

Unidad 1

Programación concurrente

Tema 1 Conceptos

Contenidos

- Introducción
- Estructuras básicas de soporte a la concurrencia en Java
 - Threads, ciclo de vida, prioridades, interrupciones, estados
 - Ejecutores
 - Ejemplo Partículas en movimiento
- Objetos y concurrencia
 - Concurrencia
 - Estructuras para la ejecución concurrente
 - Concurrencia y programación OO
 - Transformaciones y modelos de objetos
- Imposiciones de diseño
- Bibliografía

Introducción

- Un **programa concurrente**

Es un programa que hace más de una cosa a la vez

- Una definición de **proceso o hilos**

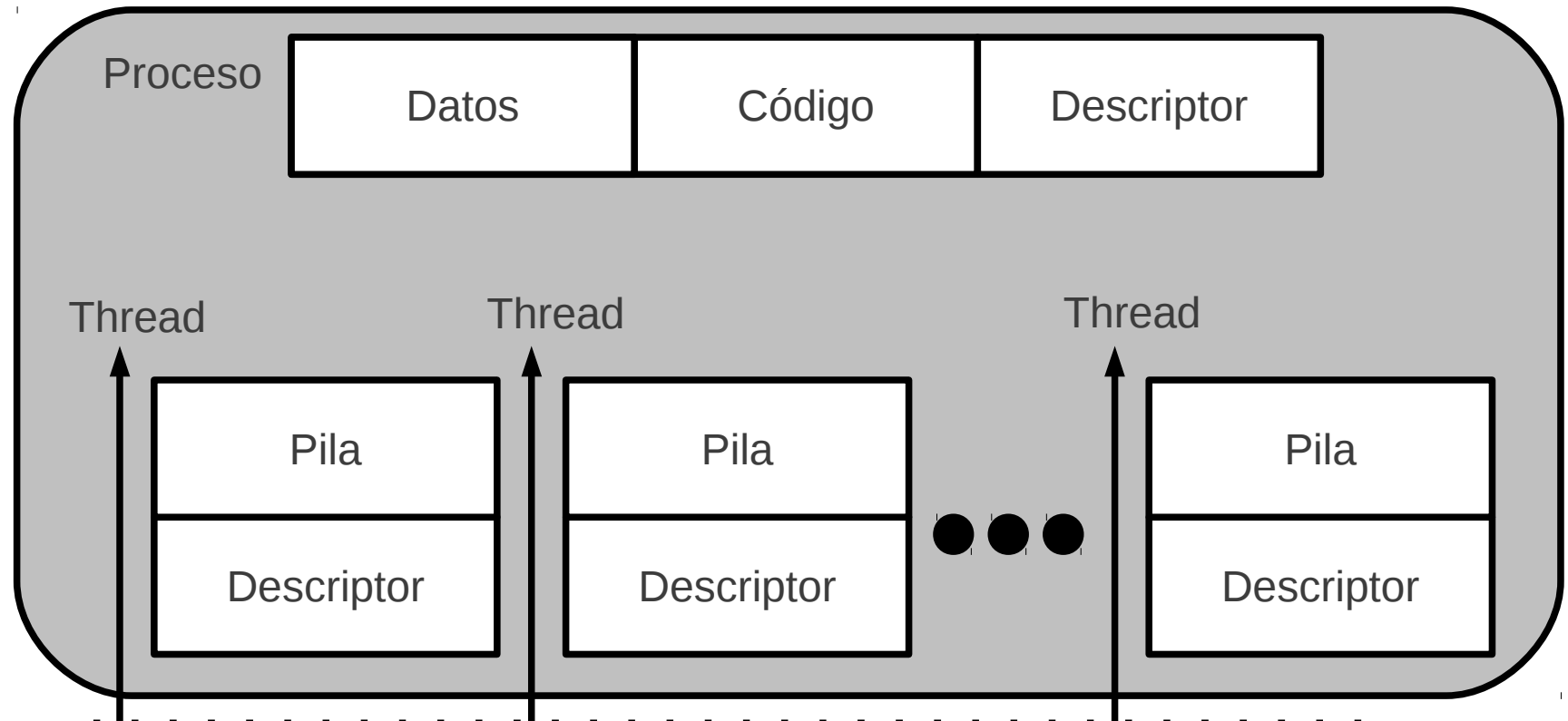
Secuencia de llamadas que se ejecuta independientemente de otras, mientras que posiblemente al mismo tiempo, comparta recursos del sistema tales como archivos, además de acceder a otros objetos contruidos en el mismo programa

- Ejecución en **paralelo** de procesos o hilos

- Si la arquitectura hardware **realmente** lo permite se mejorarán los tiempos.
- Problemas cuando las “partes” (proceso o hilos) interfieren entre sí.

Introducción

- Diagrama de proceso e hilos
 - N hilos ejecutándose dentro del contexto del proceso



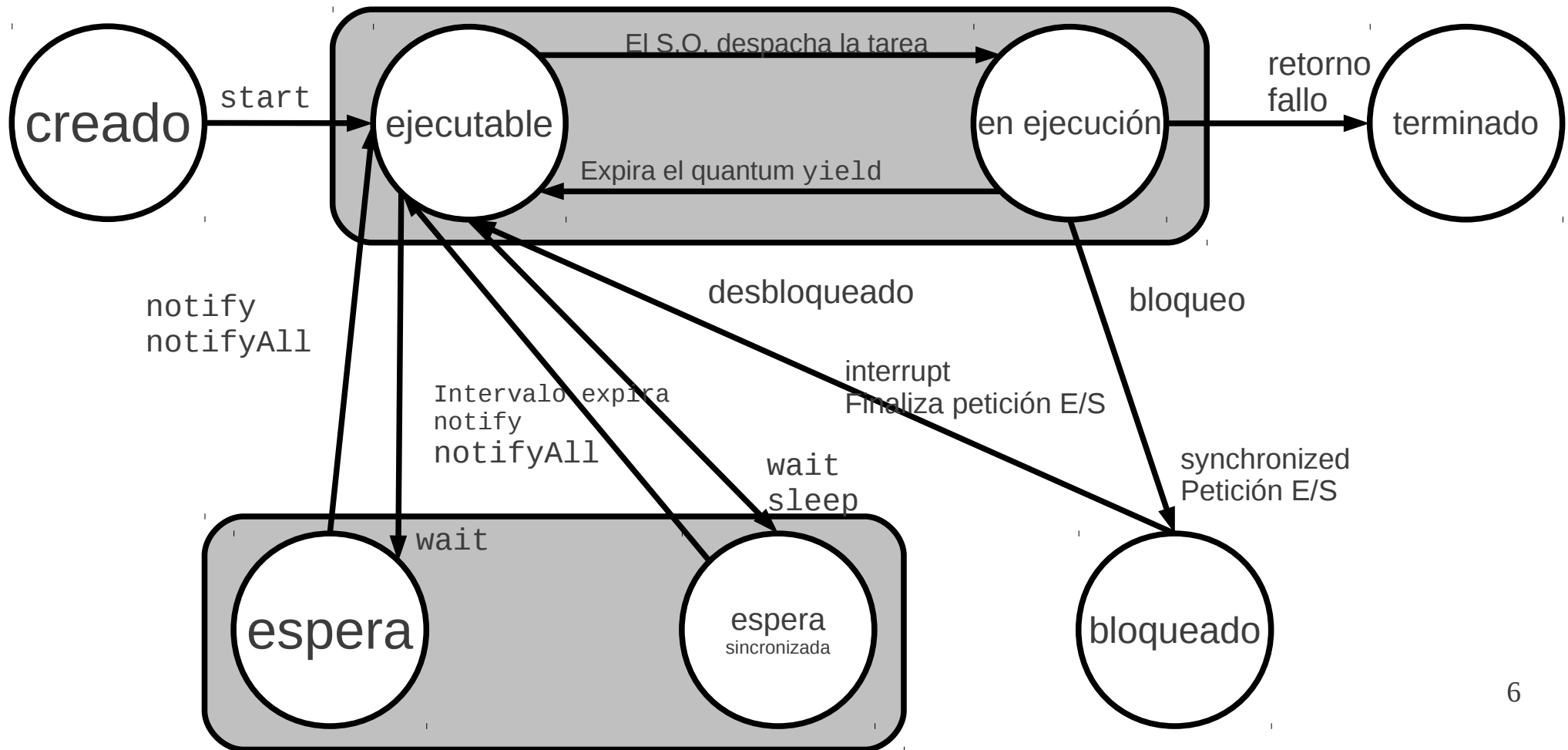
Ejecución concurrente de hilos.
Datos (objetos) compartidos.
Cada hilo mantiene su **propia pila**.

Estructuras básicas de soporte a la concurrencia en Java

- A un lenguaje secuencial se añade el soporte de hebras:
“procesos ligeros”, hilos o threads
- **No crea** procesos paralelos (aunque también se puede) sino **tareas** dentro del mismo proceso
 - Menor costo que crear y destruir **procesos**.
 - Todo proceso tiene como mínimo uno, el **hilo principal**.
- Transparente al sistema operativo con planificación **no determinista**.
- Problemas asociados a la **compartición** de memoria y ficheros.
- A partir de Java 5 con **soporte adicional de bibliotecas**.

