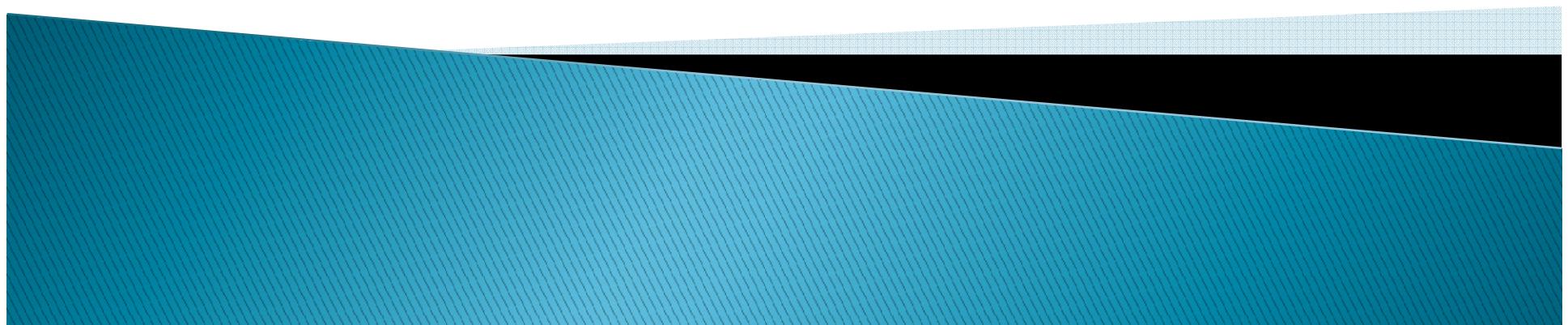


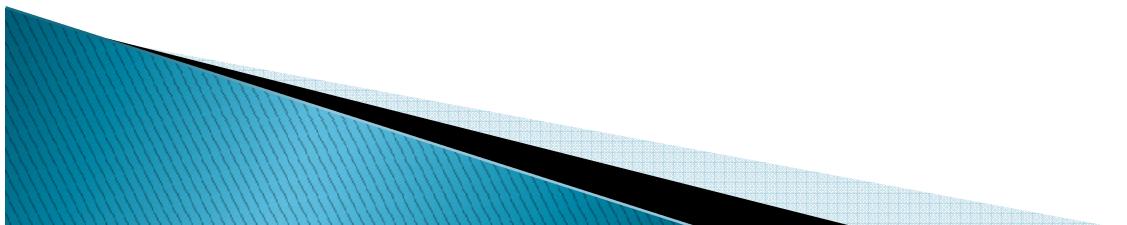
TEMA 2: PROGRAMACIÓN DE HILOS

Programación de Servicios y Procesos
José Manuel García



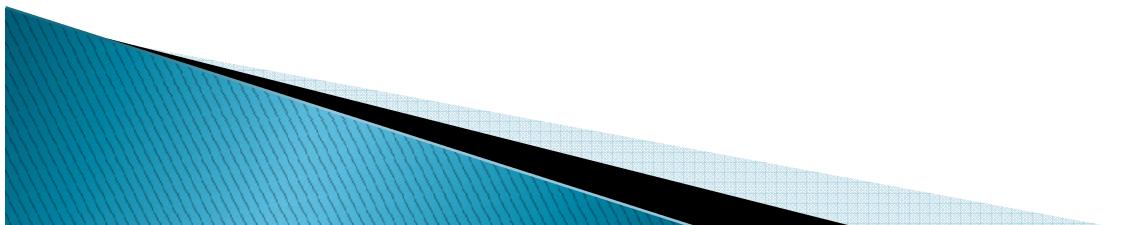
ÍNDICE

- ▶ Concepto Hilo
- ▶ Multitarea
- ▶ Recursos compartidos por Hilos
- ▶ Estados de un Hilo
- ▶ Gestión de Hilos
- ▶ Planificación de Hilos
- ▶ Sincronización de Hilos
- ▶ Mecanismos de sincronización

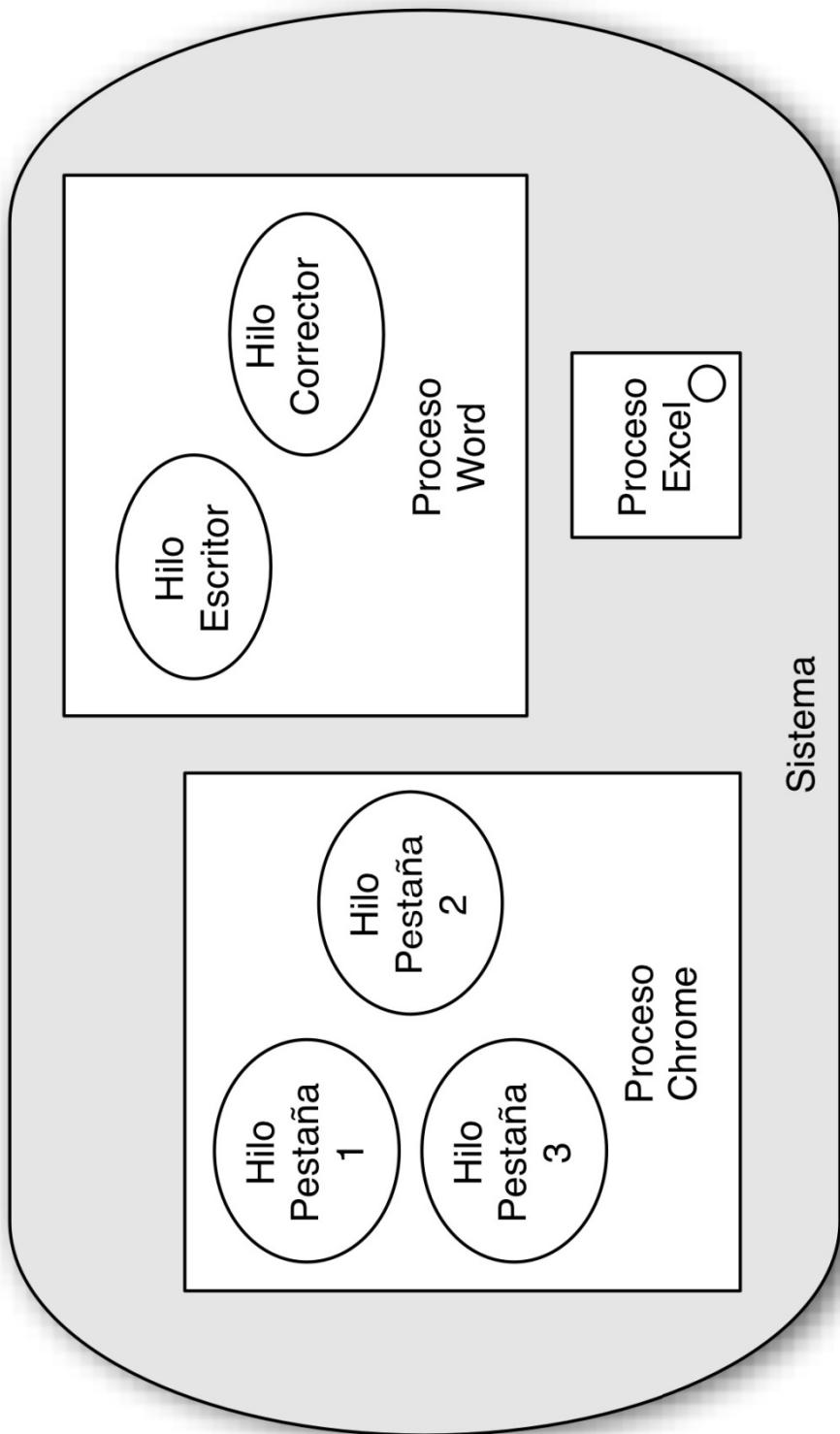


Hilo

- ▶ Hilo (thread):
 - Unidad básica de utilización de la CPU, y más concretamente de un *core* del procesador.
 - Secuencia de código que está en ejecución, pero dentro del contexto de un proceso.
- ▶ Diferencia con procesos:
 - Los hilos se ejecutan dentro del contexto de un proceso. Dependen de un proceso para ejecutarse.
 - Los procesos son independientes y tienen espacios de memoria diferentes.
 - Dentro de un mismo proceso pueden coexistir varios hilos ejecutándose que compartirán la memoria de dicho proceso.



