
```

```java
package Hilos;

public class HiloEjemploDead extends Thread{
    private boolean stopHilo = false;
    public void pararHilo(){
       stopHilo = true;
    }

    public void run() {
       while(!stopHilo){
          System.out.println("En el Hilo");
       }
       System.out.println("\n------\nHilo Parado\n-------\n");
    }
    public static void main(String[] args){
       HiloEjemploDead h = new HiloEjemploDead ();
       h.start();
       for(int i=0;i<80000;i++);

       h.pararHilo();
        }
}
```

4. GESTIÓN DE HILOS

Ejemplo de método interrupt() para parada de hilos.

```
package Hilos;

public class InterruptThread extends Thread {
    public void run() {
      try {
         while(!isInterrupted()){
           System.out.println("En el hilo");
            Thread.sleep(10);
         }
      } catch (InterruptedException e){
          System.out.println("Ha ocurrido una excepción");
      }
      System.out.println("Fin del Hilo");
    }

    public void interrumpir(){
        interrupt();
    }

    public static void main(String [] args){
       InterruptThread h = new InterruptThread();
       h.start();
       for (int i=0; i<1000000000; i++);
       h.interrumpir();
    }
}
```

```
run:
En el hilo
Ha ocurrido una excepción
Fin del Hilo
BUILD SUCCESSFUL (total time: 0 seconds)
```

4. GESTIÓN DE HILOS

Ejemplo de método join() que da prioridad al hilo que invoca a este método.

```
package Hilos;

public class JoinThread extends Thread{
    private int n;
    public JoinThread(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(1000);

          }catch (InterruptedException ignore){}
       }
        System.out.println("Fin bucle "+ getName());
    }

   public static void main(String[] args){
     JoinThread h1 = new JoinThread("Hilo1",2);
     JoinThread h2 = new JoinThread("Hilo2",5);
     JoinThread h3 = new JoinThread("Hilo3",7);

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

     try{
        h1.join(); h2.join(); h3.join();
     } catch (InterruptedException e){}
     System.out.println("Final del Progrma");
   }
}
```

```
run:
Hilo2: 1
Hilo3: 1
Hilo1: 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
Final del Progrma
BUILD SUCCESSFUL (t
```

TODOS

```
run:
Hilo1: 1
Hilo3: 1
Hilo2: 1
Hilo3: 2
Hilo1: 2
Hilo2: 2
Hilo3: 3
Hilo2: 3
Fin bucle Hilo1
Final del Progrma
Hilo2: 4
Hilo3: 4
Hilo3: 5
Hilo2: 5
Hilo3: 6
Fin bucle Hilo2
Hilo3: 7
Fin bucle Hilo3
```

HILO1

```
run:
Hilo1: 1
Hilo2: 1
Hilo3: 1
Hilo2: 2
Hilo1: 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
Final del Progrma
```

HILO3

5. GESTIÓN DE PRIORIDADES

- ✓ En Java cada hilo tiene una prioridad de cara a su ejecución.
- ✓ Es un valor entero cuyo valor puede estar entre 1 y 10.
- ✓ Por defecto un hilo hereda la prioridad de su padre, pero se puede modificar con el método **setPriority().**
- ✓ Los valores estándar de prioridad son:

  MIN_PRIORITY → 1

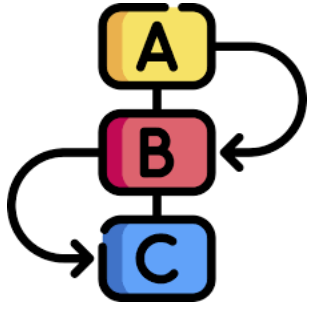
  MAX_PRIORITY → 10

  NORM_PRIORITY → 5

- ✓ El planificador de la JVM es el que decide qué hilo se ejecuta en cada momento, dando más prioridad a la ejecución de los hilos cuyo valor de la prioridad sea más alto.
- ✓ Si dos hilos tienen la misma prioridad el planificador los irá ejecutando de forma alternada (**round-robin**).

5. GESTIÓN DE PRIORIDADES

- ✓ Un hilo puede devolver el control al planificador, para que se ejecuten otros métodos de igual prioridad llamando al método **yield()**.
- ✓ Cuando se programa las prioridades de lo shilos no está garantizado que se cumpla el comportamiento esperado, depende del sistema en el que se ejecute y de otras aplicaciones que se ejecuten a la vez.
- ✓ Para evitar que un hilo acapare todo el tiempo de ejecución (hilo egoista), se suele implementar la técnica de tiempo compartido (**Time slicing**), como por ej. en Windows.
- ✓ No es habitual fijar expresamente las prioridades de los hilos.



5. GESTIÓN DE PRIORIDADES

- ✓ Programa que lanza tres hilos con distintas prioridades.
- ✓ Cada hilo incrementa un contador y al final del programa se muestra el valor que alcanza el contador.