Estructuras básicas de soporte a la concurrencia en Java: Ciclo de vida de un hilo

- Diagrama de estados y transición en los hilos



Estructuras básicas de soporte a la concurrencia en Java: Threads

- Opciones básicas de implementación de hilos:
 - Extender de `java.lang.Thread`
 - Redefinir el método `run()`
 - Lanzar el hilo con el método `start`
 - Implementar la interface `Runnable`
 - Redefinir el método `run()`
 - Pasar como argumento un objeto que implementa dicha interfaz al crear un hilo (`new Thread`)
 - Solución preferente
 - **Tip:** no hay herencia múltiple de clases en Java (pero sí de interfaces)
 - Ejemplo (con `Runnable`):
 - se lanzan 10 hilos en paralelo cada uno con un identificador. Se muestra un mensaje 1000 veces y se finaliza.

```
public class HilosBásicos {  
  
    public static void main(String[] args) {  
        for (int i = 0; i < 10; i++) {  
            Thread t = new Thread(new Hilo(i));  
            t.start(); // lanzar hilo  
        }  
        System.out.println("Esperando al fin...");  
    }  
}  
  
public class Hilo implements Runnable {  
    int id;  
    int contador = 1000;  
  
    public Hilo(int id) {  
        this.id = id;  
    }  
  
    public void run() {  
        for (; contador > 0; contador--) {  
            System.out.println("Soy el hilo: " +  
id);  
  
            Thread.yield();  
            // ceder paso, no determinista  
        }  
    }  
}
```

Estructuras básicas de soporte a la concurrencia en Java: Prioridades

- Prioridades de los hilos para su planificación:
 - MIN_PRIORITY a MAX_PRIORITY = [0..10]
 - Valor por defecto NORM_PRIORITY = 5
 - Método `setPriority` para modificar la prioridad
 - Cada hilo hereda la prioridad del hilo que lo crea
 - **No garantizan** un orden de ejecución
 - Comportamiento variable **en función de la plataforma**
 - La programación **SÍ** es independiente de la plataforma
 - El comportamiento del programa **NO**
 - **Tip:** editar, compilar y ejecutar los ejemplos varias veces y en distintas plataformas para observar esto

Estructuras básicas de soporte a la concurrencia en Java: Dormir hilos

- Los hilos se pueden “dormir” o parar por un determinado tiempo
 - Suspende el hilo actual.
- Ejemplo:
 - Dormir tareas usando TimeUnit

```
import java.util.concurrent.*; // para importar TimeUnit

public class HilosBásicos {

    public static void main(String[] args) {
        for (int i = 0; i < 10; i++) {
            Thread t = new Thread(new Hilo(i));
            t.start();
        }
        System.out.println("Esperando al fin...");
    }
}

class Hilo implements Runnable {
    int id;
    int contador = 1000;
    public Hilo(int id) {
        this.id = id;
    }

    public void run() {
        try{
            for (; contador > 0; contador--) {
                System.out.println("Soy el hilo: " + id);
                // Dormir al hilo 1 segundo
                // Thread.sleep(1000); Estilo previo a Java 5
                TimeUnit.MILLISECONDS.sleep(1000); // solución
            }
        } catch (InterruptedException ex) {
            System.err.println("Interrumpido");
        }
    }
}
```

Estructuras básicas de soporte a la concurrencia en Java: Esperar a hilos

- El método de `Thread.join()` permite un hilo esperar a la finalización de otro.
 - `t.join();`
 - Suspende el hilo actual, si `t` es un objeto `Thread` cuyo hilo se está ejecutando actualmente
- Ejemplo:
 - El hilo principal espera a escribir el mensaje hasta que no haya terminado `t`

```
public class HilosBásicos {  
    public static void main(String[] args) throws InterruptedException  
    {  
        Thread t = new Thread(new Hilo(1));  
        t.start();  
        t.join();  
        System.out.println("Programa terminado...");  
    }  
}
```

Estructuras básicas de soporte a la concurrencia en Java : Interrupciones

- Una **interrupción** es una indicación a un hilo que debería dejar de hacer lo que está haciendo y hacer otra cosa
 - Un hilo envía una interrupción invocando a `Thread.interrupt()` del hilo que se quiere interrumpir
- Es responsabilidad del programador decidir exactamente cómo un hilo responde a una interrupción
- ¿Cómo soporta un hilo las interrupciones?
 - Depende de lo que esté haciendo
 - Si está en métodos que lanzan `InterruptedException`
 - Si no consultar `Thread.interrupted()`

Estructuras básicas de soporte a la concurrencia en Java: Ejecutores

- A partir de Java 5 la gestión de hilos se puede delegar
- Los ejecutores crean y administran la ejecución de un objeto `Runnable`
- Ventajas
 - Reutilizar hilos existentes para eliminar la sobre carga de creación
 - Optimizar el rendimiento definiendo un número de hilos adecuado al entorno de ejecución en explotación
- Existen una jerarquía de tres niveles de interfaces
 - Cada uno incrementa la funcionalidad del anterior
 - `Executor.execute()` lanza un hilo con un objeto `Runnable`

```
Thread t = new Thread(...);  
t.start();  
//Método alternativo basado en ejecutores  
Executor exec = Executors.newSingleThreadExecutor();  
exec.execute(t);
```

Estructuras básicas de soporte a la concurrencia en Java: Ejecutores

- Existen una jerarquía de tres niveles de interfaces
 - `ExecutorService`: añade características que ayudan a administrar el ciclo de vida, tanto de las tareas individuales y del propio ejecutor

```
ExecutorService exec = Executors.newSingleThreadExecutor();
exec.execute(t);
//Terminación del servicio
exec.shutdown();
```

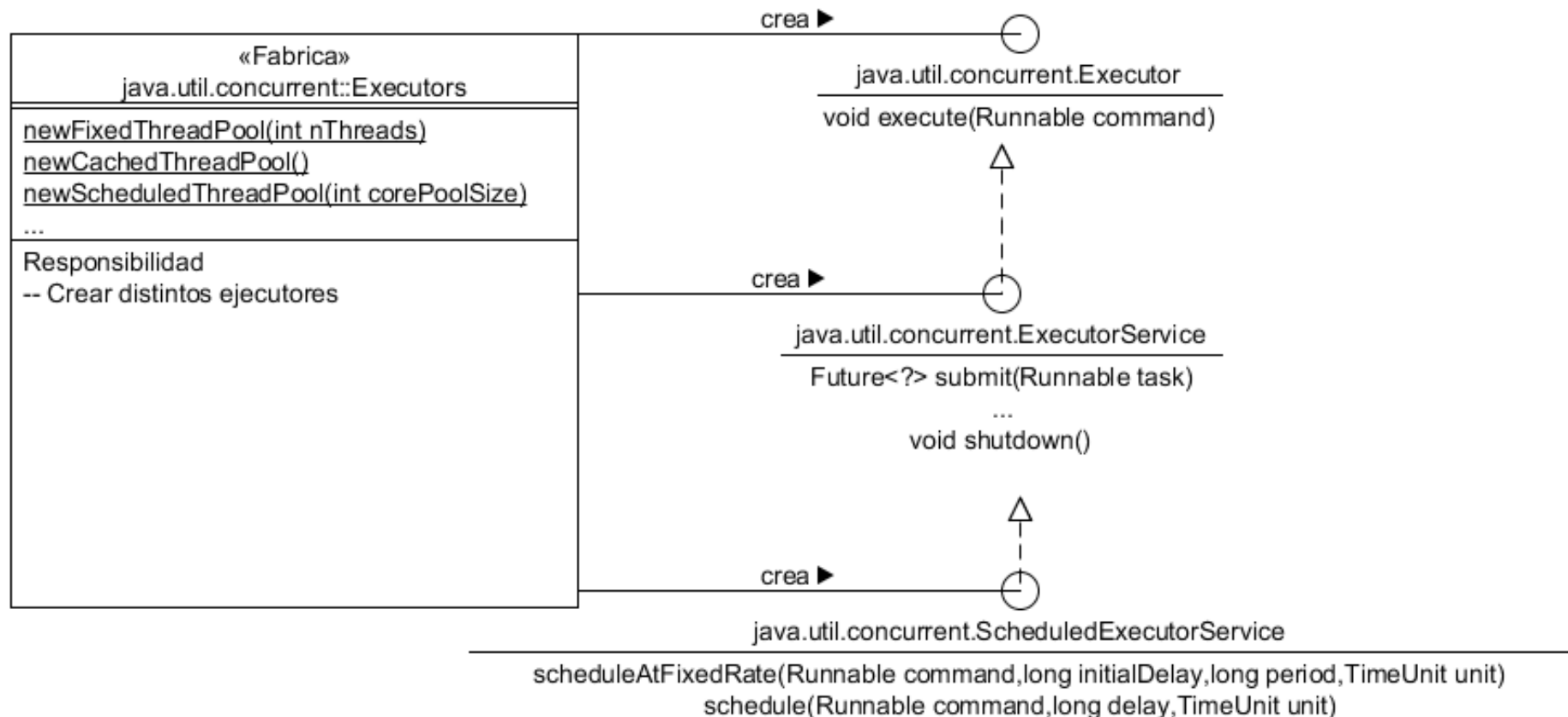
- `ScheduledExecutorService`: permite planificar con un cierto retardo (*delay*) (método `schedule`)
 - Proporciona métodos adicionales
 - `ScheduleAtFixedRate`, `scheduleWithFixedDelay`

```
ScheduledExecutorService exec =
Executors.newScheduledThreadPool(1);
//Planificar una tarea t que se repite cada 5 segundos
exec.scheduleAtFixedRate(t, 0,5, TimeUnit.SECONDS);
exec.execute(t);
```

