

PROGRAMACIÓN DE SERVICIOS Y PROCESOS

- Unidad 2: Programación multihilo -

Profesor: Marcelino Penide

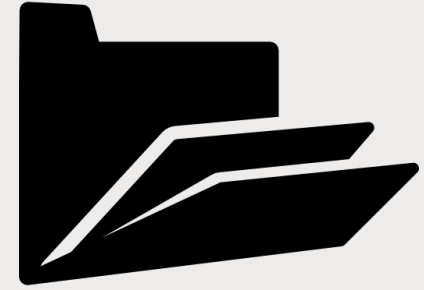
marcelimpd@educastur.org

[Curso 2023 - 2024]

(2º

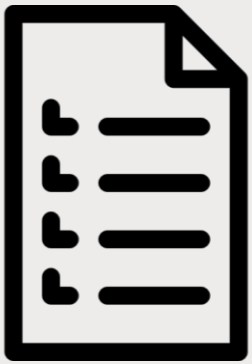
DAM)

Contenidos unidad



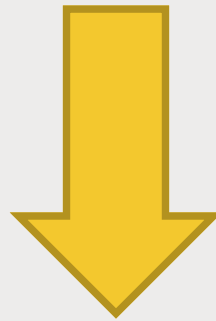
1. Introducción.
2. Conceptos básicos.
3. Hilos en Java:
 - 3.1. Aspectos básicos.
 - 3.2. Aspectos avanzados.
4. Aplicaciones multihilo.

1. INTRODUCCIÓN



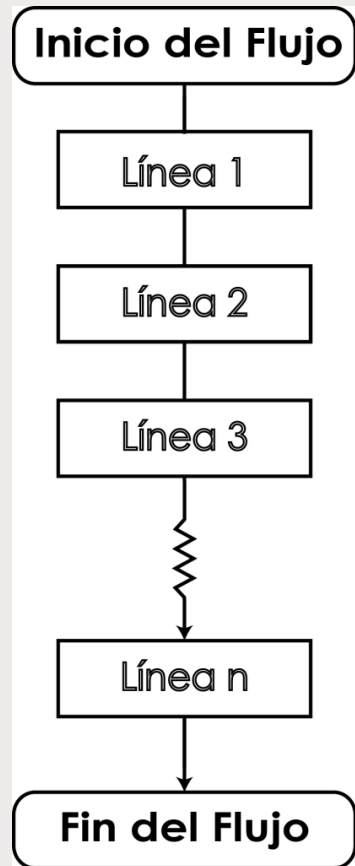
Flujos de ejecución (1/2)

- **Programa:** realiza actividades / tareas.
 - **De flujo único:** una a continuación de la otra (secuencialmente).
 - **De flujo múltiple:** “a la vez” (simultáneamente).

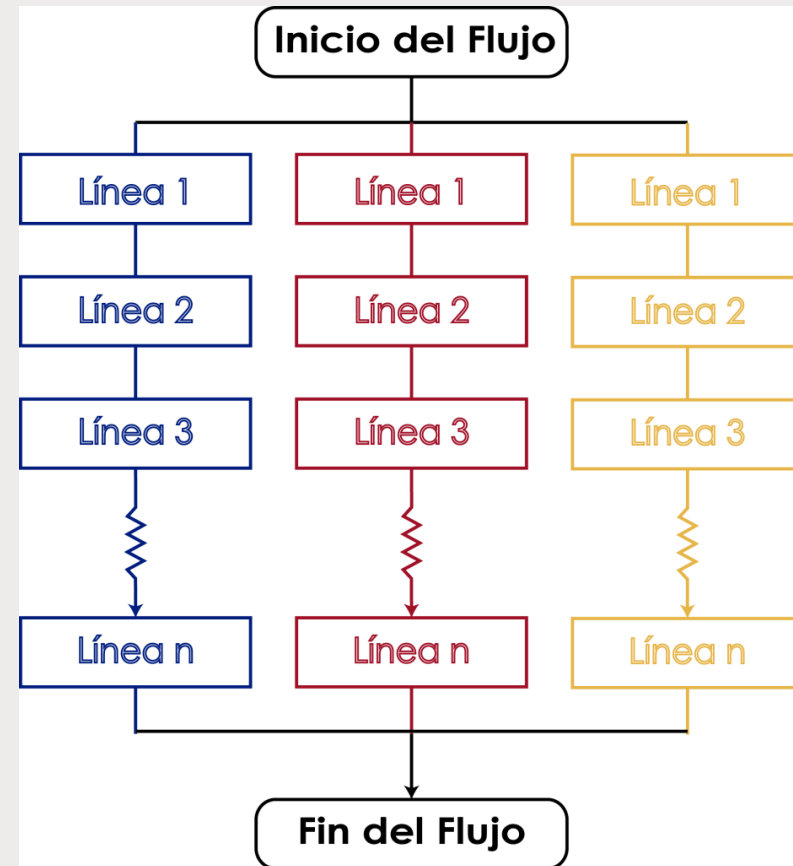


Programación multihilo (cada flujo = hilo / thread)

Flujos de ejecución (2/2)



(Flujo único)



(Flujo múltiple)

2. CONCEPTOS BÁSICOS

Hilo (o subproceso)

- **Flujo de control secuencial independiente dentro de un proceso asociado con:**
 - Una secuencia de instrucciones.
 - Un conjunto de registros.
 - Una pila.