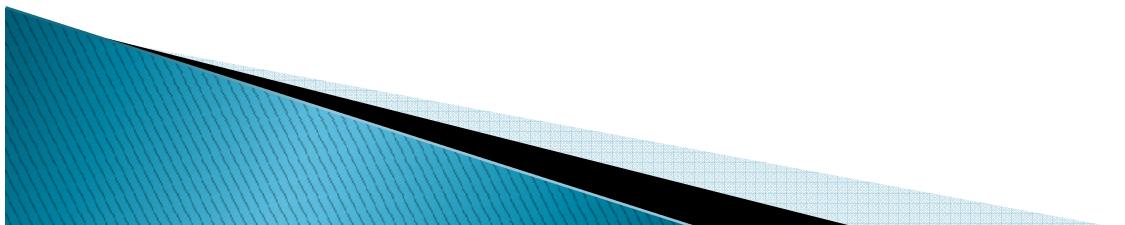
Hilo



Sistema

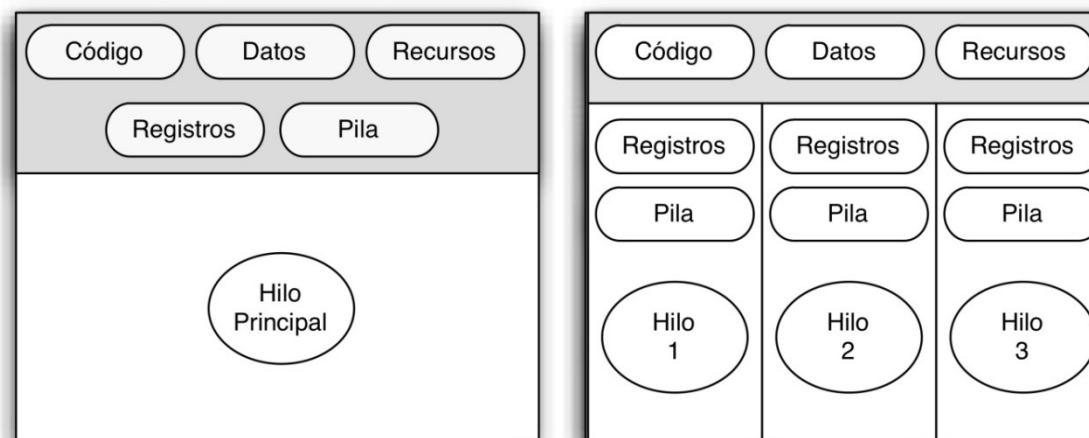
Multitarea

- ▶ **Multitarea:** ejecución simultanea de varios hilos:
 - **Capacidad de respuesta.** Los hilos permiten a los procesos continuar atendiendo peticiones del usuario aunque alguna de las tareas (hilo) que esté realizando el programa sea muy larga.
 - **Compartición de recursos.** Por defecto, los *threads* comparten la memoria y todos los recursos del proceso al que pertenecen.
 - **La creación de nuevos hilos no supone ninguna reserva adicional de memoria** por parte del sistema operativo.
 - **Paralelismo real.** La utilización de *threads* permite aprovechar la existencia de más de un núcleo en el sistema en arquitecturas *multicore*.



Recursos compartidos por Hilos

- ▶ Los procesos mantienen su propio espacio de direcciones y recursos de ejecución mientras que los hilos dependen del proceso.
 - Comparten con otros hilos la sección de código, datos y otros recursos.
 - Cada hilo tiene su propio contador de programa, conjunto de registros de la CPU y pila para indicar por dónde se está ejecutando.

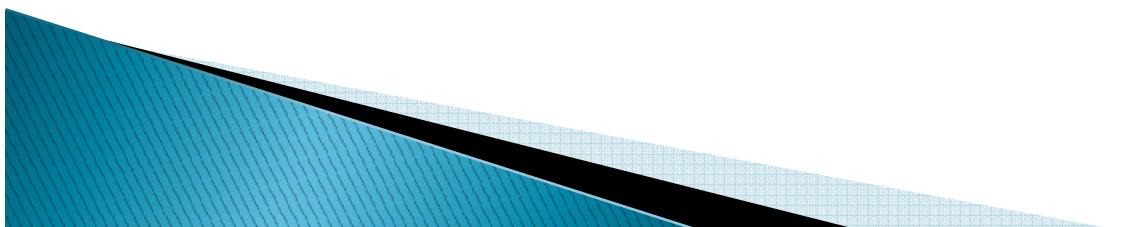


Proceso con un único thread

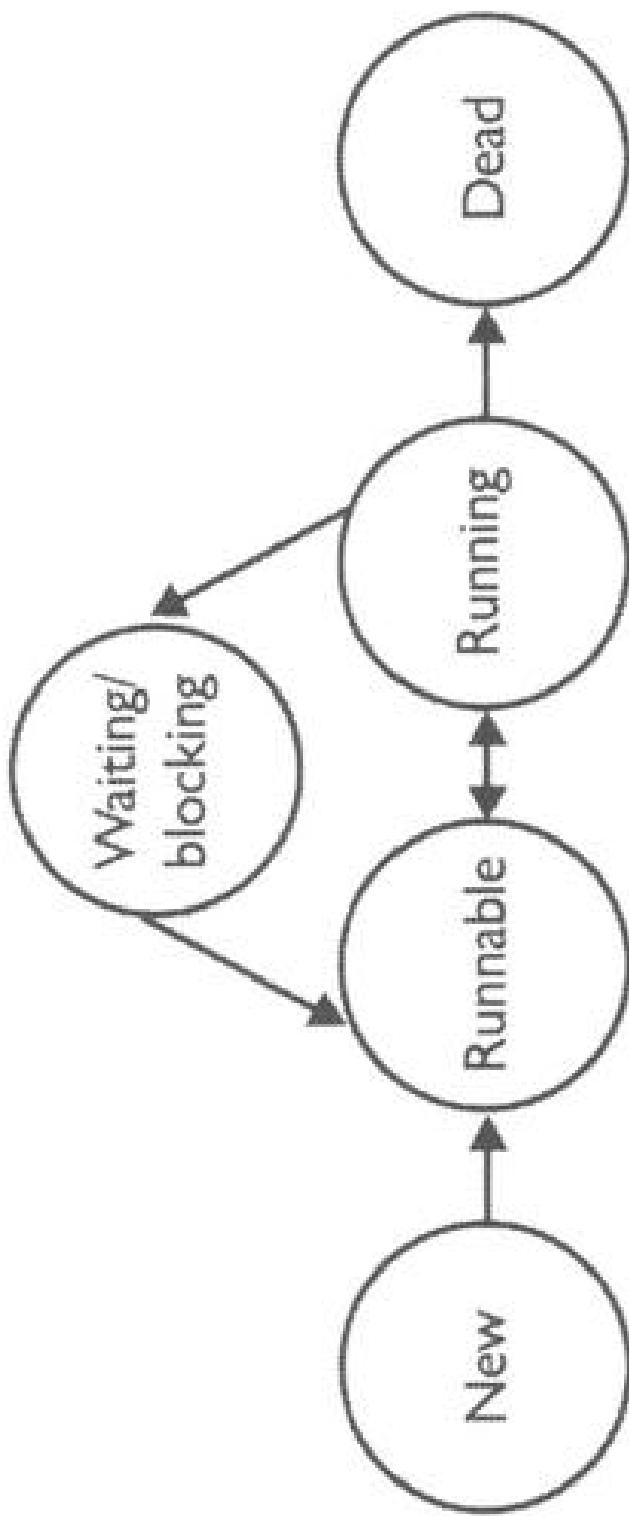
Proceso con varios threads

Estados de un hilo

- ▶ Los hilos pueden cambiar de estado a lo largo de su ejecución.
- ▶ Se definen los siguientes estados:
 - **Nuevo**: el hilo está preparado para su ejecución pero todavía no se ha realizado la llamada correspondiente en la ejecución del código del programa.
 - **Listo**: el proceso no se encuentra en ejecución aunque está preparado para hacerlo. El sistema operativo no le ha asignado todavía un procesador para ejecutarse.
 - **Runnable**: el hilo está preparado para ejecutarse y puede estar ejecutándose.
 - **Bloqueado**: el hilo está bloqueado por diversos motivos esperando que el suceso suceda para volver al estado *Runnable*.
 - **Terminado**: el hilo ha finalizado su ejecución.