Estructuras básicas de soporte a la concurrencia en Java: Depósito de hilos

- Minimizando el coste de creación de hilos
 - Aunque son menos costosos que los proceso siempre acarrea un coste adicional
 - Consisten en *working threads* independientes de los objetos `Runnable` o `Callable` (tareas)
 - `newFixedThreadPool`: con un depósito fijo de hilos
 - `newCachedThreadPool`: con un deposito dinámico de hilos. Aconsejable con muchas tareas de corta vida.
 - `newSingleThreadExecutor`: ejecuta una tarea como máximo a la vez

Estructuras básicas de soporte a la concurrencia en Java: Ejecutores y depósitos



Estructuras básicas de soporte a la concurrencia en Java: Ejecutor y depósitos

- Ejemplo de uso:
ExecutorService
Executors

```
import java.util.concurrent.*;

public class HilosEjecutores{

    public static void main(String[] args) {
        ExecutorService exec = Executors.newCachedThreadPool();

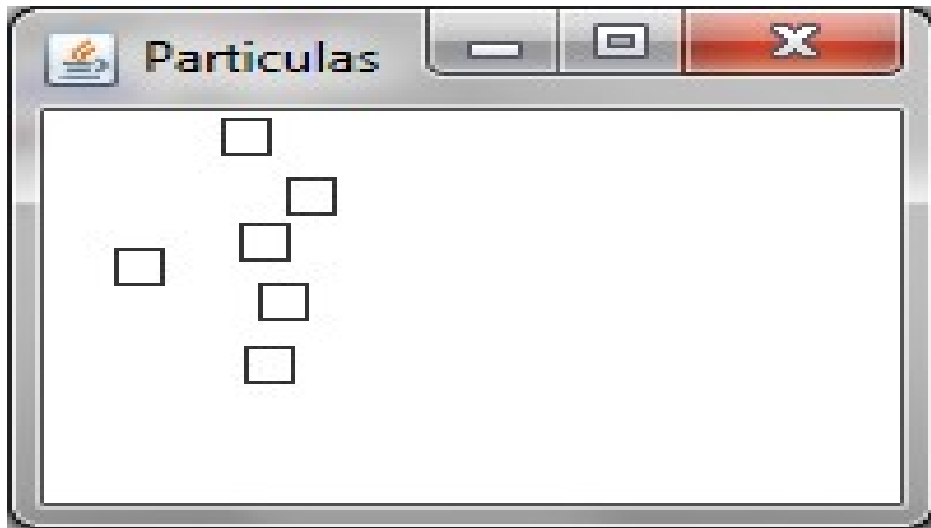
        //ExecutorService exec = Executors.newFixedThreadPool(10);
        //ExecutorService exec =
        Executors.newSingleThreadExecutor();
        for (int i = 0; i < 10; i++) {
            exec.execute(new HiloB(i));
        }
        exec.shutdown();
        System.out.println("Esperando al fin...");
    }

    class HiloB implements Runnable {
        int id;
        int contador = 1000;
        public HiloB(int id) {
            this.id = id;
        }

        public void run() {
            for (; contador > 0; contador--) {
                System.out.println("Soy el hilo: " + id);
                Thread.yield();
            }
        }
    }
}
```


Estructuras básicas de soporte a la concurrencia en Java: Ejemplo

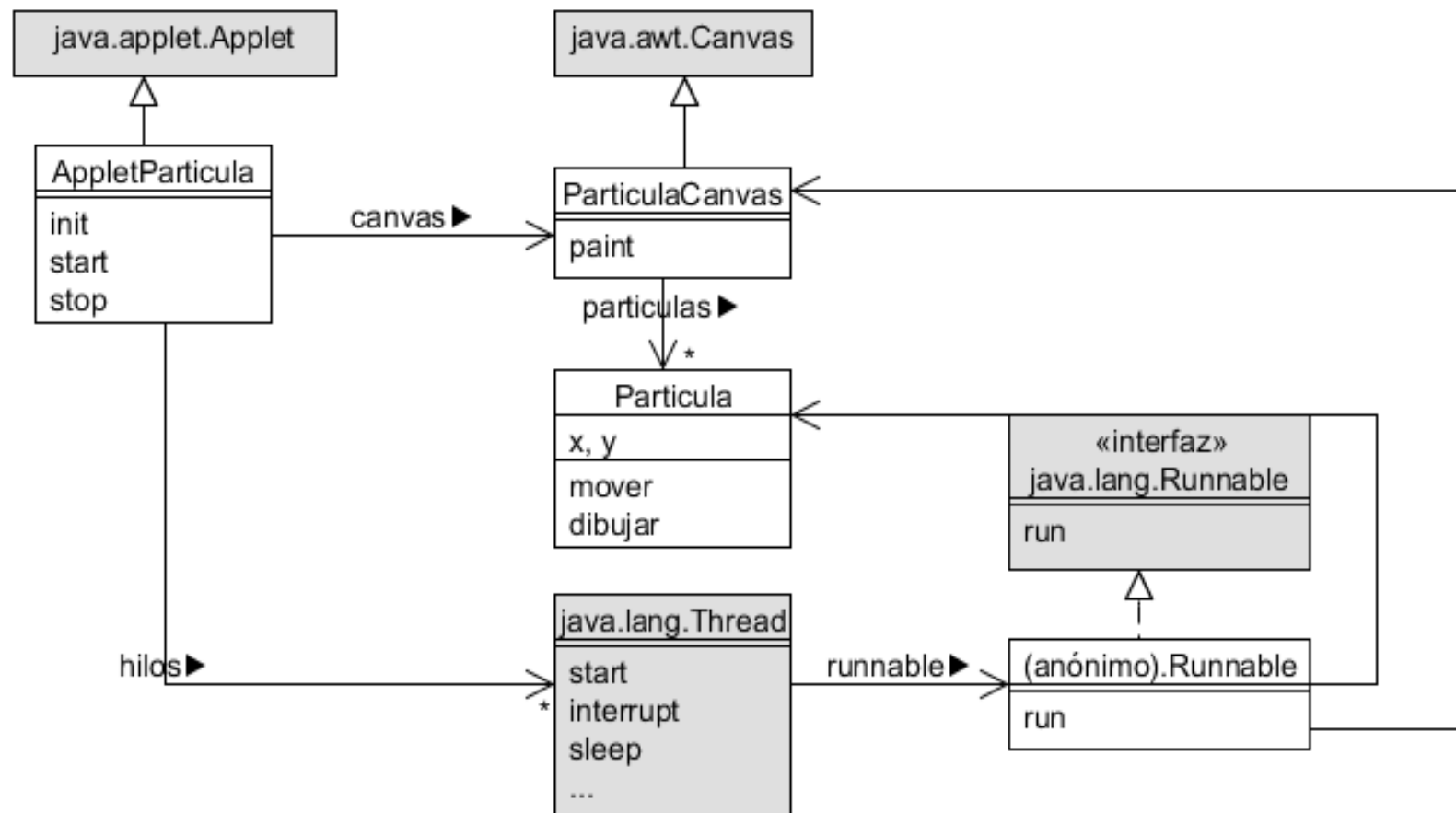
- Aplicación partículas
 - Visualiza partículas moviéndose aleatoriamente, de manera continua y autónoma
 - Una partícula define un modelo no realista de cuerpos móviles
 - Se representa por su posición (x,y)
 - Puede cambiar aleatoriamente su posición
 - Puede dibujarse como un cuadrado pequeño



```
public class Particle {  
    protected int x;  
    protected int y;  
    protected final Random rng = new Random();  
  
    public Particle(int initialX, int initialY) {  
        x = initialX;  
        y = initialY;  
    }  
  
    public synchronized void move() {  
        x += rng.nextInt(10) - 5;  
        y += rng.nextInt(20) - 10;  
    }  
  
    public void draw(Graphics g) {  
        int lx, ly;  
        synchronized (this) { lx = x; ly = y; }  
        g.drawRect(lx, ly, 10, 10);  
    }  
}
```