Hilo: elementos propios

- Identificador único.
- Contador de programa propio.
- Conjunto de registros.
- Pila (variables locales).

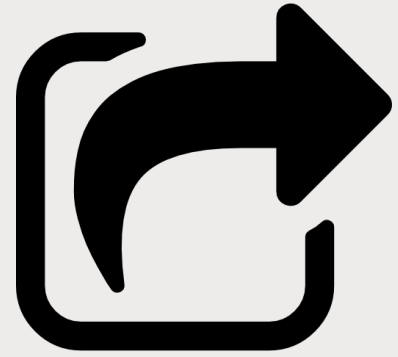


Hilo: elementos compartidos

- Código.
- Datos (variables globales).
- Otros recursos: ficheros,...



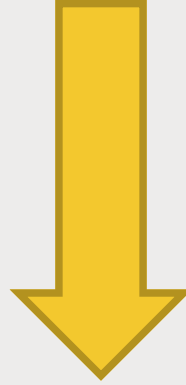
Necesario: utilizar esquemas de bloqueo y sincronización





Hilos: ventajas (vs. procesos)

1. Más rápidos (menos tiempo: crear y terminar).
2. Consumen menos recursos.
3. Cambio contexto más rápido.



“Procesos ligeros”

Hilos: uso recomendado

1. Aplicación > debe poder realizar diferentes tareas a la vez.
2. Aplicación > maneja entradas de varios dispositivos de comunicación.
3. Aplicación > se ejecuta en un entorno multiprocesador.
4. Aplicación > con tareas con prioridad variada.





CENTRO INTEGRADO de FORMACIÓN PROFESIONAL
AVILÉS

3. HILOS EN JAVA



CENTRO INTEGRADO de FORMACIÓN PROFESIONAL
AVILÉS

3.1. ASPECTOS BÁSICOS

Creación de hilos

- Clase: **Thread**
- 2 formas (a partir de):
 1. Clase Thread > **extends Thread**
 2. Interfaz Runnable > **implements Runnable**
 3. Con implementación anónima de Thread, pasando una lambda como atributo **target** (función run)
 - En los 3 casos casos: **start()** >>> **run()**

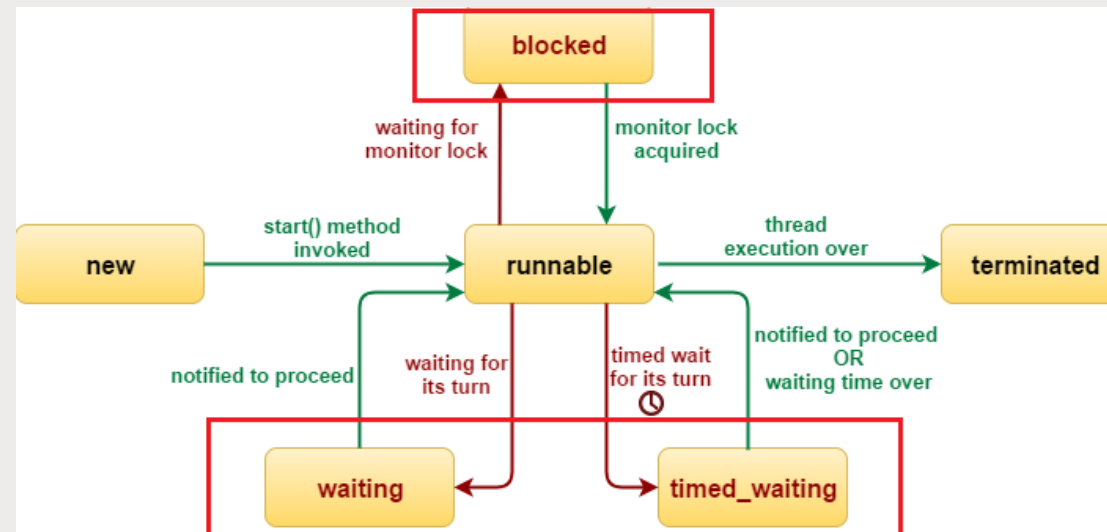
Proyecto Hilos1: Paquetes Ejemplo1, Ejemplo2 y Ejemplo3.



Ciclo de vida (1)

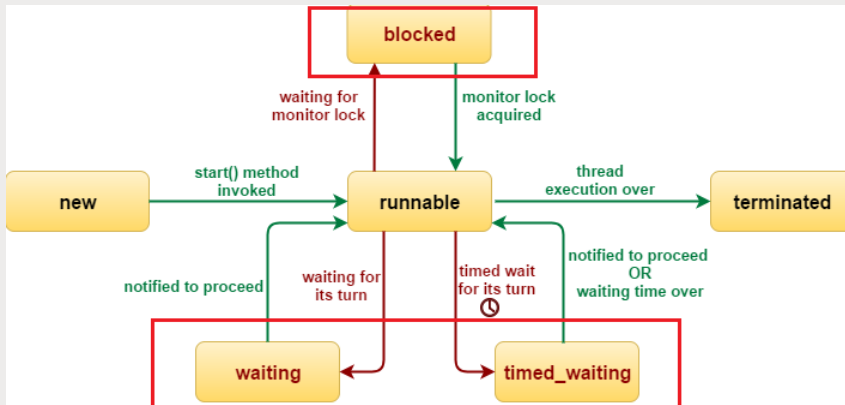
■ Diferentes estados en los que puede estar un hilo (JVM).