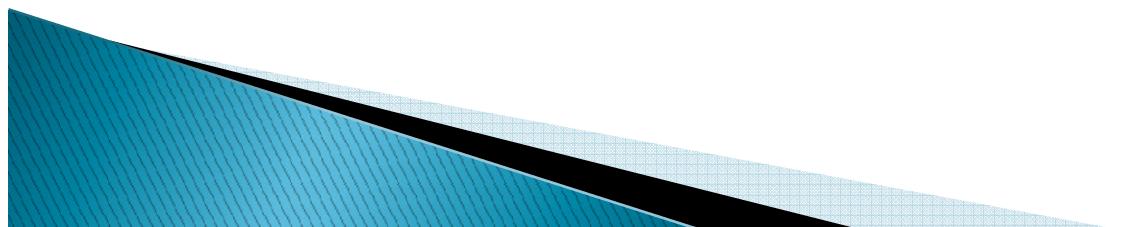


Estados de un hilo



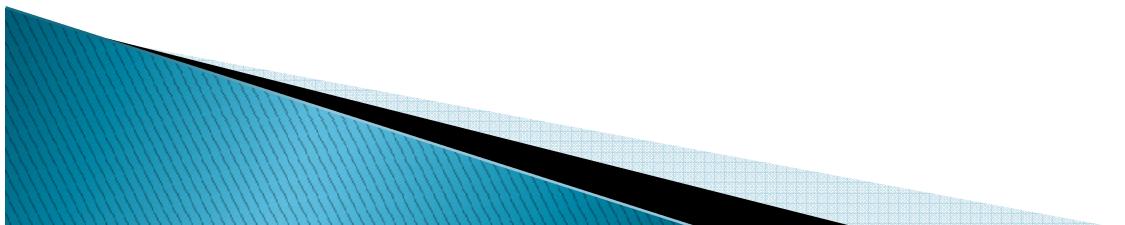
Gestión de Hilos

- ▶ Operaciones básicas:
 - Creación y arranque de hilos. Cualquier programa a ejecutarse es un proceso que tiene un hilo de ejecución principal. Este hilo puede a su vez crear nuevos hilos que ejecutarán código diferente o tareas, es decir el camino de ejecución no tiene por qué ser el mismo.
 - Espera de hilos. Como varios hilos comparten el mismo procesador, si alguno no tiene trabajo que hacer, es bueno suspender su ejecución para que haya un mayor tiempo de procesador disponible.



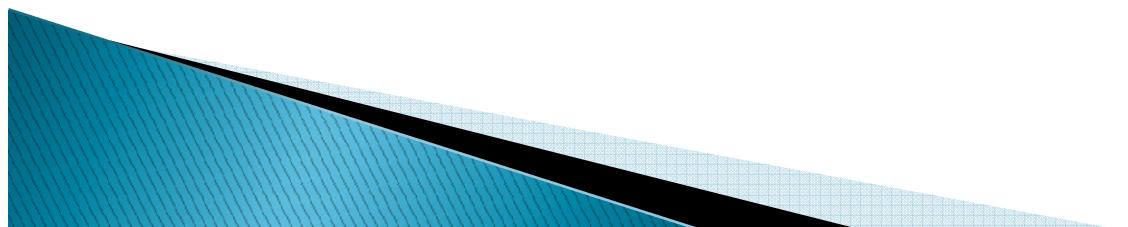
Creación y Arranque de hilos

- ▶ Creación de hilos en Java:
 - Implementando la interfaz ***Runnable***. La interfaz *Runnable* proporciona la capacidad de añadir la funcionalidad de un hilo a una clase simplemente implementando dicha interfaz. La interfaz *Runnable* debería ser utilizada si la clase solamente va a utilizar la funcionalidad *run* de los hilos.
 - Extendiendo de la clase ***Thread*** mediante la creación de una subclase. La clase *Thread* es responsable de producir hilos funcionales para otras clases e implementa la interfaz *Runnable*
- ▶ El método *run()* implementa la operación *create* contenido el código a ejecutar por el hilo. Dicho método contendrá el hilo de ejecución.
- ▶ La clase *Thread* define también el método *start()* para implementar la operación *create*. Este método es que comienza la ejecución del hilo de la clase correspondiente.



Creación y Arranque de Hilos.

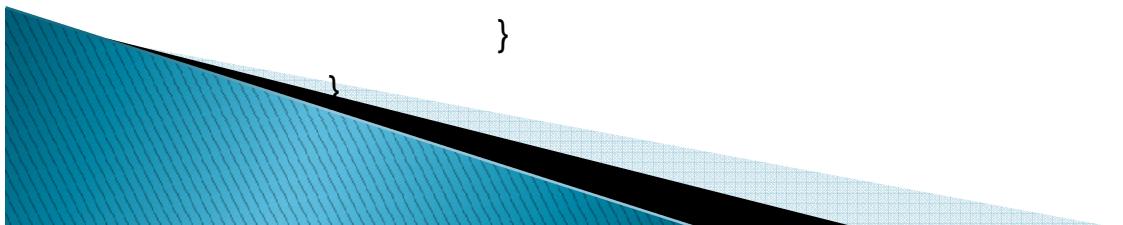
- ▶ Existen en Java, por tanto dos formas:
 - Creando una clase que herede de la clase Thread y sobrecargando el método run().
 - Implementando la interface Runnable, y declarando el método run();



Creación y Arranque de hilos

- ▶ Implementando la interfaz *Runnable*:

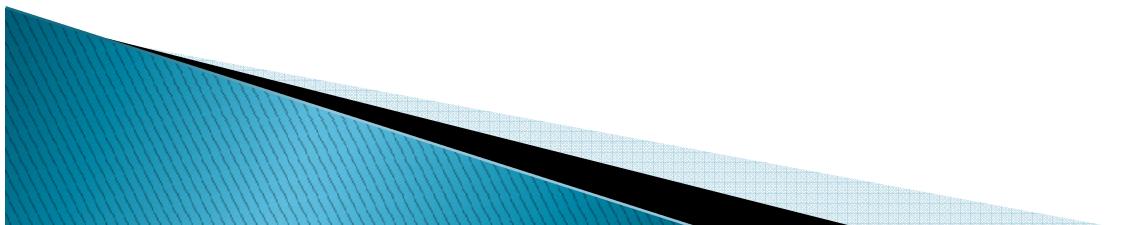
```
class HelloThread implements Runnable {  
    Thread t;  
    HelloThread () {  
        t = new Thread(this, "Nuevo Thread");  
        System.out.println("Creado hilo: " + t);  
        t.start(); // Arranca el nuevo hilo de ejecución. Ejecuta run  
    }  
  
    public void run() {  
        System.out.println("Hola desde el hilo creado!");  
        System.out.println("Hilo finalizando.");  
    }  
}  
  
class RunThread {  
    public static void main(String args[]) {  
        new HelloThread(); // Crea un nuevo hilo de ejecución  
        System.out.println("Hola desde el hilo principal!");  
        System.out.println("Proceso acabando.");  
    }  
}
```



Creación y Arranque de hilos

- ▶ Extendiendo la clase *Thread*

```
class HelloThread extends Thread {  
  
    public void run() {  
        System.out.println("Hola desde el hilo creado!");  
    }  
}  
  
public class RunThread {  
    public static void main(String args[]) {  
  
        new HelloThread().start(); // Crea y arranca un nuevo hilo de ejecución  
        System.out.println("Hola desde el hilo principal!");  
        System.out.println("Proceso acabando.");  
    }  
}
```



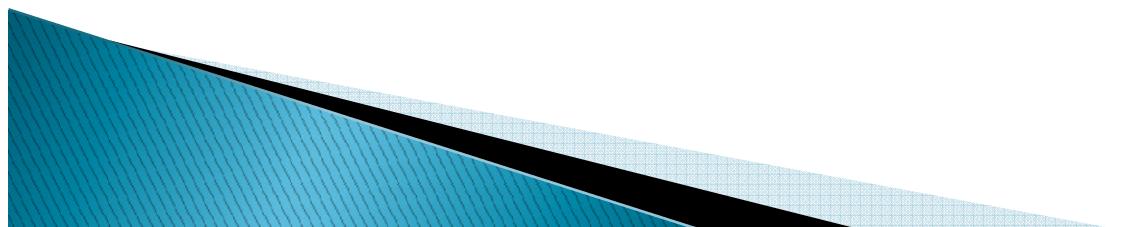
Hilos: Runnable vs Thread

```
public class Hilosimple extends Thread  
{  
    public void run{  
        for (int i=0; i<5; i++)  
            System.out.println("En  
el hilo...");  
    }  
    public class usahilo{  
        public static void main (String[] args)  
{  
            Hilosimple hs = new  
Hilosimple();  
            hs.start();  
            for (int i=0; i<5;i++)  
  
                System.out.println("Fuera del  
hilo..");  
        }  
    }
```

```
public class Hilosimple2 implements  
Runnable{  
    public void run(){  
        for (int i=0; i<5; i++)  
  
            System.out.println("En el  
hilo...");  
    }  
    public class usahilo2{  
        public static void main (String[] args)  
{  
            Hilosimple2 hs = new  
Hilosimple2();  
            Thread t=new Thread(hs);  
            t.start();  
            for (int i=0; i<5;i++)  
  
                System.out.println("Fuera del  
hilo..");  
        }  
    }
```