```
public class PrioridadHilos {

    public static void main(String[] args) {
        Hilo hilo1 = new Hilo(nombre:"Hilo1");
        Hilo hilo2 = new Hilo(nombre:"Hilo2");
        Hilo hilo3 = new Hilo(nombre:"Hilo3");
        hilo1.setPriority(newPriority: Thread.NORM_PRIORITY);
        hilo2.setPriority(newPriority: Thread.MAX_PRIORITY);
        hilo3.setPriority(newPriority: Thread.MIN_PRIORITY);
        hilo1.start();
        hilo2.start();
        hilo3.start();
        try {
            Thread.sleep(millis:10000);
        } catch (Exception e) {    }
        hilo1.paraHilo();
        hilo2.paraHilo();
        hilo3.paraHilo();
        System.out.println("Hilo2 con prioridad máxima : " + hilo2.getContador());
        System.out.println("Hilo1 con prioridad normal : " + hilo1.getContador());
        System.out.println("Hilo3 con prioridad mínima : " + hilo3.getContador());
    }
}
```

continúa

5. GESTIÓN DE PRIORIDADES

```
class Hilo extends Thread {

    private int c = 0;
    private boolean stopHilo = false;

    public Hilo(String nombre) {
        super(name: nombre);
    }

    public int getContador() {
        return c;
    }

    public void paraHilo() {
        stopHilo = true;
    }

    @Override
    public void run() {
        while (!stopHilo) {
            try {
                Thread.sleep(millis:2);
            } catch (Exception e) {
            }
            c++;
        }
        System.out.println("Fin del hilo " + this.getName());
        this.interrupt();
    }
}
```

```
--- exec:3.1.0:exec (default-cli) @ U2Multihilo ---
Fin del hilo Hilo2
Fin del hilo Hilo3
Fin del hilo Hilo1
Hilo2 con prioridad máxima : 4298
Hilo1 con prioridad normal : 4311
Hilo3 con prioridad mínima : 4305
-----------------------------------------------------
```

5. GESTIÓN DE PRIORIDADES

- ✓ Programa que lanza cinco hilos con distintas prioridades.
- ✓ No se garantiza que el hilo con la mayor prioridad finalice el primero.

```
public class PrioridadHilos2 extends Thread {
    PrioridadHilos2 (String nombre){
        this.setName(name: nombre);
    }
    @Override
    public void run() {
        System.out.println("Ejecutando hilo " + getName());
        for (int i=1; i<=5; i++)
            System.out.println(getName()+" : "+i);

    }
    public static void main(String[] args) {
        PrioridadHilos2 hilo1 = new PrioridadHilos2(nombre:"Hilo1");
        PrioridadHilos2 hilo2 = new PrioridadHilos2(nombre:"Hilo2");
        PrioridadHilos2 hilo3 = new PrioridadHilos2(nombre:"Hilo3");
        PrioridadHilos2 hilo4 = new PrioridadHilos2(nombre:"Hilo4");
        PrioridadHilos2 hilo5 = new PrioridadHilos2(nombre:"Hilo5");
        hilo1.setPriority(newPriority: MIN_PRIORITY);
        hilo2.setPriority(newPriority: 3);
        hilo3.setPriority(newPriority: NORM_PRIORITY);
        hilo4.setPriority(newPriority: 7);
        hilo5.setPriority(newPriority: MAX_PRIORITY);
        hilo1.start();
        hilo2.start();
        hilo3.start();
        hilo4.start();
        hilo5.start();
    }
}
```