- **New (Nuevo):** es el estado cuando se crea un objeto hilo . Aún no se ejecuta (no ejecuta aún su método run())
- **Runnable:** al invocar start() el hilo pasa a este estado. El s.o. tiene que asignar la CPU al hilo así que puede estar o no realmente en ejecución.



Ciclo de vida

- **Terminated (muerto).**
 - Por muerte natural, al terminar run().
 - Repentinamente por alguna excepción no capturada en run().
 - Llamando a **stop()**. Desaconsejado, porque dejas los bloqueos activos.
- **Waiting (esperando).**
 - Alguien ha llamado a sleep() del hilo.
 - El propio hilo llama a wait(). No se volverá a ejecutar hasta recibir señal por notify() o notifyAll().
 - Alguien ha llamado a su método **suspend()**. Se reactivará de nuevo con **resume()**.
- **Blocked (bloqueado).**
 - Esperando por alguna operación de E/S.
 - El hilo requiere un recurso bloqueado por otro hilo.



Los métodos `resume()`, `suspend()` y `stop()` están en desuso.
→ mejor con variables

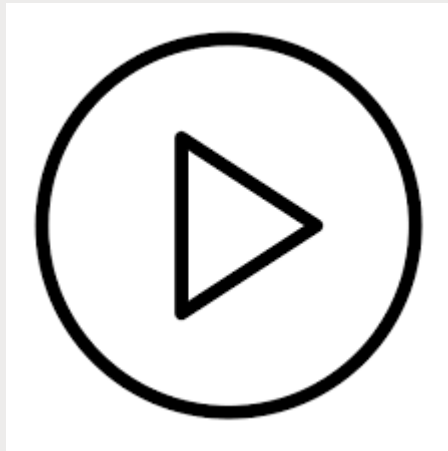
Ejemplos:

- `HiloEjemploTerminado()`

Iniciar hilo

■ Método **start()**:

- Solo puede ser llamado **1** vez (por hilo).
 - Error: `IllegalThreadStateException`
- Orden de llamada no influye en orden ejecución.
 - Orden ejecución hilos: no determinístico.



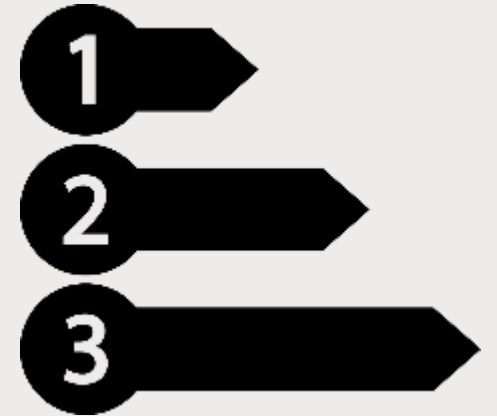
Finalizar hilo

- Hilo pasa a estado: **terminated** (muerto).
- ¿Cuándo?:
 - Forma natural: `finaliza run()`
- No se puede volver a iniciar.



Gestión de prioridad de hilos (1)

- Valor entre: 1 (mínima) y 10 (máxima).
- Por defecto, prioridad del padre.
- Constantes (estáticas en Thread)
 - MIN_PRIORITY > 1
 - NORM_PRIORITY > 5
 - MAX_PRIORITY > 10



```
getPriority():int  
setPriority(int newPriority):void
```

Gestión de prioridad de hilos (2)

El hilo de mayor prioridad sigue funcionando hasta que:

- Cede el control al planificador llamando al método `yield()`
- Deja de ser `runnable` (ya sea por muerte o por entrar en el estado de bloqueo)
- Un hilo de mayor prioridad se convierte en `runnable` (porque se encontraba dormido o su operación de E/S ha finalizado o alguien lo desbloquea llamando a los métodos `notifyAll()` / `notify()`).

El uso del método `yield()` devuelve automáticamente el control al planificador. Sin este método el mecanismo de multihilos sigue funcionando aunque algo más lentamente.



Hilos egoístas

- Hilo se ejecuta sin dar posibilidad a otros hilos.
- ¿Dónde? > SO que no implementen: **time-slicing**
- ¿Problema? > Mismo programa se ejecuta diferente según SO.