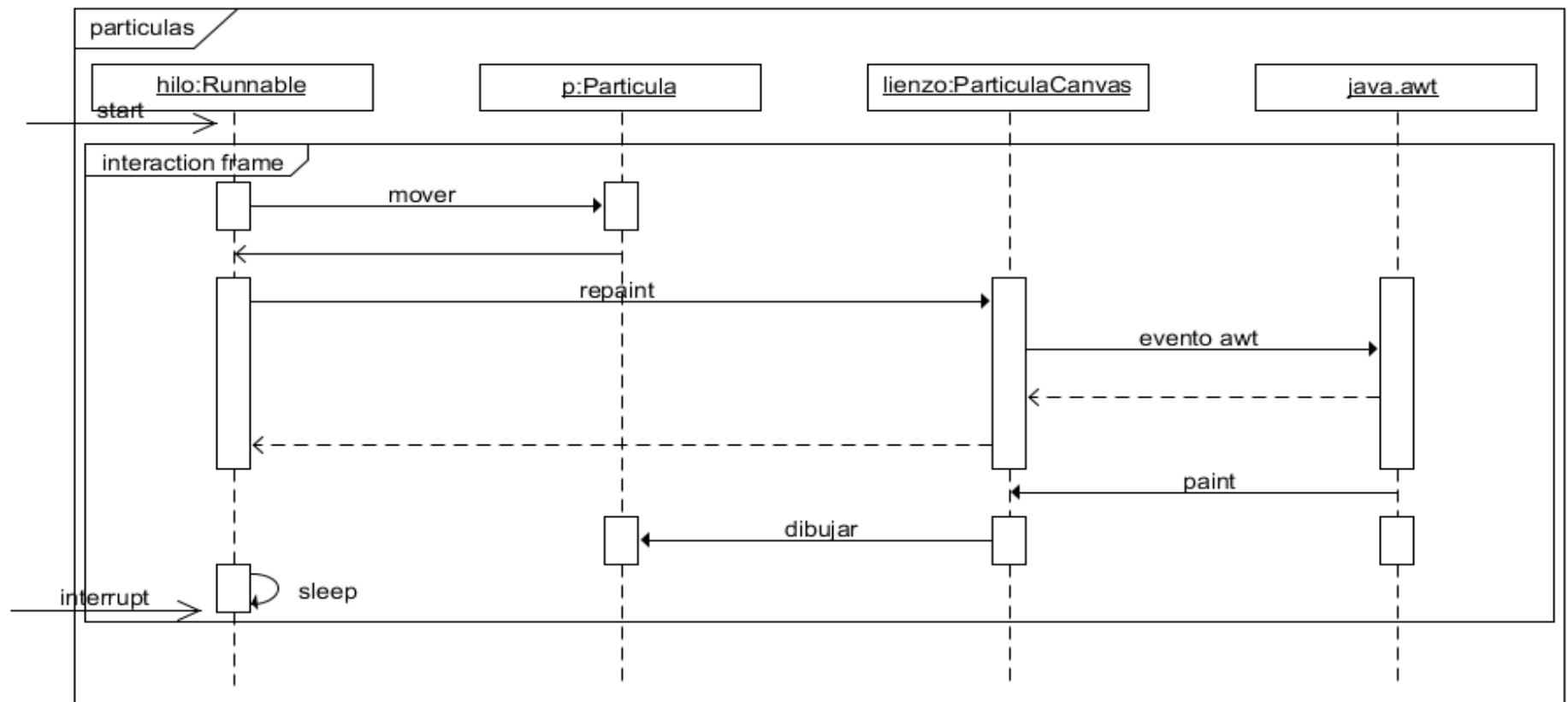
Estructuras básicas de soporte a la concurrencia en Java: Ejemplo

- Aplicación partículas



Estructuras básicas de soporte a la concurrencia en Java: Ejemplo

- Aplicación partículas
 - Movimiento de las partículas
 - Asociar un bucle independiente a cada partícula



Estructuras básicas de soporte a la concurrencia en Java: Ejemplo

- Asociar un bucle independiente a cada partícula

```
protected Thread[] threads;
protected final ParticleCanvas canvas;
...
// Método utilidad
protected Thread makeThread(final Particle p) {
    Runnable runloop = new Runnable() {
        public void run() {
            try {
                for(;;) {
                    p.move();
                    canvas.repaint();
                    Thread.sleep(100); // 100msec arbitrario
                }
            }
            catch (InterruptedException e) { return; }
        }
    };
    return new Thread(runloop);
}
```

Estructuras básicas de soporte a la concurrencia en Java: Ejemplo

- Asociar un bucle independiente a cada partícula

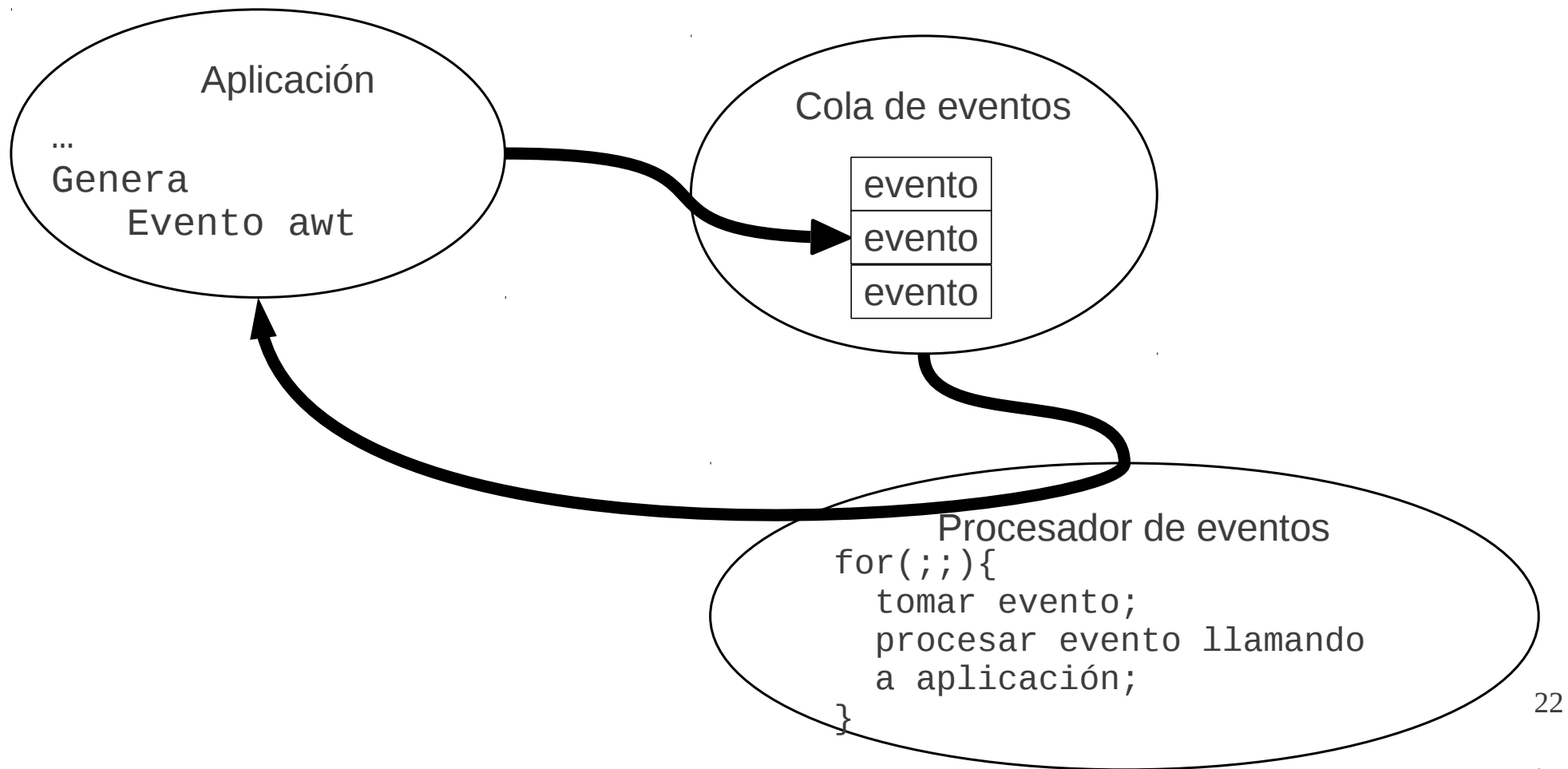
```
...
    public synchronized void start() {
        int n = 10;
        if (threads == null) {
            Particle[] particles = new Particle[n];
            for (int i = 0; i < n; ++i)
                particles[i] = new Particle(50, 50);
            canvas.setParticles(particles);
            threads = new Thread[n];
            for (int i = 0; i < n; ++i) {
                threads[i] = makeThread(particles[i]);
                threads[i].start();
            }
        }
    }

    public synchronized void stop() {
        if (threads != null) {
            for (int i = 0; i < threads.length; ++i)
                threads[i].interrupt();
            threads = null;
        }
    }

    public void finalize(){
        stop();
    }
}
```

Estructuras básicas de soporte a la concurrencia en Java: Ejemplo

- Interacción con el hilo de eventos de `java.awt`
 - Dos actividades: generar eventos y procesar eventos