```
Ejecutando hilo Hilo4
Ejecutando hilo Hilo1
Ejecutando hilo Hilo3
Ejecutando hilo Hilo2
Ejecutando hilo Hilo5
Hilo5 : 1
Hilo5 : 2
Hilo2 : 1
Hilo3 : 1
Hilo1 : 1
Hilo1 : 2
Hilo1 : 3
Hilo4 : 1
Hilo5 : 3
Hilo5 : 4
Hilo5 : 5
Hilo2 : 2
Hilo3 : 2
Hilo1 : 4
Hilo1 : 5
Hilo4 : 2
Hilo2 : 3
Hilo3 : 3
Hilo3 : 4
Hilo4 : 3
Hilo2 : 4
Hilo3 : 5
Hilo4 : 4
Hilo2 : 5
Hilo4 : 5
```

6. COMUNICACIÓN Y SINCRONIZACIÓN DE HILOS

- ✓ En múltiples situaciones los hilos necesitan comunicarse para realizar su tarea.
- ✓ El método habitual de comunicación entre hilos es la compartición de un objeto.

Ejemplo **compartición de contador**:

- ✓ Dos hilos comparten un contador.
- ✓ El Hilo1 lo incrementa y el Hilo2 lo decrementa.
- ✓ El resultado esperado debería ser:

  Hilo_1 : su contador vale 100

  Hilo_2 : su contador vale 100

- ✓ Pero si lo ejecutamos varias veces vemos que no es así.

```
public class CompartirContador extends Thread{
    public static void main(String[] args) {
        Contador cont = new Contador(cont: 100);
        Hilo1 h1 = new Hilo1 (nombre:"Hilo_1", cont);
        Hilo2 h2 = new Hilo2 (nombre:"Hilo_2", cont);
        h1.start();
        h2.start();
        }
    }
class Contador {
    private int cont = 0;
    Contador (int cont) {
        this.cont = cont;
    }
    public void incrementa() {
        cont++;
    }
    public void decrementa() {
        cont--;
    }

    public int valor() {
        return cont;
    }
}
```

continúa

6. COMUNICACIÓN Y SINCRONIZACIÓN DE HILOS

Ejemplo **compartición de contador**:

- ✓ Al final el contador vale lo mismo en ambos casos pero no podemos predecir su valor ya que el acceso al objeto no está sincronizado.
- ✓ No se garantiza la ejecución de los métodos incrementa y decrementa en todas las iteraciones

```
--- exec:3.1.0:exec (default-cli)
Hilo_1 : su contador vale 95
Hilo_2 : su contador vale 95
---------------------------------
BUILD SUCCESS
```

```
--- exec:3.1.0:exec (default-cli) @
Hilo_2 : su contador vale 96
Hilo_1 : su contador vale 96
---------------------------------
BUILD SUCCESS
```

```java
class Hilo1 extends Thread {
    private Contador contador;
    public Hilo1(String nombre, Contador cont) {
        setName(name: nombre);
        contador = cont;
    }
    @Override
    public void run() {
        for (int i = 0; i < 100; i++) {
            contador.incrementa();
            try {
                sleep(millis:10);
            } catch (InterruptedException e) { }
        }
        System.out.println(getName() + " : su contador vale " + contador.valor());
    }
}