Solución: `yield()`

(Se puede interrumpir hilo + dar oportunidad a otro)

Obtener hilo actual + mostrar información

```
public class E01_ObtenerInfoHilo {  
  
    public static void main(String[] args) {  
        System.out.println("Hola mundo");  
        // Obtener hilo actual  
        Thread hilo = Thread.currentThread();  
        // Mostrar información hilo  
        System.out.println("Nombre hilo: " + hilo.getName());  
        System.out.println("ID hilo: " + hilo.getId());  
        System.out.println("Prioridad hilo: " + hilo.getPriority());  
        System.out.println("Estado hilo: " + hilo.getState());  
    }  
}
```

[1] Crear hilo (extends Thread)


```
public class E02_CrearHilo1 extends Thread {  
  
    @Override  
    public void run() {  
        System.out.println("Hilo creado usando la clase Thread");  
    }  
  
    public static void main(String[] args) {  
        // Crear hilo  
        Thread hilo = new E02_CrearHilo1();  
        // Iniciar hilo  
        hilo.start();  
    }  
}
```

[2] Crear hilo (implements Runnable)


```
public class E03_CrearHilo2 implements Runnable {  
  
    public void run() {  
        System.out.println("Hilo creado usando la interfaz Runnable");  
    }  
  
    public static void main(String[] args) {  
        // Crear hilo  
        E03_CrearHilo2 r = new E03_CrearHilo2();  
        Thread hilo = new Thread(r);  
        // Iniciar hilo  
        hilo.start();  
    }  
}
```


Crear varios hilos

```
public class E04A_CrearVariosHilos {  
  
    public static void main(String[] args) {  
        // Crear hilos  
        Thread hilo1 = new E04B_HiloThread();  
        Thread hilo2 = new E04B_HiloThread("Hola");  
        Thread hilo3 = new Thread(new E04C_HiloRunnable());  
        // Iniciar hilos  
        hilo1.start();  
        hilo2.start();  
        hilo3.start();  
    }  
}
```

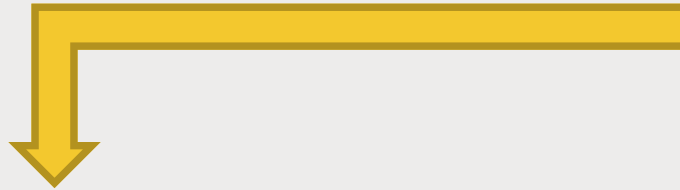


```
public class E04B_HiloThread extends Thread {  
    private String cadena;  
  
    public E04B_HiloThread() {  
        cadena = "Hilo creado usando la clase Thread";  
    }  
  
    public E04B_HiloThread(String cadena) {  
        this.cadena = cadena;  
    }  
  
    @Override  
    public void run() {  
        for(int i = 1; i < 100; i++) {  
            System.out.println(cadena);  
        }  
    }  
}
```



```
public class E04C_HiloRunnable implements Runnable {  
  
    public void run() {  
        for(int i = 1; i < 100; i++) {  
            System.out.println("Hilo creado usando la interfaz Runnable");  
        }  
    }  
}
```

Mostrar estado hilo



```
public class E05B_HiloThread extends Thread {
    private String cadena = "Hilo creado usando la clase Thread";

    @Override
    public void run() {
        for(int i = 1; i < 10; i++) {
            System.out.println(cadena);
            try {
                Thread.sleep(1000); // Detener ejecución hilo (milisegundos)
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```
public class E05A_MostrarEstadoHilo {

    public static void main(String[] args) {
        // Crear hilo
        Thread hilo = new E05B_HiloThread();
        // Mostrar información (hilo creado)
        System.out.println(">> Hilo creado");
        System.out.println("Estado = " + hilo.getState());
        System.out.println("¿Está vivo? = " + hilo.isAlive());
        // Iniciar hilo
        hilo.start();
        // Mostrar información (hilo iniciado)
        System.out.println(">> Hilo iniciado");
        System.out.println("Estado = " + hilo.getState());
        System.out.println("¿Está vivo? = " + hilo.isAlive());
        // Esperar hasta que termine el hilo
        try {
            hilo.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        // Mostrar información (hilo muerto)
        System.out.println(">> Hilo muerto");
        System.out.println("Estado = " + hilo.getState());
        System.out.println("¿Está vivo? = " + hilo.isAlive());
    }
}
```

Gestionar prioridades (1/2)

```
public class E06A_GestionarPrioridades {  
  
    public static void main(String[] args) {  
        int contador = 10;  
        // Crear arrays de hilos  
        Thread[] hiloMIN = new Thread[contador];  
        Thread[] hiloNOR = new Thread[contador];  
        Thread[] hiloMAX = new Thread[contador];  
        // Crear hilos  
        for(int i = 0; i < contador; i++) { // Prioridad mínima  
            hiloMIN[i] = new E06B_HiloThread(Thread.MIN_PRIORITY);  
        }  
        for(int i = 0; i < contador; i++) { // Prioridad estándar  
            hiloNOR[i] = new E06B_HiloThread();  
        }  
        for(int i = 0; i < contador; i++) { // Prioridad máxima  
            hiloMAX[i] = new E06B_HiloThread(Thread.MAX_PRIORITY);  
        }  
        // Iniciar hilos  
        for(int i = 0; i < contador; i++) {  
            hiloMIN[i].start();  
            hiloNOR[i].start();  
            hiloMAX[i].start();  
        }  
    }  
}
```