Creación y Arranque de hilos

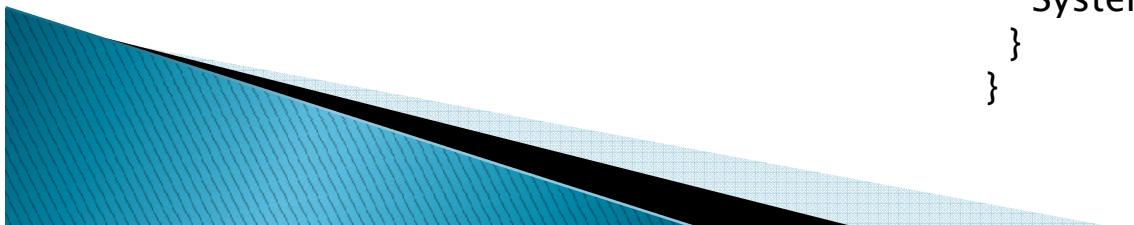
- ▶ De las dos alternativas, ¿cuál utilizar? Depende de la necesidad:
 - La utilización de la interfaz *Runnable* es más general, ya que el objeto puede ser una subclase de una clase distinta de *Thread*
 - La utilización de la interfaz *Runnable* no tiene ninguna otra funcionalidad además de *run()* que la incluida por el programador.
 - La extensión de la clase *Thread* es más fácil de utilizar, ya que está definida una serie de métodos útiles para la administración de hilos,
 - La extensión de la clase *Thread* está limitada porque los clases creadas como hilos deben ser descendientes únicamente de dicha clase.



Multihilos. Runnable

```
class NewThread implements Runnable {  
    String name; // nombre del hilo  
    Thread t;  
    NewThread(String threadname) {  
        name = threadname;  
        t = new Thread(this, name);  
        System.out.println("Nuevo hilo: " + t);  
        t.start(); // Comienza el hilo  
    }  
    // Este es el punto de entrada del hilo.  
    public void run() {  
        try {  
            for(int i = 5; i > 0; i--) {  
                System.out.println("Hilo " + name + ": " +  
i);  
                Thread.sleep(1000);  
            }  
        }  
    }  
}
```

```
catch (InterruptedException e) {  
    System.out.println(name + " InterrupciÃ³n del  
hilo hijo " + name);  
}  
System.out.println("Sale del hilo hijo " + name);  
}  
}  
class MultiThreadDemo {  
    public static void main(String args[]) {  
        new NewThread("Uno"); // comienzan los hilos  
        new NewThread("Dos");  
        new NewThread("Tres");  
        try {  
            // espera un tiempo para que terminen los  
            // otros hilos  
            Thread.sleep(10000);  
        } catch (InterruptedException e) {  
            System.out.println("InterrupciÃ³n del hilo  
principal");  
        }  
        System.out.println("Sale del hilo principal.");  
    }  
}
```



Multihilos. Thread

Compilación y ejecución
desde consola

```
C:\>Javac usahilo.java  
C:\>Java usahilo
```

```
public class Hilo1 extends Thread{  
    private int c; //contador de cada hilo  
    private int hilo;  
    public Hilo1(int hilo){ //constructor  
        this.hilo=hilo;  
        System.out.println("Creando Hilo: " + hilo);}  
    public void run() {//metodo run  
        c=0;  
        while (c<=5) {  
            System.out.println("Hilo:" + hilo + " C = " + c);  
            c++;  
        } }  
    public class usahilo {  
        public static void main (String[] args) {  
            Hilo1 h=null;  
            for (int i=0;i<3;i++) { //creando hilos  
                h=new Hilo1(i+1);  
                h.start(); }  
            System.out.println("3 Hilos Creados...");  
        } }//clase
```

```
import java.applet.*; import java.awt.*; import java.text.SimpleDateFormat; import java.util.*;
public class relojmultihilo extends Applet implements Runnable{
    private Thread hilo=null; //hilo
    private Font fuente; //tipo de letra para la hora
    private String horaActual="";
    public void init(){
        fuente=new Font("Verdana", Font.BOLD,26);      }
    public void start(){
        if (hilo==null){
            hilo=new Thread(this);
            hilo.start();          }      }
    public void run(){
        Thread hiloActual=Thread.currentThread();
        while (hilo==hiloActual){
            SimpleDateFormat sdf=new SimpleDateFormat("HH:mm:ss");
            Calendar cal=Calendar.getInstance();
            horaActual=sdf.format(cal.getTime());
            repaint(); //actualiza el contenido del applet
            try{
                Thread.sleep(1000);
            } catch (InterruptedException e) {}}
        }
    public void paint (Graphics g){
        g.clearRect(1, 1, getSize().width, getSize().height);
        setBackground(Color.yellow); //color de fondo
        g.setFont(fuente); //fuente
        g.drawString(horaActual,20,50); //muestra la hora      }
    public void stop(){
        hilo=null;      } }
```

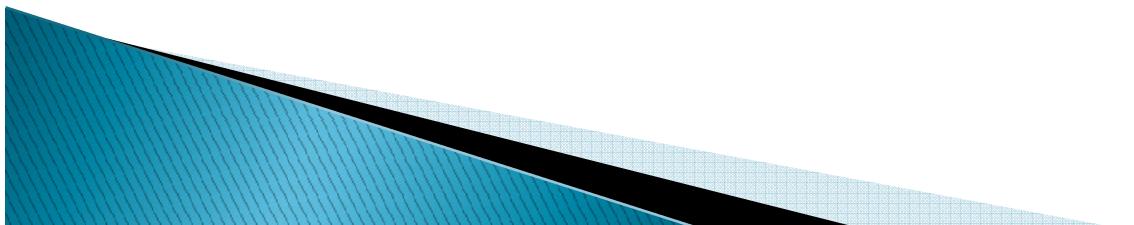
Reloj

Reloj.html Arranque Applet

```
<html>
  <applet code="relojmultihilo.class" width="200" height="100"></applet>
</html>
```

Lanza Applet

```
C:>javac relojmultihilo.java
C:>appletviewer reloj.html
```



Applet en Java.

- ▶ Import java.applet.*;
- ▶ Public class Applet1 extends Applet{
- ▶ Dar Funcionalidad:
 - Función **init**: se llama al cargar el applet. Inicializa, establece propiedades. Se ejecuta solo una vez
 - El método **paint** nos suministra el contexto gráfico de *g*
 - Mostrar objeto: llamar desde *g* a **drawString**
 - *El navegador controla el applet y llama al método paint cada vez que lo necesita*
- ▶ Ejecución
 - Método **init**. Solo una vez
 - *El navegador llama a paint . (Tantas veces como lo necesite)*
 - A continuación se llama al método **start**
 - Al dejar la página se llama al método **stop**
 - Al salir del navegador se llama al método **destroy**.

Espera de hilos

- ▶ Operaciones:
 - **Join**: La ejecución del hilo puede suspenderse esperando hasta que el hilo correspondiente por el que espera finalice su ejecución.
 - **Sleep**: duerme un hilo por un período especificado
- ▶ Ambas operaciones de espera pueden ser interrumpidas, si otro hilo interrumpe al hilo actual mientras está suspendido por dichas llamadas.
 - Una interrupción es una indicación a un hilo que debería dejar de hacer lo que esté haciendo para hacer otra cosa.
 - Un hilo envía una interrupción mediante la invocación del método **interrupt()** en el objeto del hilo que se quiere interrumpir.
- ▶ **isAlive()**: comprueba si el hilo no ha finalizado su ejecución antes de trabajar con él.