class Hilo2 extends Thread {
    private Contador contador;
    public Hilo2(String nombre, Contador cont) {
        setName(name: nombre);
        contador = cont;
    }
    @Override
    public void run() {
        for (int i = 0; i < 100; i++) {
            contador.decrementa();
            try {
                sleep(millis:10);
            } catch (InterruptedException e) { }
        }
        System.out.println(getName() + " : su contador vale " + contador.valor());
    }
}
```

6. COMUNICACIÓN Y SINCRONIZACIÓN DE HILOS

Métodos de sincronización

- ✓ **Bloques sincronizados o región crítica**: Un bloque sincronizado es un bloque de código que solo puede ser ejecutado por un hilo a la vez. Para sincronizar un bloque de código, se utiliza la palabra clave ***synchronized*** junto con un objeto de sincronización.
- ✓ **Métodos sincronizados**: Un método sincronizado es un método que solo puede ser ejecutado por un hilo a la vez. Se declara como ***synchronized***.
- ✓ **Bloqueo de hilos**: también podemos bloquear la ejecución de un hilo a la espera de que un objeto compartido esté disponible. Con los métodos notify y wait (dentro de un bloque sincronizado).
- ✓ **Modelo productor-consumidor**: Es un caso típico de sincronización en el que un hilo produce datos que son consumidos por otro hilo. Para que no haya un problema si productor y consumidor no llevan el mismo ritmo es necesario sincronizar sus actividades, mediante los métodos anteriores.

6. COMUNICACIÓN Y SINCRONIZACIÓN DE HILOS

Bloques sincronizados o región crítica

- ✓ Para evitar que haya inconsistencia en la compartición de un objeto, como en el ejemplo anterior, es necesario garantizar que las operaciones de incremento y decremento del contador se realizan de **manera atómica**. Esto es, que siempre se complete una operación antes de realizar la siguiente.
- ✓ Para ello se usa la técnica de bloques sincronizados que consiste en declarar como sincronizado una parte del código mediante la palabra clave **synchronized**.

```
Synchronized (objeto) {
        Sentencias críticas a ejecutar de forma atómica
}
```

- ✓ Cada vez que un hilo intenta acceder a un bloque sincronizado pregunta al objeto si hay algún hilo que lo tenga bloqueado y en caso afirmativo espera a que se libere.

6. COMUNICACIÓN Y SINCRONIZACIÓN DE HILOS

Bloques sincronizados o región crítica

Ejemplo **compartición de contador con synchronized**

✓ Modificamos el ejemplo anterior añadiendo la definición de un bloque sincronizado dentro del método run() de cada uno de los hilos.

```
class HiloA extends Thread {
    private MiContador contador;
    public HiloA(String nombre, MiContador cont) {
        setName(name: nombre);
        contador = cont;
    }
    @Override
    public void run() {
        synchronized (contador)  {
            for (int i = 0; i < 100; i++) {
                contador.incrementa();
                try {
                    sleep(millis:10);
                } catch (InterruptedException e) { }
            }
            System.out.println(getName() + " : su contador vale " + contador.valor());
        }
    }
}
```

Bloque sincronizado

Ahora la salida siempre es la misma:

```
--- exec:3.1.0:exec (default-c
Hilo_1 : su contador vale 200
Hilo_2 : su contador vale 100
------------------------------
BUILD SUCCESS
```

6. COMUNICACIÓN Y SINCRONIZACIÓN DE HILOS

Métodos sincronizados → Método de exclusión mutua

✓ Sincronizar métodos implica que no se puede invocar dos métodos del mismo objeto al mismo tiempo.

CASO DE LA CUENTA COMPARTIDA POR DOS PERSONAS

```
package Hilos;

public class CuentaHilosSincronizados {

    public static 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){
synchronized
          if(getSaldo() >= cant){
            System.out.println(nom+": va a retirar saldo, el actual es: " + getSaldo());
            try{
               Thread.sleep(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());
          }
       }//retirar
    }//Cuenta
```

6. COMUNICACIÓN Y SINCRONIZACIÓN DE HILOS

Métodos sincronizados → Método de exclusión mutua

✓ Sincronizar métodos implica que no se puede invocar dos métodos de mismo objeto al mismo tiempo.

CASO DE LA CUENTA COMPARTIDA POR DOS PERSONAS

```
    public static 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(10, getName());
          }
        }//run
    }//SacarDinero

    public static void main(String [] args){
        Cuenta c = new Cuenta(40);
        SacarDinero h1 = new SacarDinero("Ana", c);
        SacarDinero h2 = new SacarDinero("Juan", c);
        h1.start();
        h2.start();
    }
}
```

Sin synchronized