Gestionar prioridades (2/2)

```
public class E06B_HiloThread extends Thread {  
  
    public E06B_HiloThread() { // Constructor por defecto  
        // Hereda prioridad del hilo padre  
    }  
  
    public E06B_HiloThread(int prioridad) { // Constructor personalizado  
        setPriority(prioridad); // Establecer prioridad indicada  
    }  
  
    @Override  
    public void run() {  
        String cadena = "";  
        for(int i = 0; i < 100; i++) {  
            cadena = cadena + "P";  
            Thread.yield(); // Se puede interrumpir hilo + dar oportunidad a otro  
        }  
        System.out.println("Hilo de prioridad " + getPriority() + " acaba de terminar");  
    }  
}
```

Práctica



Práctica 2.1: Creación de hilos

Práctica 2.2: ContadorNumerosPrimos



CENTRO INTEGRADO de FORMACIÓN PROFESIONAL
AVILÉS

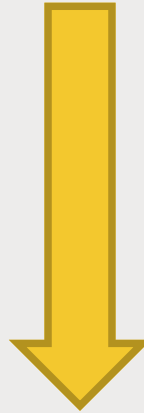
3.2. ASPECTOS AVANZADOS

Problemas trabajar con hilos



Se dan cuando

- 2 o más hilos compiten por obtener un mismo recurso (ej.: contador, fichero,...).
- 2 o más hilos colaboran para obtener un fin común (ej.: hilo produce información que necesita otro hilo).



Solución: sincronización + comunicación

Formas: sincronización + comunicación

1. Monitores:

- Bloques de código: **synchronized**

2. Notificaciones:

- `wait()` >>> `notify()` / `notifyAll()`

3. Semáforos:

- Clase > Semaphore:
 - `acquire()` y `release()`



[1] Monitores

- **Problema:** Secciones críticas que no se deben ejecutar concurrentemente (acceden a recurso compartido).
- **Solución** > sincronización (método: **synchronized**): Solo 1 hilo puede ejecutar **un bloque sincronizado de un mismo objeto** en cada instante.
- **Garantiza la atomicidad de las operaciones. Exclusión mutua.**
- Los métodos o sentencias **synchronized** en Java generan internamente un **monitor** o **bloqueo intrínseco**.
- **Más info:**
<https://docs.oracle.com/javase/tutorial/essential/concurrency/locksinc.html>

[1] Monitores: ejemplo (1/2)

```
public class E07A_GestionarFicheroHilos {  
  
    public static void main(String[] args) {  
        E07B_Fichero fichero = new E07B_Fichero("pablo.txt");  
        for(int i = 1; i <= 50; i++) {  
            new E07C_HiloSaludo(fichero).start();  
        }  
        for(int i = 1; i <= 50; i++) {  
            new E07D_HiloDespedida(fichero).start();  
        }  
    }  
}
```

Sin synchronized se puede provocar una IOException, por escribir en un flujo cerrado por otro hilo.

```
import java.io.File;  
import java.io.FileWriter;  
import java.io.IOException;  
  
public class E07B_Fichero {  
    private File fichero;  
    private FileWriter fw;  
  
    public E07B_Fichero(String nombreFichero) {  
        fichero = new File (nombreFichero);  
    }  
  
    public synchronized void insertarSaludo() {  
        try {  
            fw = new FileWriter(fichero, true);  
            fw.write("Hola\n");  
            fw.close();  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
    }  
  
    public synchronized void insertarDespedida() {  
        try {  
            fw = new FileWriter(fichero, true);  
            fw.write("Adiós\n");  
            fw.close();  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

[1] Monitores: ejemplo (2/2)

```
public class E07C_HiloSaludo extends Thread {
    private E07B_Fichero fichero;

    public E07C_HiloSaludo(E07B_Fichero fichero) {
        this.fichero = fichero;
    }

    @Override
    public void run() {
        fichero.insertarSaludo();
    }
}
```



```
public class E07D_HiloDespedida extends Thread {
    private E07B_Fichero fichero;

    public E07D_HiloDespedida(E07B_Fichero fichero) {
        this.fichero = fichero;
    }

    @Override
    public void run() {
        fichero.insertarDespedida();
    }
}
```



[2] Notificaciones - Motivación

- Synchronized garantiza atomicidad (exclusión mutúa), pero no define el orden de ejecución.
- A veces nos interesa que un hilo se quede bloqueado a la espera de que ocurra algún evento, como la llegada de un dato para tratar o que el usuario termine de escribir algo en una interface de usuario.
- podemos esperar a que se de una condición
 - Durmiendo (sleep()). → Tendremos que despertar cada poco y comprobar.
 - Dando vueltas un bucle → “Espera Activa” → Malgasto de CPU
 - ¡ojo con esperar dentro de una zona crítica! → Riesgo de interbloqueo (deadlock)

¿no sería mejor poner un despertador?

- que nos avise de que la situación ha cambiado
- minimizando el bloqueo de zonas críticas.



[2] Notificaciones - Implementación

- Las notificaciones se realizan desde instancias de la clase Object.
- Los hilos esperan por estados de objetos.
- Funciones:
 - **wait()**: detiene el hilo a la espera del objeto y libera su bloqueo (synchronized) hasta ser notificado.
 - **notify()**: notifica **al primer hilo** en hacer wait() en ese objeto que ya puede continuar.
 - **notifyAll()**: notifica **a todos** los hilos puestos en espera en ese objeto que ya pueden continuar.