Interrupción

- ▶ Indica a un hilo que debe dejar de hacer lo que esté haciendo para hacer otra cosa.
 - Objeto hilo.interrupt() y se recibe excepción InterruptedException

```
public void run() {  
    for (int i = 0; i < NDatos; i++) {  
        try {  
            System.out.println("Esperando a recibir dato!");  
            Thread.sleep(500);  
        } catch (InterruptedException e) {  
            System.out.println("Hilo interrumpido.");  
            return;  
        }  
        // Gestionar dato i  
    }  
    System.out.println("Hilo finalizando correctamente.");  
}
```

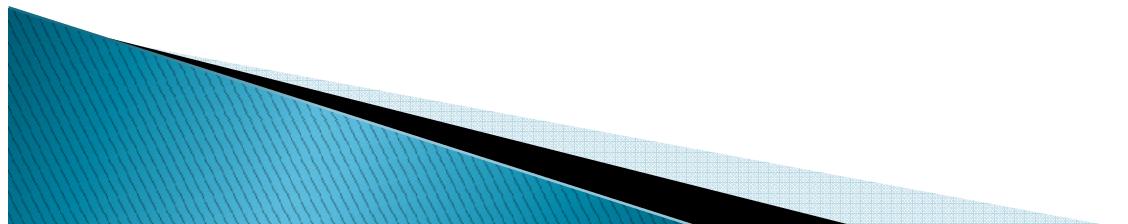


Clase Thread

Método	Tipo Retorno	Descripción
start()	void	Implementa la operación Create. Comienza la ejecución del hilo de la clase correspondiente. Llama al método run()
run()	void	Si el hilo se construyó implementando la interfaz Runnable, entonces se ejecuta el método run() de ese objeto. En caso contrario no hace nada.
currentThread()	static Thread	Devuelve una referencia al objeto hilo que se está ejecutando actualmente
join()	void	Implementa la operación join para hilos. Suspende la ejecución del hilo hasta que finalice otro
sleep(long milis)	static void	El hilo que se está ejecutando se suspende durante el número de milisegundos especificado
interrupt()	void	Interrumpe el hilo del objeto
interrupted()	static boolean	Comprueba si el hilo ha sido interrumpido
isAlive()	boolean	Devuelve true en caso de que el hilo esté vivo, es decir, no haya terminado el método run() en su ejecución

Práctica. Multihilo

- ▶ Diseña un programa que cree 4 hilos de forma que:
 - uno sume los primeros 10 números naturales
 - otro los multiplique
 - otro devuelva la suma de los 10 primeros números pares
 - otro la suma de los 10 primeros números impares.
- ▶ El programa principal deberá devolver la suma de los cuatro resultados.
- ▶ Implementar una clase con 4 propiedades y escribir en ellas cada resultado.

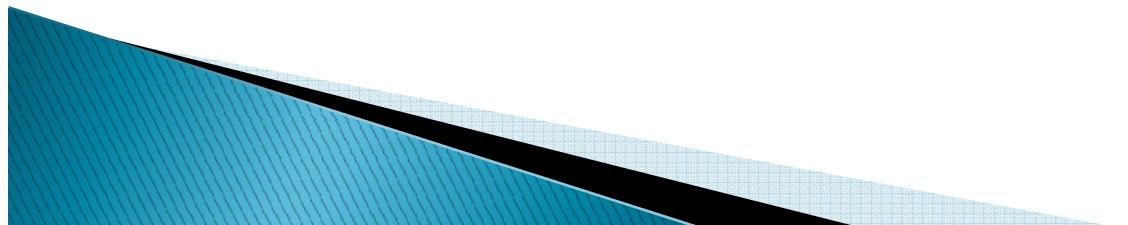


Práctica. Multihilo

- ▶ Escribe un programa que calcule la suma desde 1 hasta un número Entero N, dividiéndose en 2 hilos.
Ej: N = 20, Hilo 1 de 1 a 10, Hilo 2 de 11 a 20.
- ▶ Escribe un programa que se le pase por parámetro un número entero N, y calcule la suma desde 1 hasta N, lanzando tantos hilos como sea necesario para que todos sumen la misma cantidad de números. Número máximo de Hilos será 20.
 - Ej: $10.000 / 500 = 20$ hilos y cada uno suma 500 números.
 - Calculo un divisor y lanzo tantos hilos como me de el cociente siempre que no sea superior a 20.
 - Si el número resulta ser primo, se lanzará solo 1 hilo

Práctica. Multihilo

- ▶ Escribir un programa que genere un array de N^4 números aleatorios.
- ▶ Posteriormente este programa padre creará 4 hilos que recorran cada uno $\frac{1}{4}$ del array y busque el número más alto.
- ▶ El programa principal pintará los cuatro números obtenidos de cada hijo e indicará cual es el más alto y cual es el más bajo de los cuatro.



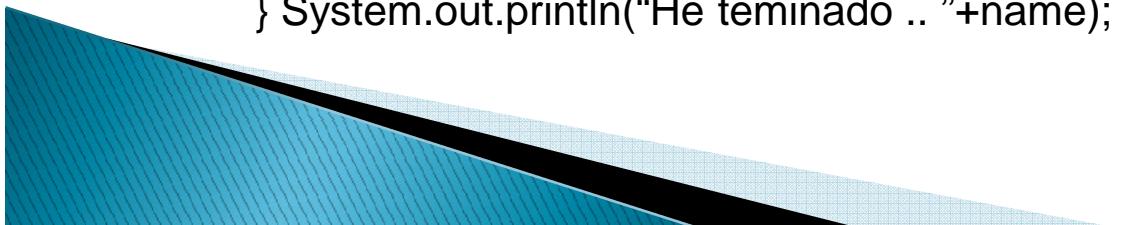
Planificación de Hilos

- ▶ Cuando se trabaja con varios hilos, a veces es necesario pensar en la planificación de *threads*, para asegurarse de que cada hilo tiene una oportunidad justa de ejecutarse.
- ▶ El planificador del sistema operativo determina qué proceso es el que se ejecuta en un determinado momento en el procesador (uno y solamente uno).
 - Dentro de ese proceso, el hilo que se ejecutará estará en función del número de núcleos disponibles y del algoritmo de planificación que se esté utilizando.
 - Java, por defecto, utiliza un planificador apropiativo cuando un hilo que se está ejecutando pasa al estado *Runnable*. Si los hilos tienen la misma prioridad, será el planificador el que asigne a uno u otro el núcleo correspondiente para su ejecución utilizando tiempo compartido.
- ▶ La Prioridad se puede establecer mediante el método **setPriority()** de la clase Thread. Las prioridades de un hilo varían entre **MIN_PRIORITY** y **MAX_PRIORITY**, definidas en la propia clase habitualmente entre 1 y 10

Un nuevo hilo hereda la prioridad del proceso que lo creo.

Planificación de Hilos. Prioridades

```
//Multihilo con Prioridades
class cuentaThread extends Thread {
    String name;    int contando;
    public cuentaThread(String name){
        super(); //llama al constructor de la clase padre
        this.name=name; }
    public void run() {
        int count=0; int espera=0;
        while (count<=1000) {
            try {
                for(espera=0;espera < 10000000;espera++)
                    this.contando=espera;
                sleep(1);
            } catch (InterruptedException e){ e.printStackTrace(); }
            //if (count==1000)
            //    count=0;
            System.out.println(name+": "+count); count++;
        } System.out.println("He terminado .. "+name); }}
```



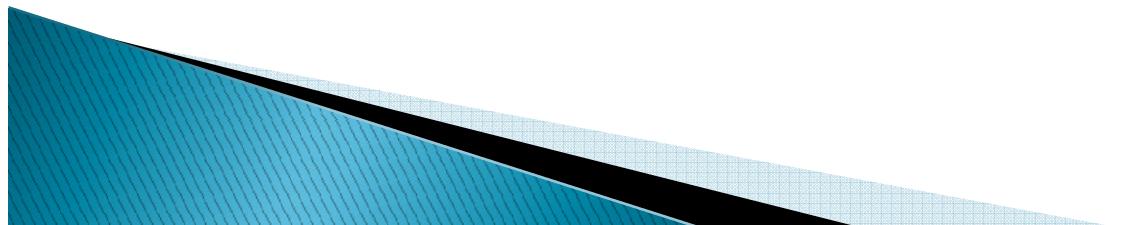
Prioridades (Cont)

```
public class multihilo_prioridades {  
    public static void main(String args[]) {  
        System.out.println("Hola desde el hilo principal!");  
  
        cuentaThread hilo1 = new cuentaThread("hilo1");  
        hilo1.setPriority(1);  
        cuentaThread hilo2 = new cuentaThread("hilo2");  
        hilo2.setPriority(1);  
        cuentaThread hilo3 = new cuentaThread("hilo3");  
        hilo3.setPriority(1);  
        cuentaThread hilo4 = new cuentaThread("hilo4");  
        hilo4.setPriority(1);  
        cuentaThread hilo5 = new cuentaThread("hilo5");  
        hilo5.setPriority(1);  
        cuentaThread hilo6 = new cuentaThread("hilo6");  
        hilo6.setPriority(10);  
  
        hilo1.start();  
        hilo2.start();  
        hilo3.start();  
        hilo4.start();  
        hilo5.start();  
        hilo6.start(); //se pueden poner join para que el padre espere  
        System.out.println("Proceso Padre acabando.");  
    }  
}
```

Práctica. Pruebas Multihilo.