Estructuras básicas de soporte a la concurrencia en Java: Ejemplo

- Partículas – Monitor de su ejecución



Objetos y concurrencia

Concurrencia

- Un programa concurrente
 - Es un programa que hace más de una cosa a la vez
 - Ejemplo: Navegador Web
 - Petición http para acceder a página html + ejecutar audio +
- La maquina virtual Java (MVJ) y el sistema operativo subyacente hacen posible la simultaneidad aparente
 - Paralelismo físico o tiempo compartido
- Programación concurrente (PRGC) vs. Programación distribuida (PRGD)
 - PRGC en Java se restringen a estructuras que afectan a una sola JVM
 - PRGD múltiples JVM en distintos sistemas informáticos

Objetos y concurrencia

Concurrencia

- Aplicaciones concurrentes y su motivación
 - Servicios Web
 - Demonios http, servidores de aplicaciones
 - Soportar múltiples conexiones concurrentes para no esperar la terminación de una conexión para empezar otra
 - Mejorar tiempos de latencia del servicio
 - Cálculo numérico
 - Tareas intensivas de cálculo
 - Maximizar rendimiento haciendo uso de paralelismo real de CPU
 - Procesamiento de entrada-salida
 - Dispositivos de E/S independientes CPU (paralelismo)
 - Uso eficiente de los recursos

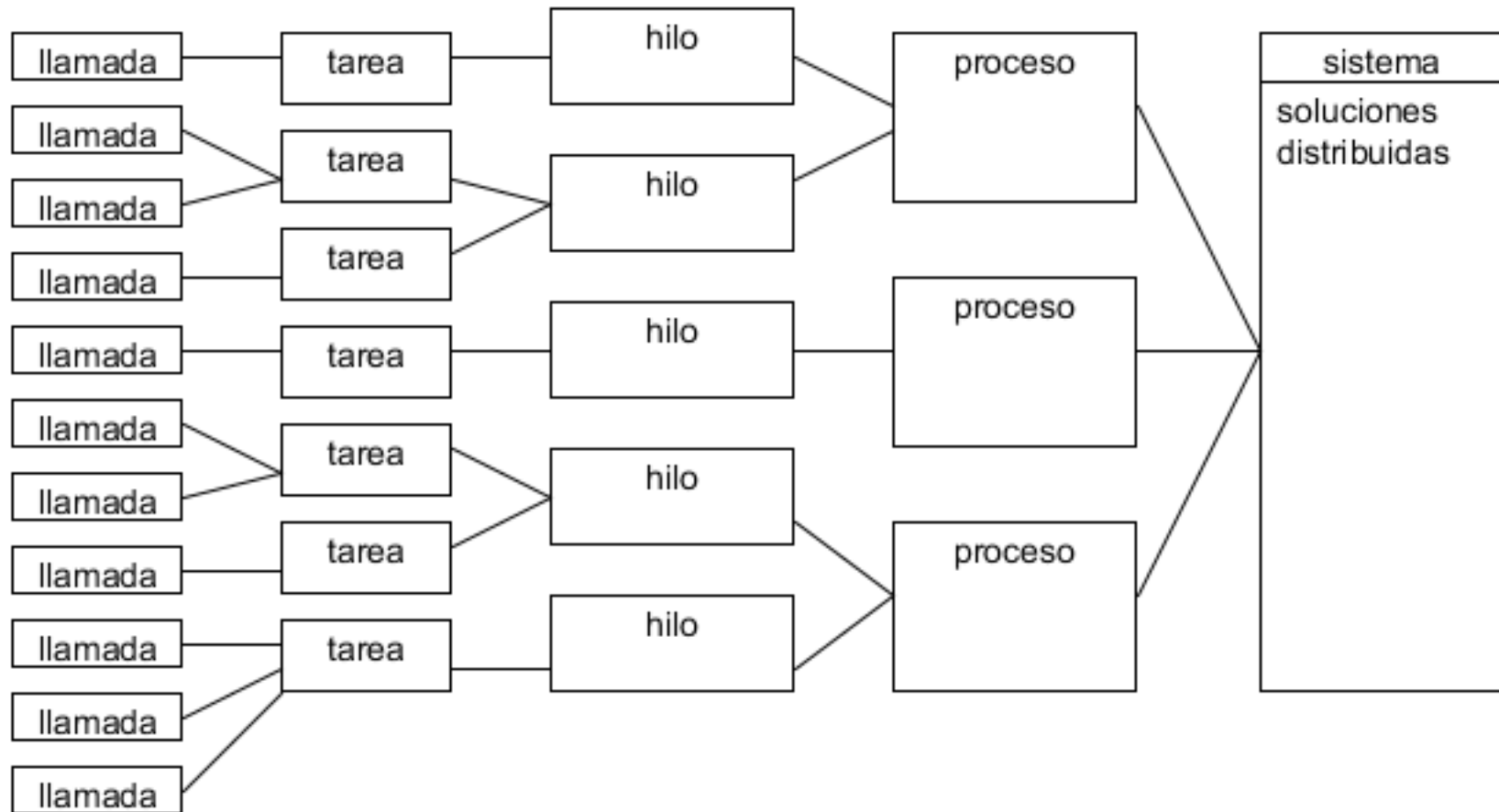
Objetos y concurrencia

Concurrencia

- Aplicaciones concurrentes y motivos
 - Simulación
 - Objetos físicos con comportamientos autónomos independientes
 - Aplicaciones basadas en GUI
 - Permite el uso de componentes gráficos incluso con acciones disparadas con eventos gráficos que consuman mucho tiempo
 - Software basado en componentes
 - Componente editor multimedia integrado en aplicación
 - Mejorar autonomía y funcionamiento
 - Código móvil
 - `Java.applet` descarga de código mediante hilos separados
 - Aislar y controlar los efectos de código desconocido
 - Sistemas empujados
 - Pequeños dispositivos que realizan tareas en tiempo real

Objetos y concurrencia

Estructuras para la ejecución concurrente



Objetos y concurrencia

Estructuras para la ejecución concurrente

- Características de las estructuras
 - Autonomía e independencia
 - Administración de la estructura
 - Planificación
 - Creación, gestión y comunicación
 - Capacidad para compartir recursos subyacentes
 - CPU's, memoria, canales E/S
 - Comunicación entre las mismas estructuras
 - Canales (sockets) – paso de mensajes
 - Áreas de memoria compartidas
 - Mecanismos de sincronización basados en memoria
 - Cerrojos – mecanismos de espera y notificación

Objetos y concurrencia

Concurrencia y programación OO

- Historia de los lenguajes de programación concurrente
 - Simula (1966) → C, C ++ → Ada
- Diferencias en programación secuencial OO
 - Ejecuciones no deterministas
 - Comprensión de código no es secuencial
 - Las **interferencias entre las actividades** necesitan diseño conservador
 - Sentencia de asignación depende de actividades
- Programación basada en eventos
 - Múltiples diseños
 - Múltiples hilos de bucles de eventos
 - Procesando concurrentemente cada uno de los eventos
 - Efectos secundarios de interferencias y coordinación entre actividades concurrente

Objetos y concurrencia

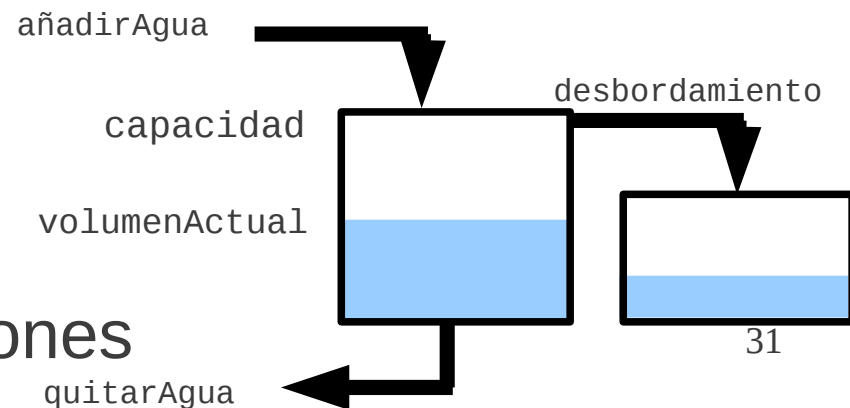
Concurrencia y programación OO

- Programación de sistemas concurrentes
 - Diferencia de programación multihilo de C
 - Incluye construcciones que permiten
 - Encapsulación
 - Modularidad
 - Extensibilidad
 - Seguridad
 - En Java el soporte a la concurrencia está en el propio lenguaje
 - Similar a las bibliotecas de los pthreads POSIX
- Otros lenguajes de programación concurrente
 - Cada uno define sus propias características de concurrencia
 - Se puede simular características de concurrencia a través de bibliotecas y convenciones de codificación
 - Semáforos, barreras...