```
synchronized method(...) {  
    while(!condicion())  
        wait();  
    //Otro hilo llama a notify(). Fin wait  
    //hago mis cosas;  
    notifyAll();  
}
```

Estas funciones se deben llamar **SIEMPRE** desde bloques synchronized.

El hilo debe ser el propietario del monitor y lo libera hasta que lo notifiquen. → Recupera el monitor.

WAITING → RUNNABLE

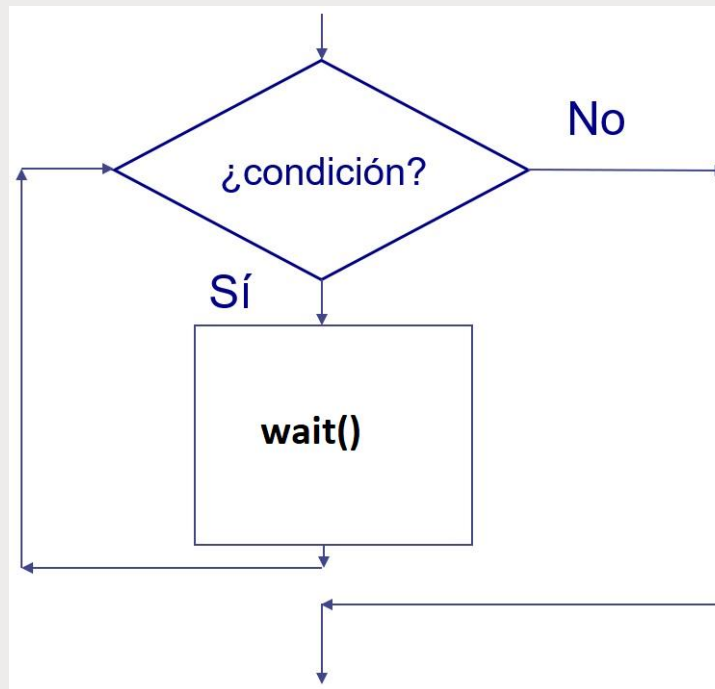
[2] Notificaciones – Por qué en while?



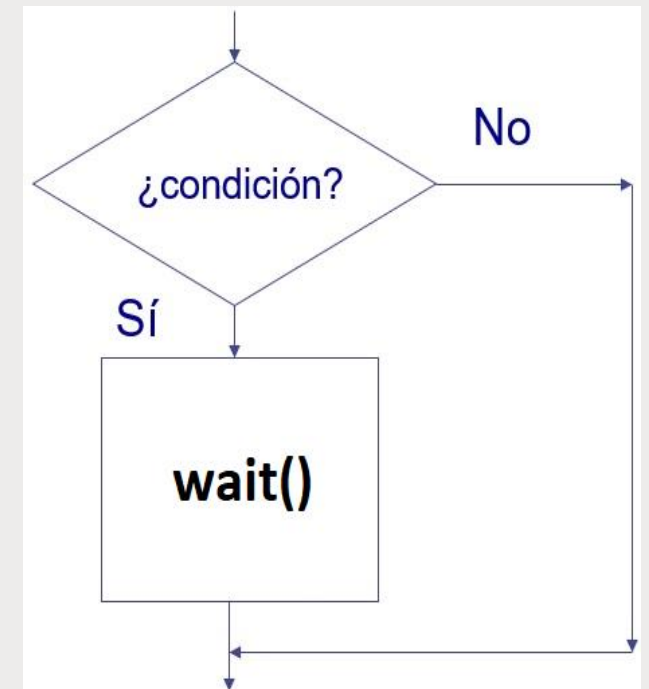
Por qué un while y no un if?

Tras liberarse el bloqueo, se debe volver a comprobar la condición que lo provocó antes de proceder con código protegido.