Recurso Compartido

- ▶ Basándonos en las prácticas anteriores.
- ▶ Crear un programa que cree dos hilos uno que sume los 10 primeros números naturales y otro que los reste sobre la misma variable.
- ▶ Observar que ocurre.
- ▶ Probar con diferentes valores y rangos.

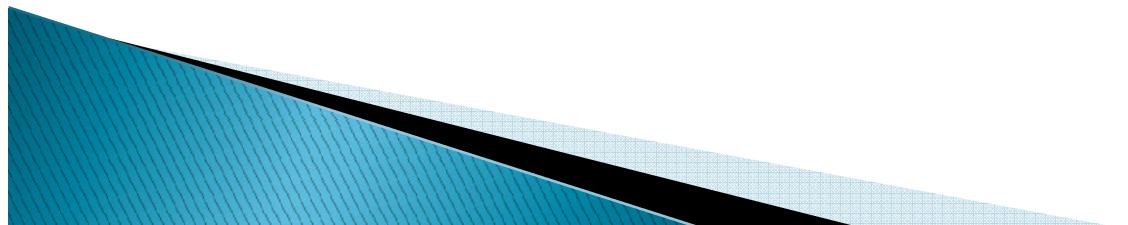


Sincronización de Hilos

- ▶ Los *threads* se comunican principalmente mediante el intercambio de información a través de variables y objetos en memoria.
 - Los *threads* pertenecen al mismo proceso, y pueden acceder a toda la memoria asignada a dicho proceso utilizando las variables y objetos del mismo para compartir información, siendo este el método de comunicación más eficiente.
- ▶ Cuando varios hilos manipulan a la vez objetos compartidos, pueden ocurrir diferentes problemas:
 - **Condición de carrera:** Resultado depende del orden de acceso a memoria
 - **Inconsistencia de memoria:** Los hilos tienen una visión diferente de lo que debe ser el mismo dato.
 - **Inanición:** A un hilo se le deniega continuamente el acceso a un recurso compartido
 - **Interbloqueo:** Uno o varios hilos esperan indefinidamente por un evento que solo puede generar un proceso o hilo bloqueado.
 - **Bloqueo activo:** Es similar al anterior pero, el estado de los dos procesos envueltos en el bloqueo activo cambia constantemente con respecto al otro.

Problemas de Sincronización

- ▶ CONDICIONES DE CARRERA
 - Si el resultado de la ejecución de un programa depende del orden concreto en que se realicen los accesos a memoria.
- ▶ Ej: sumar o restar 1 en ensamblador
 - registroX = cuenta
 - registroX = registroX (operación: suma o resta) 1
 - cuenta = registroX
- ▶ Supongamos *cuenta* vale 10, y dos hilos *sumador* y *restador* ejecutándose a la vez sobre *cuenta*. Puede suceder
 - T0: *sumador* registro1 = cuenta {registro1 = 10}
 - T1: *sumador* registro1 = registro1 + 1 {registro1 = 11}
 - T2: *restador* registro2 = cuenta {registro2 = 10}
 - T3: *restador* registro2 = registro2 - 1 {registro2 = 9}
 - T4: *sumador* cuenta = registro 1 {cuenta = 11}
 - T5: *restador* cuenta = registro2 {cuenta = 9}
- ▶ El resultado final *cuenta* = 9 es incorrecto. Condición de carrera



Problemas de Sincronización

▶ INCONSISTENCIA DE MEMORIA

Se produce cuando diferentes hilos tienen una visión diferente de lo que debería ser el mismo dato.

▶ INANICIÓN

- Cuando un proceso nunca llega a tomar el control de un recurso debido a que el resto siempre toman el control antes que él por diferentes motivos.

▶ INTERBLOQUEO

- Se produce cuando dos o más procesos o hilos están esperando indefinidamente por un evento que solo puede generar un proceso o hilo bloqueado

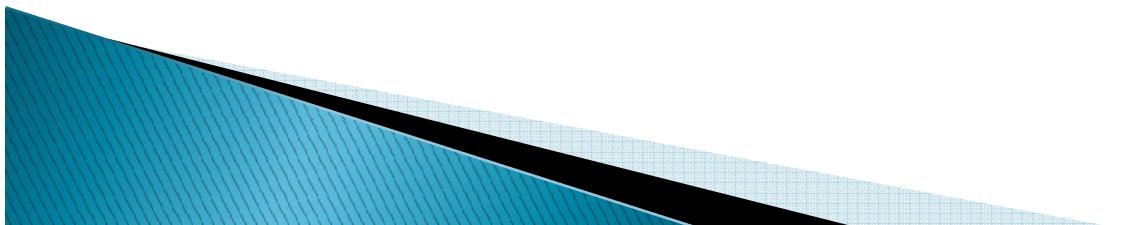
▶ BLOQUE ACTIVO

- Es similar a un interbloqueo, excepto que el estado de los dos procesos envueltos en el bloqueo activo cambia constantemente con respecto al otro.
- Frente al interbloqueo, donde los procesos se encuentran bloqueados, en un bloqueo activo no lo están, sino que es una forma de inanición debido a que un proceso no deja avanzar al otro.



Sincronización de hilos

- ▶ Las condiciones de carrera y las inconsistencias de memoria se producen porque se ejecutan varios hilos concurrentemente pudiendo ser ordenados de forma diferente a la esperada.
 - La solución pasa por provocar que cuando los hilos accedan a datos compartidos, los accesos se produzcan de forma ordenada o **síncrona**.
 - Cuando estén ejecutando código que no afecte a datos compartidos, podrán ejecutarse libremente en paralelo, proceso también denominado ejecución **asíncrona**.



Sincronización de hilos

▶ CONDICIONES DE BERNSTEIN

Las condiciones de Bernstein describen que dos segmentos de código *i* y *j* son independientes y pueden ejecutarse en paralelo de forma asíncrona en diferentes hilos sin problemas, si cumplen una serie de condiciones:

1. **Dependencia de flujo:** todas las variables de entrada del segmento *j* tienen que ser diferentes de las variables de salida del segmento *i*. Si no fuera así, el segmento *j* dependería de la ejecución de *i*.
2. **Antidependencia:** todas las variables de entrada del segmento *i* tienen que ser diferentes de las variables de salida del segmento *j*. Es el caso contrario a la primera condición, ya que si no fuera así, el segmento *i* tendría una dependencia del otro segmento.
3. **Dependencia de salida:** todas las variables de salida del segmento *i* tienen que ser diferentes de las variables de salida del segmento *j*. En caso contrario, si dos segmentos de código escriben en el mismo lugar, el resultado será dependiente del último segmento que ejecutó.



Sincronización de hilos

OPERACIÓN ATÓMICA

- Es una operación que sucede completa sin interrupciones, por lo que ningún otro hilo puede leer o modificar datos relacionados mientras se esté realizando la operación.
- En sistemas multiprocesador, con múltiples hilos de ejecución, asegurar atomicidad es muy complicado.
- Una forma de asegurar atomicidad es mediante la creación de **Secciones críticas**.