```
run:
Juan: va a retirar saldo, el actual es: 40
Ana: va a retirar saldo, el actual es: 40
        Juan retira =>10;  ACTUAL(20)
        Ana retira =>10;  ACTUAL(20)
Juan: va a retirar saldo, el actual es: 20
Ana: va a retirar saldo, el actual es: 20
        Ana retira =>10;  ACTUAL(0)
        Juan retira =>10;  ACTUAL(0)
Juan No puede retirar dinero, NO HAY SALDO(0)
Juan No puede retirar dinero, NO HAY SALDO(0)
Ana No puede retirar dinero, NO HAY SALDO(0)
Ana No puede retirar dinero, NO HAY SALDO(0)
BUILD SUCCESSFUL (total time: 1 second)
```

Con synchronized

```
run:
Ana: va a retirar saldo, el actual es: 40
        Ana retira =>10;  ACTUAL(30)
Juan: va a retirar saldo, el actual es: 30
        Juan retira =>10;  ACTUAL(20)
Ana: va a retirar saldo, el actual es: 20
        Ana retira =>10;  ACTUAL(10)
Juan: va a retirar saldo, el actual es: 10
        Juan retira =>10;  ACTUAL(0)
Ana No puede retirar dinero, NO HAY SALDO(0)
Ana 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)
BUILD SUCCESSFUL (total time: 2 seconds)
```

6. COMUNICACIÓN Y SINCRONIZACIÓN DE HILOS

Bloqueo de hilos → wait() y notify()

```
package Hilos;

public class BloqueoHilos {
    public static class ObjetoCompartido{
        public void PintaCadena(String s){
            System.out.println(s);
        }
    }

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

```
public void run(){
 synchronized(objeto){
   for(int j = 0; j < 10; j++){
      objeto.PintaCadena(cad);
      objeto.notify();
      try{
         objeto.wait();
      }catch(InterruptedException e){ e.printStackTrace();}
   }//for
   objeto.notify();
  }
  System.out.print("\n"+cad+" finalizado");
 }//run
```

```
run:
A
A
A
A
A
A
A
A
A
A
A
B
B
B
B
B
B
B
B
B
B
B
BUILD
```

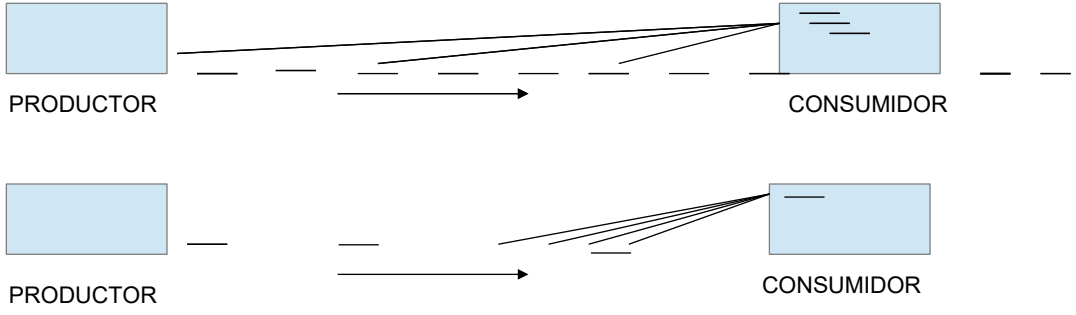
```
run:
A
B
A
B
A
B
A
B
A
B
A
B
B
A
B

A finalizado
B finalizadoBUILD
```

Los métodos **notify** y **wait** sólo se pueden llamar desde dentro de un bloque o sentencia sincronizado.

6. COMUNICACIÓN Y SINCRONIZACIÓN DE HILOS

Bloqueo de hilos → wait() y notify()

MODELO DE PRODUCTOR CONSUMIDOR.