```
synchronized method(...) {  
    while(!condicion())  
        wait();  
    //Otro hilo llama a notify(). Fin wait  
    //hago mis cosas;  
    notifyAll();  
}
```

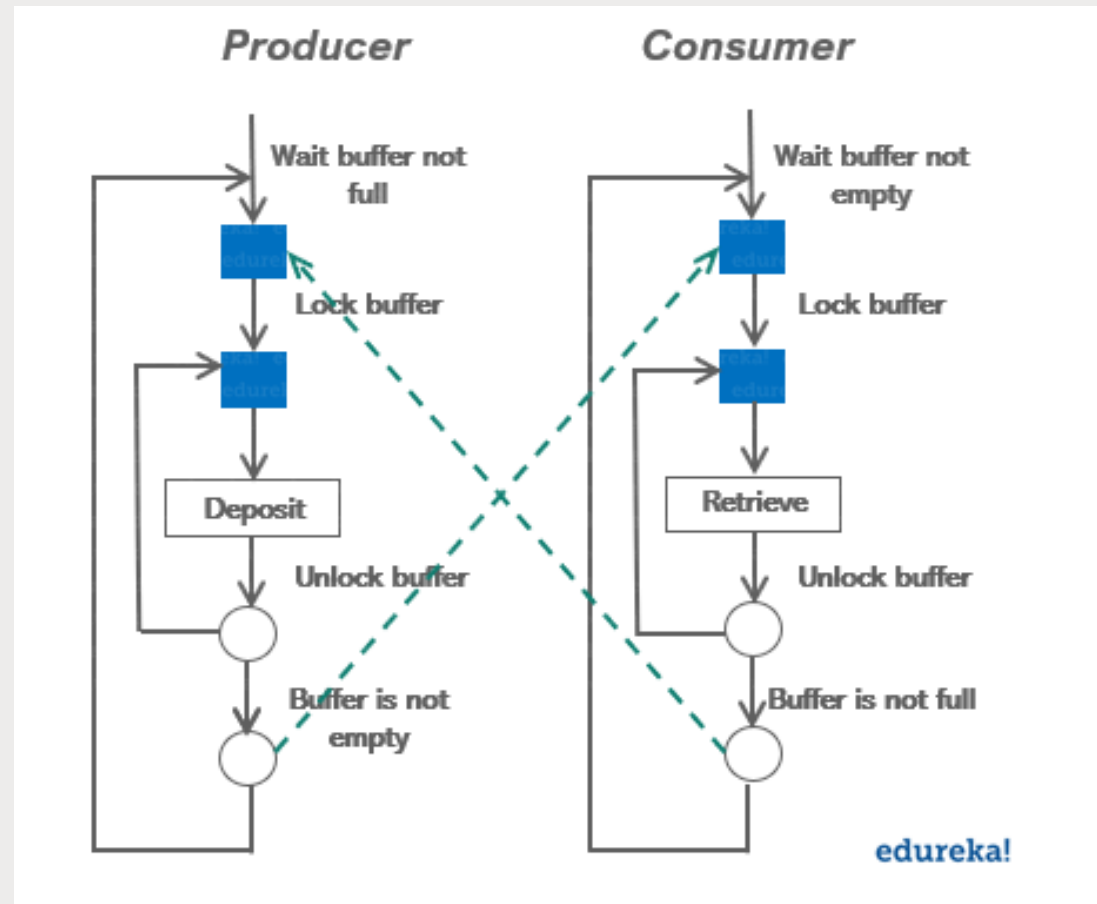


Flujo seguro, con while



Flujo peligroso, con if

[2]Notificaciones. Productor/Consumidor



[2] Notificaciones: ejemplo (1/3)

```
public class E08A_GestionarAlmacenCuadros {  
  
    public static void main(String[] args) {  
        // Crear almacén  
        E08B_AlmacenCuadros almacen = new E08B_AlmacenCuadros();  
        // Crear hilos  
        Thread pintor = new E08C_HiloPintor(almacen);  
        Thread vendedor = new E08D_HiloVendedor(almacen);  
        // Iniciar hilos  
        pintor.start();  
        vendedor.start();  
    }  
}
```


[2] Notificaciones: ejemplo (2/3)

```
public class E08B_AlmacenCuadros {  
    private int cuadros = 0;  
  
    public synchronized void guardar() { // Añadir cuadro  
        while(cuadros != 0) { // Mientras que hay cuadros en el almacén  
            try {  
                wait(); // Esperar  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
        }  
        System.out.println("Nº cuadros: " + cuadros);  
        cuadros++;  
        System.out.println("Cuadro en almacén");  
        notify(); // Notificar que se ha producido el evento  
    }  
  
    public synchronized void sacar() { // Vender cuadro  
        while(cuadros == 0) { // Mientras que no hay cuadros en el almacén  
            try {  
                wait(); // Esperar  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
        }  
        System.out.println("Nº cuadros: " + cuadros);  
        cuadros--;  
        System.out.println("Vendido cuadro. Sale de almacén");  
        notify(); // Notificar que se ha producido el evento  
    }  
}
```

[2] Notificaciones: ejemplo (3/3)

```
public class E08C_HiloPintor extends Thread {
    private E08B_AlmacenCuadros almacen;

    public E08C_HiloPintor(E08B_AlmacenCuadros almacen) {
        this.almacen = almacen;
    }

    @Override
    public void run() { // Pintar (y guardar) 30 cuadros en almacén
        for(int i = 1; i <= 30; i++) {
            almacen.guardar();
        }
    }
}
```



```
public class E08D_HiloVendedor extends Thread {
    private E08B_AlmacenCuadros almacen;

    public E08D_HiloVendedor(E08B_AlmacenCuadros almacen) {
        this.almacen = almacen;
    }