Objetos y concurrencia

Transformaciones y modelos de objetos

- Objetos software vs. objetos reales
 - Sólo se consideran las características en el campo de la computación
 - Atributos
 - Restricciones de estado invariantes
 - Operaciones
 - Conexiones con otros objetos
 - Precondiciones y postcondiciones sobre las operaciones
 - Protocolos
 - Relacionados con tiempo
 - Relacionado con operaciones



Objetos y concurrencia

Transformaciones y modelos de objetos

- Modelos de Objetos – TanqueAgua

- Estructura estática – atributos y métodos

- Encapsulación

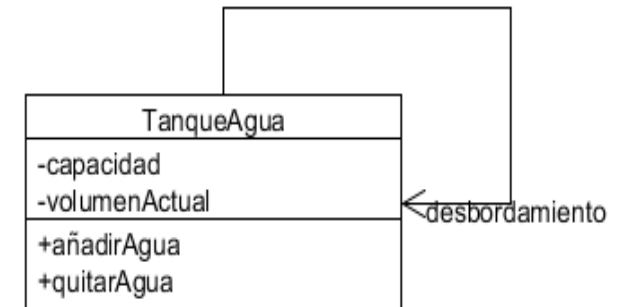
- Comunicación – paso de mensajes

- Identidad

- Conexiones – canal que permite el paso de mensajes

- Acciones

- Aceptar un mensaje
- Actualizar el estado
- Enviar un mensaje
- Crear un objeto



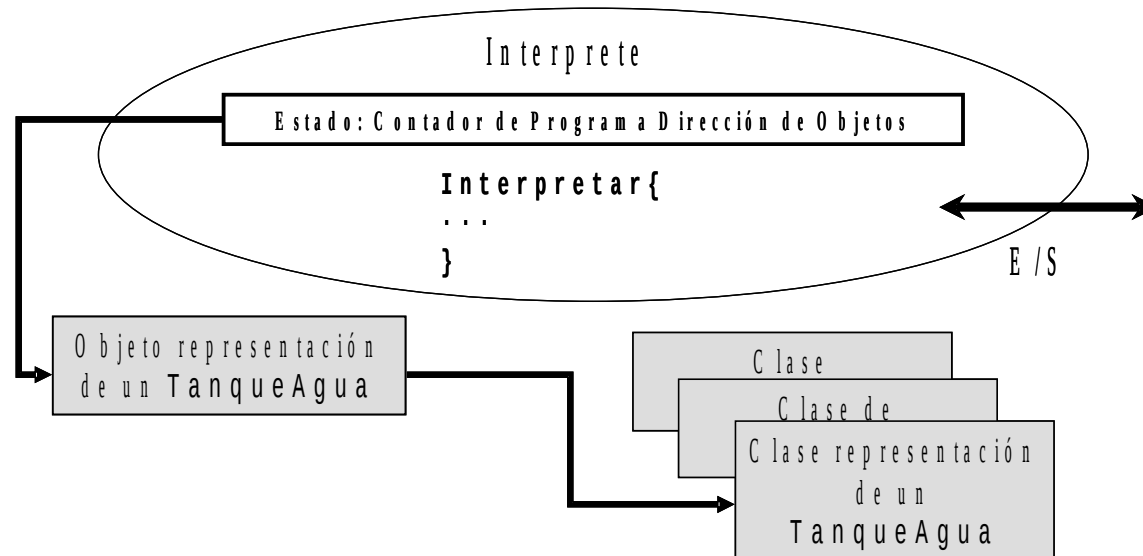
```
public class TanqueAgua {  
    private final float CAPACIDAD;  
    private float volumenActual;  
    private TanqueAgua desbordamiento;  
  
    public TanqueAgua(float cap){  
        CAPACIDAD = cap;  
        volumenActual = 0.0f;  
    }  
    public void anadirAgua(float cantidad)  
        throws ExcepcionDesbordamiento {...}  
  
    public void quitarAgua(float cantidad)  
        throws ExcepcionVacía {...}  
}
```


Objetos y concurrencia

Transformaciones y modelos de objetos

- Transformaciones secuenciales
 - El computador de propósito general (CPU, bus, memoria, E/S) aparenta ser cualquier un objeto
 - Carga el descriptor de la clase `.class`
 - Construir representación pasiva de una instancia
 - Interpretar las operaciones
 - JVM es en sí misma un objeto que finge ser cualquier objeto

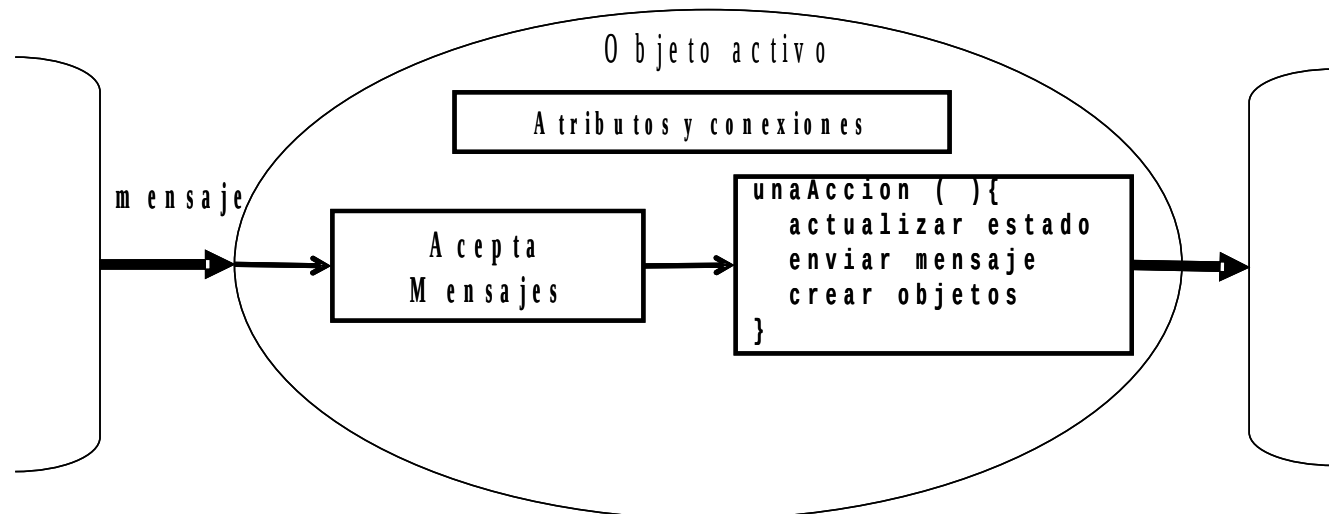
JVM Secuencial



Objetos y concurrencia

Transformaciones y modelos de objetos

- Modelo de objetos activos o modelo de actores
 - Cada objeto es autónomo
 - Cada uno es tan potente como una JVM secuencial
 - Una clase y la representación del objeto pueden ser la misma que la usada en entornos pasivos
 - Sistemas distribuidos – problemas de localización y dominio de administración del objeto
 - Mensajes mediante comunicación remota
 - Sockets