Sincronización de hilos

SECCIÓN CRÍTICA

- ▶ Se denomina **sección crítica** a una región de código en la cual se accede de forma ordenada a variables y recursos compartidos, de forma que se puede diferenciar de aquellas zonas de código que se pueden ejecutar de forma asíncrona. Este concepto se puede aplicar tanto a hilos como a procesos concurrentes, la única condición es que comparten datos o recursos.

```
do {  
    Sección de entrada;  
    SECCIÓN CRÍTICA  
    Sección de salida;  
  
    SECCIÓN RESTANTE  
} while (TRUE);
```

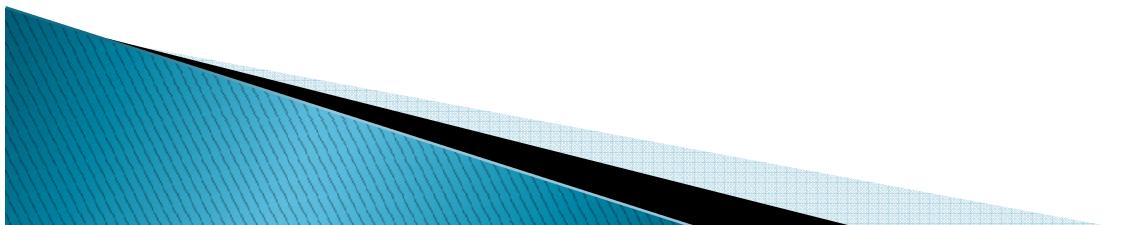
- ▶ Cuando un proceso está ejecutando su sección crítica, ningún otro proceso puede ejecutar su correspondiente sección crítica, ordenando de esta forma la ejecución concurrente
Esto garantiza que los cambios en los datos son visibles a todos los procesos.

Sincronización de hilos

- ▶ El problema de la sección crítica consiste en diseñar un protocolo que permita a los procesos cooperar. Cualquier solución al problema de la sección crítica debe cumplir:
 1. **Exclusión mutua**: si un proceso está ejecutando su sección crítica, ningún otro proceso puede ejecutar su sección crítica.
 2. **Progreso**: si ningún proceso está ejecutando su sección crítica y hay varios procesos que quieren entrar en su sección crítica, solo aquellos procesos que están esperando para entrar pueden participar en la decisión de quién entra definitivamente.
 3. **Espera limitada**: debe existir un número limitado de veces que se permite a otros procesos entrar en su sección crítica después de que otro proceso haya solicitado entrar en la suya y antes de que se le conceda.
- ▶ Para la implementación de la sección crítica se necesita un mecanismo de sincronización que actúe tanto antes de entrar en la sección crítica como después de salir de ejecutarla.

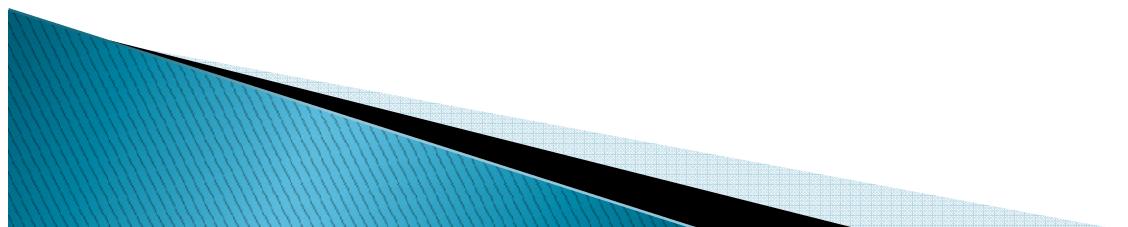
Mecanismos de sincronización.SEMAFOROS

- ▶ Un semáforo se representa como:
 - Variable entera donde su valor representa el número de instancias libres o disponibles en el recurso compartido.
 - Cola donde se almacenan los procesos o hilos bloqueados esperando para usar el recurso.



Mecanismos de sincronización. SEMAFOROS

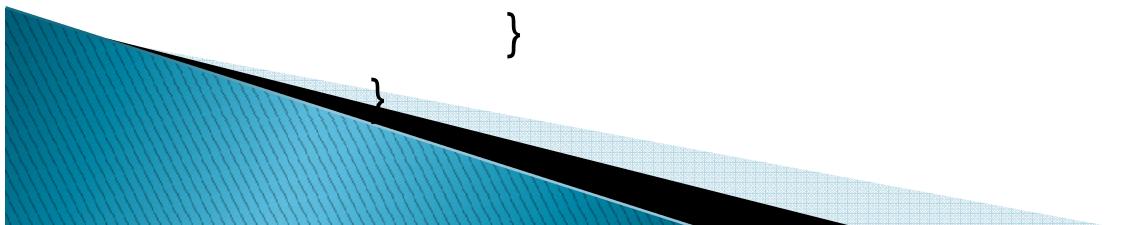
- ▶ Se accede mediante dos operaciones atómicas
 - **Wait(espera).**
 - Disminuye el número de estancias disponibles en uno, ya que se supone que la va a utilizar. Si el valor es menor que 0, significa que no hay estancias. El proceso queda bloqueado hasta que el semáforo se libere. El valor negativo especifica cuántos hilos están bloqueados esperando por el recurso.
 - **Signal(señal).**
 - Cuando se termina de usar la instancia del recurso compartido se avisa de su liberación mediante esta señal y aumentar en uno al semáforo. Si hay hilos bloqueados despertara uno de forma aleatoria, Esto depende de la implementación del semáforo y del S.O.



Mecanismos de Sincronización.Semáforos

- ▶ ***Las operaciones wait y signal deben ser atómicas para evitar problemas***

```
wait(Semaphore S) {  
    S.valor--;  
    if (S.valor < 0) {  
        Añadir el proceso o hilo a la lista S.cola  
        Bloquear la tarea  
    }  
}  
  
signal(Semaphore S) {  
    S.valor++;  
    if(S.valor <= 0) {  
        Sacar una tarea P de la lista S.cola  
        Despertar a P  
    }  
}
```



Mecanismos de sincronización. SEM. MUTEX

- ▶ **MUTEX** (Semáforo Binario)
 - Semáforo binario, también denominado *mutex* (*MUTual EXclusion*, “exclusión mutua” en español) que registra si un único recurso está disponible o no. Un *mutex* solo puede tomar los valores 0 y 1.
 - **Con un semáforo binario se puede resolver el problema de la sección crítica:**

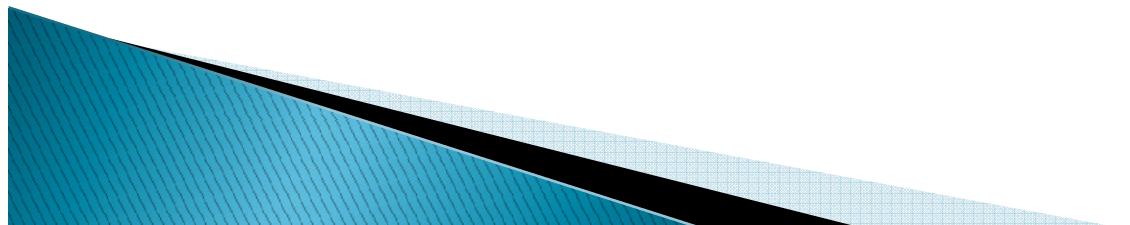
```
//Inicializado a 1 y compartido por varios procesos
Semáforo S;
wait(S);
    SECCIÓN CRÍTICA
signal(S);
```



Clase Semaphore

- ▶ En Java, el uso de semáforos se realiza mediante el paquete **java.util.concurrent** y la clase **Semaphore**.