    @Override
    public void run() { // Vender (y sacar) 30 cuadros del almacén
        for(int i = 1; i <= 30; i++) {
            almacen.sacar();
        }
    }
}
```

Problema a evitar

- **Interbloqueo / bloqueo mutuo (deadlock):** uno o más hilos se bloquean o esperan indefinidamente.
- **Casos:**
 - *Cada hilo espera a que le llegue un aviso de otro hilo que nunca le llega.*
 - *Todos los hilos esperan para acceder a un mismo recurso.*



Práctica 2.3



- Crear un programa en Java usando hilos (mediante notificaciones) que simule el funcionamiento de una panadería teniendo en cuenta que:
 - Habrá un panadero y un vendedor.
 - Cuando se fabrica una barra de pan se deposita en el mostrador.
 - En el mostrador sólo puede haber 2 barras de pan como máximo (hasta que no se venden no se fabrican más).

[Simular: fabricación y venta de 50 barras de pan].



[3] Semáforos (1/2)



- Clase: **Semaphore**
- Uso: controlar acceso a recurso compartido.
- Constructor: **Semaphore (int permisos)**
 - Permisos semáforo binario:
 - 0 > bloqueado.
 - 1 > desbloqueado.

[3] Semáforos (2/2)



- Métodos importantes:
 - **acquire()** > adquirir semáforo:
 - Contador interno 0: espera.
 - Contador interno 1: decrementa 1 y sigue.
 - Si no hay “permits” disponibles, el hilo pasa a waiting.
 - **release()** > liberar semáforo:
 - Contador interno: incrementa 1 y sigue.

[3] Semáforos: ejemplo (1/3)

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

public class E09A_GestionarAlmacenCuadros {

    public static void main(String[] args) {
        // Crear semáforos
        Semaphore semaforoPintor = new Semaphore(1);
        Semaphore semaforoVendedor = new Semaphore(0);
        // Crear almacén
        E09B_AlmacenCuadros almacen = new E09B_AlmacenCuadros(semaforoPintor, semaforoVendedor);
        // Crear hilos
        Thread pintor = new E09C_HiloPintor(almacen);
        Thread vendedor = new E09D_HiloVendedor(almacen);
        // Iniciar hilos
        pintor.start();
        vendedor.start();
    }
}
```

[3] Semáforos: ejemplo (2/3)



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

public class E09B_AlmacenCuadros {
    private Semaphore semaforoPintor;
    private Semaphore semaforoVendedor;
    private int cuadros = 0;

    public E09B_AlmacenCuadros(Semaphore semaforoPintor, Semaphore semaforoVendedor) {
        this.semaforoPintor = semaforoPintor;
        this.semaforoVendedor = semaforoVendedor;
    }

    public void guardar() { // Añadir cuadro
        try {
            semaforoPintor.acquire();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Nº cuadros: " + cuadros);
        cuadros++;
        System.out.println("Cuadro en almacén");
        semaforoVendedor.release();
    }

    public void sacar() { // Vender cuadro
        try {
            semaforoVendedor.acquire();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Nº cuadros: " + cuadros);
        cuadros--;
        System.out.println("Vendido cuadro. Sale de almacén");
        semaforoPintor.release();
    }
}
```


[3] Semáforos: ejemplo (3/3)



```
public class E09C_HiloPintor extends Thread {
    private E09B_AlmacenCuadros almacen;

    public E09C_HiloPintor(E09B_AlmacenCuadros almacen) {
        this.almacen = almacen;
    }

    @Override
    public void run() { // Pintar (y guardar) 30 cuadros en almacén
        for(int i = 1; i <= 30; i++) {
            almacen.guardar();
        }
    }
}
```

```
public class E09D_HiloVendedor extends Thread {
    private E09B_AlmacenCuadros almacen;

    public E09D_HiloVendedor(E09B_AlmacenCuadros almacen) {
        this.almacen = almacen;
    }

    @Override
    public void run() { // Vender (y sacar) 30 cuadros del almacén
        for(int i = 1; i <= 30; i++) {
            almacen.sacar();
        }
    }
}
```



CENTRO INTEGRADO de FORMACIÓN PROFESIONAL
AVILÉS

4. APLICACIONES MULTIHILO

Propiedades

- **Seguridad** > utilizar correctamente recursos compartidos.
- **Viveza** > ausencia de: interbloqueos / bloqueos mutuos (deadlock).



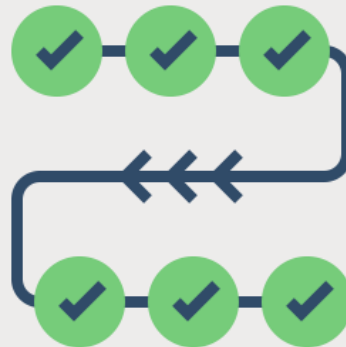
Características: uso librerías

1. Facilitan la programación.
2. Mayor rendimiento.
3. Mayor fiabilidad.
4. Menor mantenimiento.
5. Mayor productividad.



Otras utilidades de concurrencia

- Dentro de `> java.util.concurrent` (multitud de herramientas):
 - Interfaz: `Executor`
 - Paquete: `locks (java.util.concurrent.locks)`
 - ...



Software de calidad (tareas a realizar)

- **Documentación interna** > aportar legibilidad.
- **Depuración** > corregir fallos y/o errores.
 - Problema: complicada en aplicaciones multihilo.
 - `getStackTrace()` > muestra información hilo concreto.
 - `getAllStackTraces()` > muestra información hilos vivos.



Tarea > Prog2.4



- Realiza el Prog2.3 usando semáforos.



Tarea > Prog2.5



- Crear un programa en Java usando hilos:
 - Hilo 1: mostrará 50 veces la palabra PIN.
 - Hilo 2: mostrará 50 veces la palabra PAN.
 - Hilo 3: mostrará 50 veces la palabra PUN.

Utilizando notificaciones debes conseguir que se muestre por pantalla:

PIN PAN PUN PIN PAN PUN...



Tarea > Prog2.6



- Realiza el Prog2.5 usando semáforos.