Objetos y concurrencia

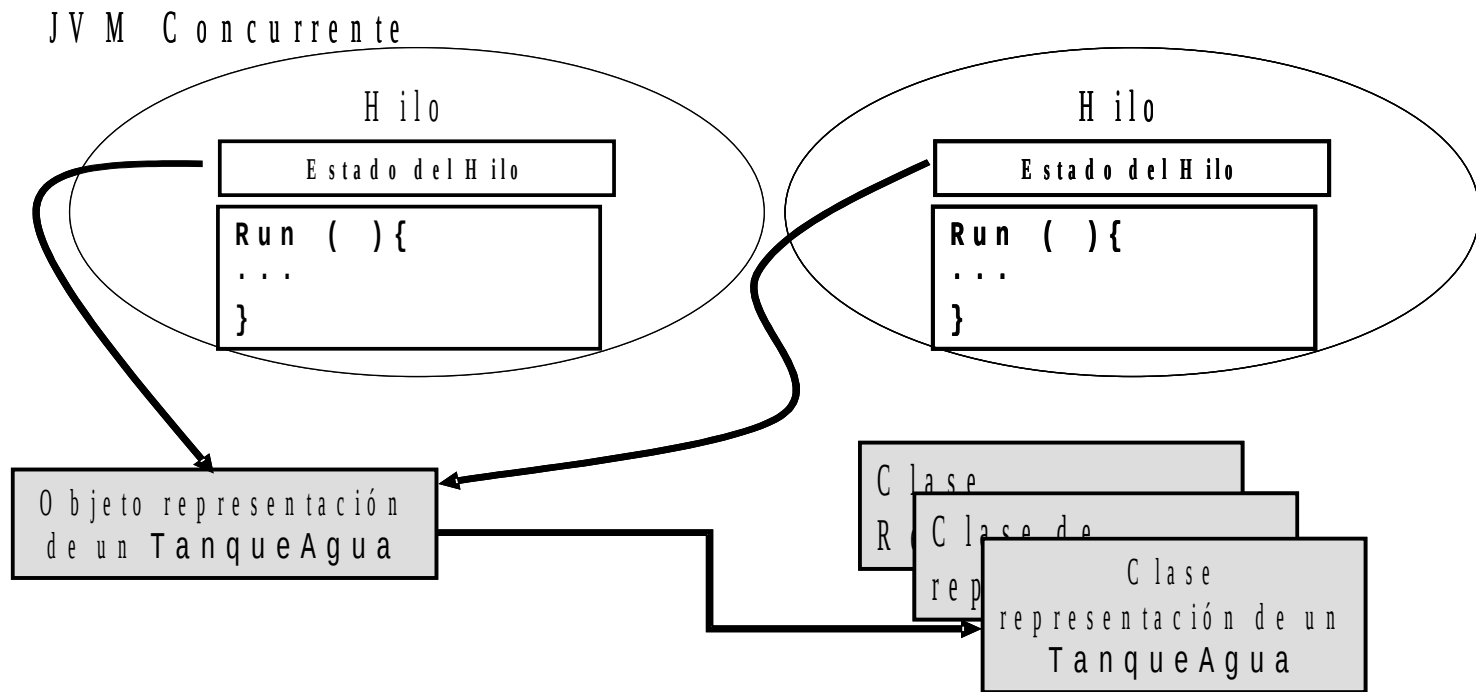
Transformaciones y modelos de objetos

- Modelos mixtos
 - Una JVM puede estar compuesta de múltiples hilos
 - Los hilos comparten las representaciones subyacentes
 - Una clase y la representación del objeto pueden ser la misma que la usada en entornos pasivos
 - Se separan **objetos** normales/**pasivos** de los **activos**/hilos
 - Protección de objetos pasivos de coincidencia de hilos mediante cerrojos
 - Objetos activos más simples sólo soportan una operación `run()`
 - Direcciones del diseño concurrente
 1. Que los objetos pasivos vivan en un contexto multihilo
 2. “silenciando” objetos activos para que puedan ser expresados en estructuras de hilo

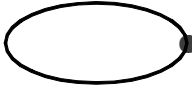

Objetos y concurrencia

Transformaciones y modelos de objetos

- Modelos mixtos
 - El grado de control del desarrollador sobre estas transformaciones es un rasgo de distinción entre programación concurrente y paralela
 - La programación concurrente deja la mayoría de las transformaciones a la JVM y al SO subyacente



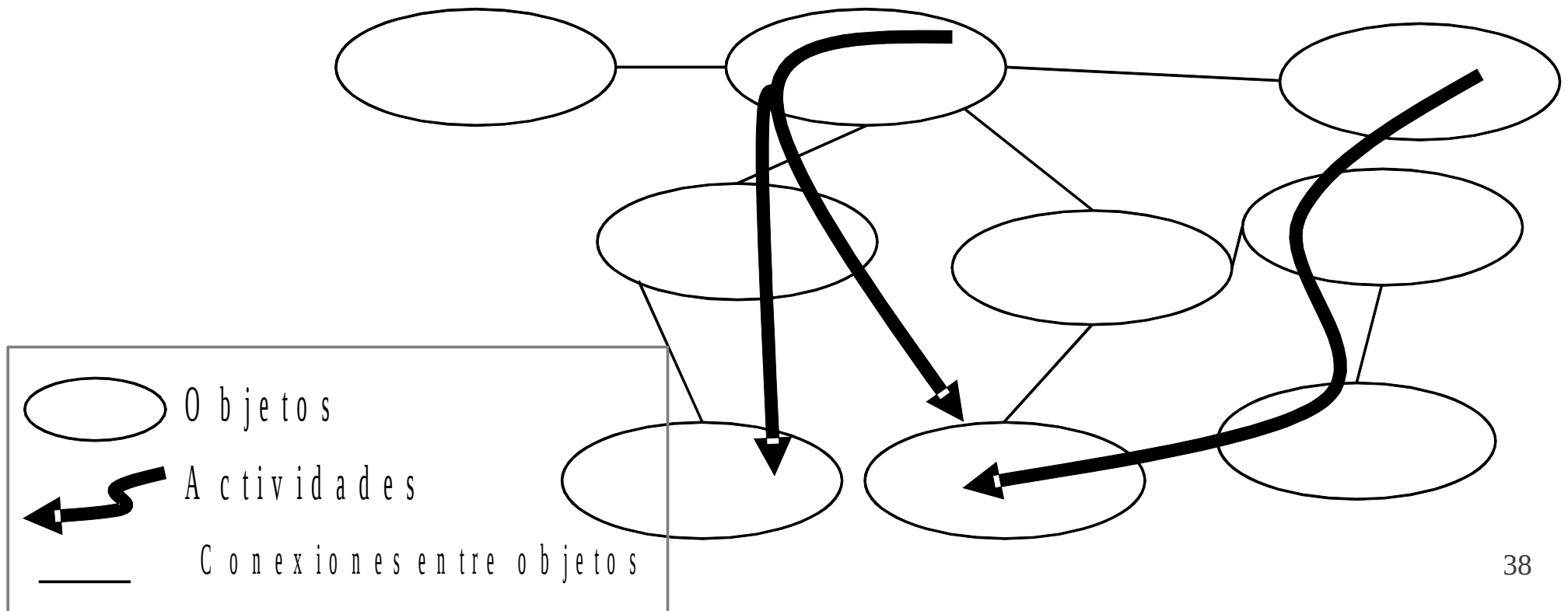
Imposiciones de diseño

- Programación secuencial vs. Programación concurrente
 - Aparecen nuevos problemas de diseño
- Dos perspectivas complementarias de un sistema orientado a objetos
 -  Centrado en objetos – conjunto de objetos
 -  Centrado en la actividad – conjunto de actividades
 - Mensajes, cadenas de llamadas, secuencia eventos, tareas, sesiones...

Sistemas = Objetos + Actividades

Imposiciones de diseño

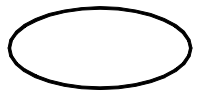
- Las dos perspectiva son complementarias
 - Un objeto se puede implicar en varias actividades
 - Una actividad puede atravesar múltiples objetos



Imposiciones de diseño

- Las estrategias de diseño son complementarias

- Diseño respecto a la **corrección**



- **Seguridad** – nunca sucede nada malo a un objeto



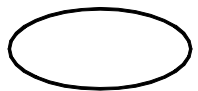
- **Vivacidad** – algo sucede eventualmente dentro de una actividad

- La elección de la estrategia complementaria depende del balance de efectos de fallos

- Por defecto se prima la seguridad

- Ejemplo para primar vivacidad respecto a seguridad barra de progreso

- Diseño respecto a la **calidad**



- **Reutilización** – utilización de objetos y de clases en múltiples contextos



- **Rendimiento** – grado en que las actividades se ejecutan pronto y rápidamente

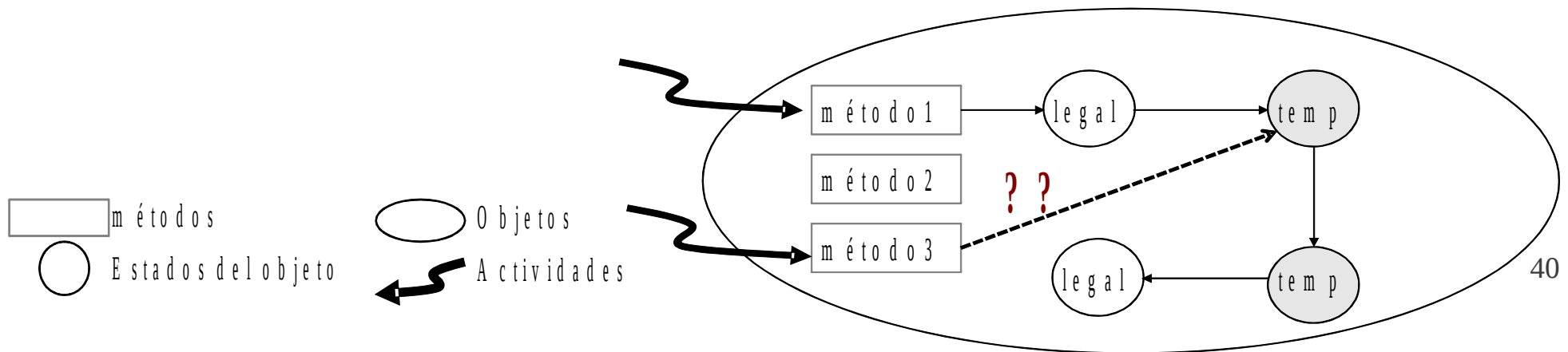
Imposiciones de diseño

Seguridad

- A la seguridad en programación concurrente se agrega una **dimensión temporal**
- Objetivo en la preservación de la seguridad

Asegurar que todos los objetos del sistema mantienen **estados coherentes**

Estado coherente – todos los campos del objeto y todos los campos de los otros objetos de los cuales depende, poseen **valores legales** y significativos