Método	Tipo de Retorno	Descripción
Semaphore (int valor)	void	Inicialización del semáforo. Indica el valor inicial del semáforo antes de comenzar su ejecución
acquire()	void	Implementa operación wait
release()	void	Implementa operación signal. Desbloquea un hilo que esté esperando en el método wait()



Ej. Orden de ejecución de dos hilos

```
import java.util.concurrent.Semaphore;

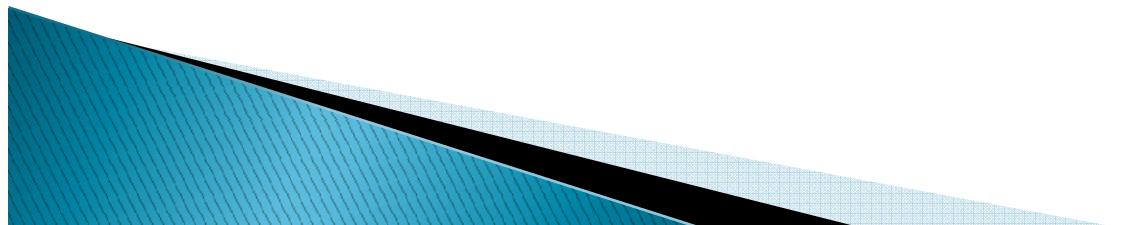
class Saludo extends Thread {
    private Semaphore sem;
    private int id;
    Saludo (int orden, Semaphore s) {
        this.id = orden;
        this.sem = s;
    }
    public void run() {
        if (id == 1)
        {
            try {
                sem.acquire();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        System.out.println("Hola, soy el thread " + id);
        if (id == 2){
            sem.release();
        }
    }
}
```

```
public class Orden {
    public static void main(String[] args) {
        Semaphore semaphore = new Semaphore(0);
        Saludo t1 = new Saludo(1,semaphore);
        Saludo t2 = new Saludo(2,semaphore);
        t1.start();
        t2.start();
        try {
            t1.join();
            t2.join();
        } catch (InterruptedException e) {
            System.out.println("Hilo principal
                del proceso interrumpido.");
        }
        System.out.println("Proceso acabando.");
    }
}
```

Cual se ejecuta Primero?

Ej. Semáforo con valor 1

- ▶ Se usa en este caso para crear sección crítica para proteger un acumulador al que acceden muchos hilos sumadores.



Ej. Semáforo de valor 1

```
import java.util.concurrent.Semaphore;

class Acumula{
public static int acumulador=0; //Inicializa el acumulador a 0
}
class Sumador extends Thread {
private int cuenta;
private Semaphore semaforo;

//Constructor
Sumador (int hasta, int id, Semaphore semaforo){
this.cuenta=hasta;
this.semaforo=semaforo;
}
//Método incrementa el acumulador
public void sumar(int Acu){
Acumula.acumulador=Acu+1;
}

public void run(){
Int Acu;
for (int i=0; i<cuenta;i++){
try{
semaforo.acquire(); //Señal Wait del Semaforo. Disminuye en 1 el semaforo
Acu=Acumula.acumulador;
System.out.println("En el hilo " + this.getName() + " Acu = " +
Acu + " Vuelta " + i);
sumar(Acu);
semaforo.release(); //Señal Signal del semaforo. Incrementa en 1 el semaforo

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

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

```
public class semaforoseccioncritica {

private static Sumador sumadores[];
private static Semaphore semaforoprincipal= new Semaphore(1);

public static void main (String[] args) {

int n_sum=4; //int n_sum=Integer.parseInt(args[0]);

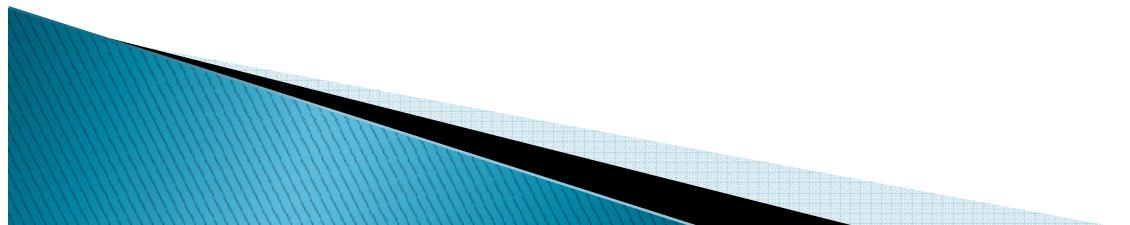
//Crea Array de procesos Sumadores
sumadores=new Sumador[n_sum];
//Inicia los Procesos del Array
for(int i=0;i<n_sum;i++){
sumadores[i]=new Sumador(10000,i,semaforoprincipal);
sumadores[i].start();

}
//Indica al Programa Principal espere a que todos los Procesos estén muertos
for (int i=0; i<n_sum;i++){
try{
sumadores[i].join();
} catch (InterruptedException e){
e.printStackTrace();
}
}

System.out.println ("Acumulador; "+ Acumula.acumulador);
}
}
}
```

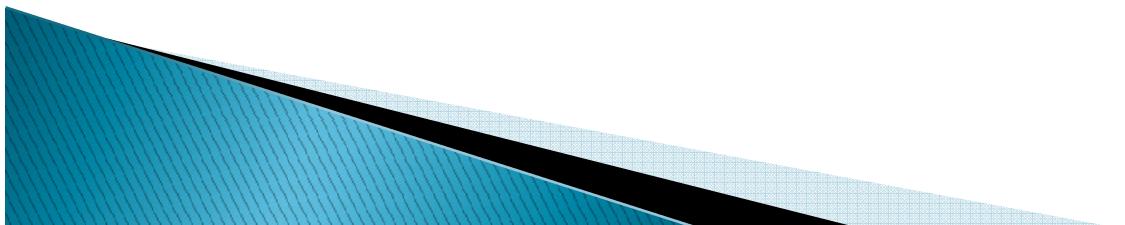
Prácticas Semáforos

- ▶ Implementar los ejercicios vistos en clase mediante semáforos.
- ▶ Programa que implementa un hilo que suma y otro que hilo que resta sobre el mismo objeto resultado. (Ejercicio planteado al inicio de Semáforos)
- ▶ Programa que suma N números Enteros y cada hilo suma N/2 pero ambos escriben su resultado en el mismo objeto resultado_suma.



Prácticas de Semáforos. Aeropuerto

- ▶ Implementar aplicación de control de aterrizaje de aviones en un aeropuerto con una sola pista.
- ▶ El tiempo ambiente influye en el tiempo que el avión tarda en aterrizar. (1 soleado hasta 5 nieve)
- ▶ Visualizar los aviones aterrizados y los pendientes de aterrizar en cola.
- ▶ Cada X tiempo llega un avión nuevo al aeropuerto solicitando aterrizaje.
- ▶ Una vez implementado con una. Probar con varias pistas.



Propuesta solución Aeropuerto

Clase aviones extends Thread

Void run

Solicitar aterrizaje

Pista.acquire()

Aviones esperando –

Voy a aterrizar, Aterrizando

Sleep(n) según el tiempo que haga

Aterrizado

Avionesaterrizados ++

Pista.release()

Class Tiempo extends Thread

Aleatorio calculo de tiempo

Mientras cierto

t= aleatorio.nextInt(5)

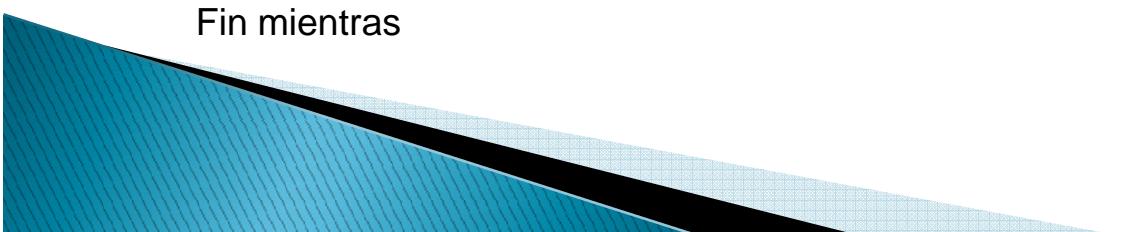
Sleep(1000)

Fin mientras

main

```
Int Tiempoactual,  
aviones_aterrizados;  
aviones_esperando;  
Semaphore Pista =new Semaphore(1)  
Tiempo t= new tiempo;  
tiempo.start()  
Desde ( 1 hasta 100)  
Aviones avion=new aviones  
Avion.start()  
Avionesesperando++  
Sleep(1000) hasta el próx avión  
Fin desde  
Esperar aviones.  
detener tiempo
```

Fin main



Prácticas Semáforos. Tarro Galletas

- ▶ <https://encodingthecode.wordpress.com/2013/04/05/seguimos-con-concurrencia-ejemplo-de-uso-de-semaforos/>
- ▶ Supongamos que disponemos de un tarro de galletas, y ‘k’ niños accediendo al bote, cuando este toca a su fin, los niños avisarán a la madre para que reponga el tarro
- ▶ Primero, tendremos 3 clases en nuestro proyecto JAVA, una clase ‘niño’, encargada de actuar como el monstruo de las galletas xD, la clase ‘mama’, encargada de reponer el tarro cuando los niños lloren porque se acabó su “merienda” (al estilo Shin Chan xD), y por último, una clase principal encargada de ‘crear’ a los niños y a la madre, para después lanzarlos