6. COMUNICACIÓN Y SINCRONIZACIÓN DE HILOS

Bloqueo de hilos → wait() y notify()

MODELO DE PRODUCTOR CONSUMIDOR.

```
public class ProductorConsumidor {

    public static class Cola{
      private int numero;
      private boolean disponible = false;

      public int get(){
         if(disponible){
           disponible = false;
           return numero;
         }
         return -1;
      }
      public void put(int valor){
         numero = valor;
         disponible = true;
      }
    }

    public static class Productor extends Thread {
       private Cola cola;
       private int n;
       public Productor(Cola c, int n){
         cola = c;
         this.n = n;
       }
       public void run(){
          for(int i=0;i<5;i++){
            cola.put(i);
            System.out.println(i + "->Productor : " + n
                    + ", produce: " + i);
            try{
               sleep(100);
            }catch(InterruptedException e){}
          }
       }
```

```
    public static class Consumidor extends Thread {
       private Cola cola;
       private int n;
       public Consumidor(Cola c, int n){
         cola = c;
         this.n = n;
       }
       public void run(){
          int valor = 0;
         for(int i=0;i<5;i++){
           valor=cola.get();
           System.out.println(i + "->Consumidor : " + n
                   + ", consume: " + valor);
         }
       }
    }

    public static void main (String [] args){
       Cola cola = new Cola();
       Productor p = new Productor(cola,1);
       Consumidor c = new Consumidor(cola,1);
       p.start();
       c.start();
    }
}
```

```
run:
0->Consumidor : 1, consume: 0
1->Consumidor : 1, consume: -1
0->Productor : 1, produce: 0
2->Consumidor : 1, consume: -1
3->Consumidor : 1, consume: -1
4->Consumidor : 1, consume: -1
1->Productor : 1, produce: 1
2->Productor : 1, produce: 2
3->Productor : 1, produce: 3
4->Productor : 1, produce: 4
BUILD SUCCESSFUL (total time: 0 seconds)
```

6. COMUNICACIÓN Y SINCRONIZACIÓN DE HILOS

Bloqueo de hilos → wait() y notify()

MODELO DE PRODUCTOR CONSUMIDOR.

```
public class ProductorConsumidor {

    public static class Cola{
      private int numero;
      private boolean disponible = false;

      public int get(){
         if(disponible){
           disponible = false;
           return numero;
         }
         return -1;
      }
      public void put(int valor){
         numero = valor;
         disponible = true;
      }
    }

    public static class Productor extends Thread {
       private Cola cola;
       private int n;
       public Productor(Cola c, int n){
         cola = c;
         this.n = n;
       }
       public void run(){
          for(int i=0;i<5;i++){
            cola.put(i);
            System.out.println(i + "->Productor : " + n
                    + ", produce: " + i);
            try{
               sleep(100);
            }catch(InterruptedException e){}
          }
       }
```

```
public synchronized int get(){
   while(!disponible){
      try{
        wait();
      }catch(InterruptedException e){}
   disponible = false;
   notify();
   return numero;
}
```

```
public synchronized void put(int valor){
   while (disponible){
      try{
        wait();
      }catch(InterruptedException e){}
   numero = valor;
   disponible = true;
   notify();
}
```

```
    public static class Consumidor extends Thread {
       private Cola cola;
       private int n;
       public Consumidor(Cola c, int n){
         cola = c;
         this.n = n;
       }
       public void run(){
          int valor = 0;
          for(int i=0;i<5;i++){
            valor=cola.get();
            System.out.println(i + "->Consumidor : " + n
                    + ", consume: " + valor);
          }
       }
    }

    public static void main (String [] args){
       Cola cola = new Cola();
       Productor p = new Productor(cola,1);
       Consumidor c = new Consumidor(cola,1);
       p.start();
       c.start();
    }
}
```

```
run:
0->Productor : 1, produce: 0
0->Consumidor : 1, consume: 0
1->Productor : 1, produce: 1
1->Consumidor : 1, consume: 1
2->Productor : 1, produce: 2
2->Consumidor : 1, consume: 2
3->Consumidor : 1, consume: 3
3->Productor : 1, produce: 3
4->Productor : 1, produce: 4
4->Consumidor : 1, consume: 4
BUILD SUCCESSFUL (total time: 0 seconds)
```

¡MUCHAS GRACIAS!

THE CORE
CENTRO DE FORMACIÓN PROFESIONAL