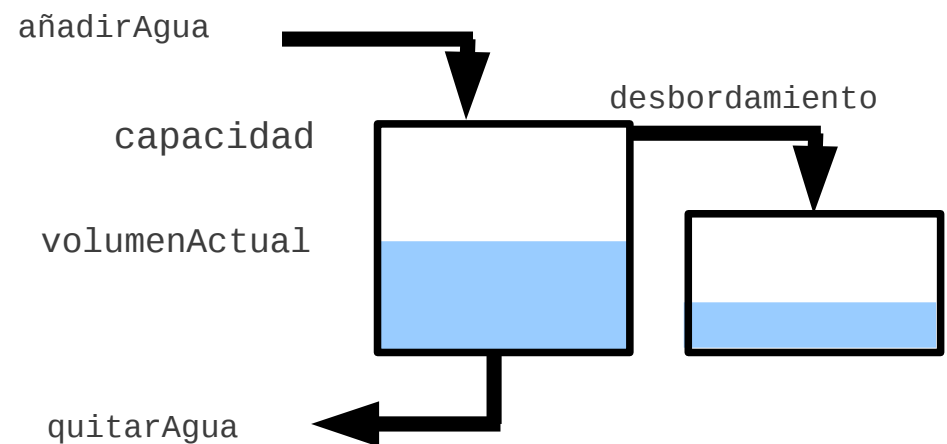


Imposiciones de diseño

Seguridad

- “**Valores legales**” implica cumplir invariantes a nivel conceptual
 - $\text{VolumenActual} \geq 0 \text{ AND } \text{VolumenActual} \leq \text{Capacidad}$
- Un método público cambia el estado del objeto
 - añadirAgua, quitarAgua

En programación se necesitan técnicas de protección para **garantizar valores legales** (cumplir los invariantes)



Imposiciones de diseño

Seguridad

Técnicas de protección para **garantizar valores legales** (cumplir los invariantes)

1. Procesamiento de excepciones (prg. secuencial)



2. **Técnicas de exclusión** para garantizar la atomicidad de las acciones públicas (prg. concurrente)



- Cada acción se ejecuta hasta la terminación sin interferencia de otras
- Nuevos problemas por **condiciones de competencia** producen **conflictos de almacenamiento** a bajo nivel
 - Lectura/Escritura
 - Un hilo lee el valor de un campo mientras otro escribe en el mismo, es difícil prever el valor leído
 - Escritura/Escritura
 - Dos hilos intentan escribir en el mismo campo es difícil conocer el valor de la siguiente lectura

Imposiciones de diseño

Seguridad

- Comprensión de atributos y restricciones conceptuales
 - Una CuentaBancaria tiene un saldo que es igual a la suma de todos los depósitos e intereses menos los reintegros y gastos de servicios
 - Un Paquete tiene un destino que tiene que ser una dirección IP
 - Un Contador tiene un valor entero no negativo
 - Un Termostato tiene una temperatura igual a la lectura más reciente del sensor
 - Una Figura tiene una posición, dimensión y color que obedece a una serie de pautas de estilo correspondientes a un juego de herramientas GUI
 - Un BufferLimitado siempre tiene un contador de elementos entre cero y una capacidad

Imposiciones de diseño

Seguridad

- Comprensión sobre las restricciones de representación
 - Pueden aparecer nuevos invariantes adicionales por decisiones de implementación
 - Categorías de campos y restricciones
 - Directas del valor: `Buffer indiceInsertar`
 - De valores derivados `CuentaBancaria saldodeudor`
 - Representaciones lógicas de estado `LectorTarjeta pinValido`, `JuegoTablero turno`
 - Variables sobre el estado de ejecución. Enumeración `CONECTANDO`, `ACTUALIZANDO`, `ESPERANDO`
 - Variables históricas `CuentaBancaria ultimoSaldoRegistrado`
 - Variables para el seguimiento `Termostato númeroLectura`
 - Referencias a relaciones `ManejadorPeticiones WebServer`
 - Referencias a relaciones de objetos
 - `CuentaBancaria` referencia a un campo de tipo `Deposito`

Imposiciones de diseño

Vivacidad

- Los problemas de seguridad se tienen que equilibrar con problemas de vivacidad
- En **sistemas vivaces**, cada actividad progresa hacia la terminación
 - De forma transitoria, una actividad puede no progresar
 - Bloqueo `synchronized` un hilo adquiere un cerrojo
 - Espera `Object.wait`
 - Entrada por un método de E/S
 - Competencia de CPU
 - Fallo

El ciclo de vida de un hilo puede incluir un número de bloqueos transitorios

Imposiciones de diseño

Vivacidad

- Problemas de vivacidad o de bloqueo permanente
 - Interbloqueo
 - Dependencias circulares cerrojos
 - Señales perdidas
 - Un hilo empieza a esperar después de la notificación para que despierte
 - Cerrojos anidados en un monitor
 - Un hilo que espera para adquirir un cerrojo que podría ser necesitado por cualquier otro hilo que trata de despertarlo

Imposiciones de diseño

Vivacidad

- Problemas de vivacidad o de bloqueo permanente
 - Falta de vivacidad
 - Una acción repetida falla continuamente
 - Inanición
 - JVM no puede asignar CPU
 - Falta de recursos
 - Los hilos tienen un límite de recursos
 - Fallo distribuido
 - No accesible máquina remota conectada mediante socket

Imposiciones de diseño

Rendimiento

- Las imposiciones de rendimiento amplían los problemas de vivacidad
 - Además de exigir que cada método se ejecute, los objetivos del rendimiento es que se ejecute pronto y rápido
- Medidas de rendimiento
 - Productividad, latencia, capacidad, eficiencia, escalabilidad, degradación
- En los diseños concurrentes multihilo se empeora la **eficiencia** para mejorar la **latencia**

Imposiciones de diseño

Rendimiento

- El soporte de concurrencia introduce **sobrecarga**
 - **Cerrojos** mayor sobrecarga métodos `synchronized`
 - **Monitores** `Object.wait`, `Object.notify`, `Object.notifyAll` pueden ser más costosos que otras operaciones JVM
 - **Cambio de contexto** la asignación de hilos a CPU
 - **Planificación** cálculos y políticas de asignación de hilos
 - **Localización** en sistemas multiprocesadores múltiples hilos que comparten acceso a objetos
 - **Algoritmos** secuenciales eficientes no sirven para contextos concurrentes

Minimizar el uso de estructuras concurrentes para adecuarse a las medidas de rendimiento

Imposiciones de diseño

Reutilización

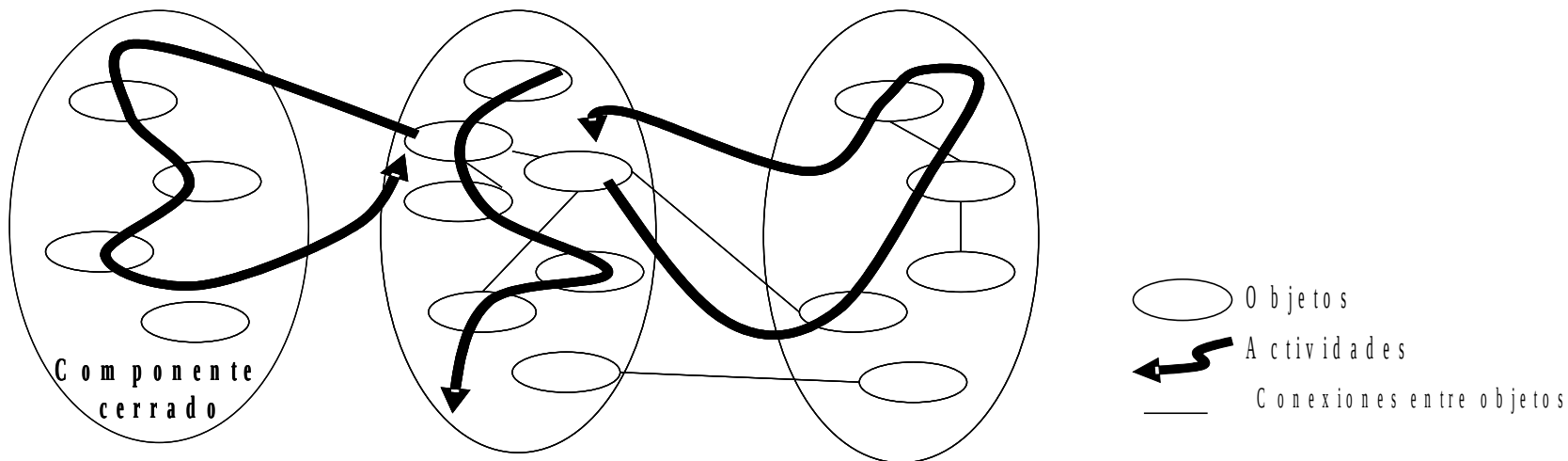
- Una clase o un objeto es reutilizable cuando puede ser utilizado fácilmente en distintos **contextos**
 - Componentes de caja negra
 - Extensión de caja blanca por medio de herencia
- La dualidad entre seguridad y vivacidad dificulta la reutilización
 - Un componente seguro no puede reutilizarse por no ser vivaz en todos los **contextos**
 - Alternativas de diseño de sistemas concurrentes
 - Basados en seguridad → sistemas lentos
 - Basados en vivacidad → ejecuciones incorrectas

Documentar los contextos de uso de los componentes

Imposiciones de diseño

Reutilización

- Tendencias para abordar la **dependencia del contexto**
 1. Reducir incertidumbre aislando parte del sistema
 - Subsistemas cerrados
 - Comunicación externa restringida
 - Estructura interna/estática determinista: n° objetos + n° hilos
 - Seguros pero frágiles si cambia comunicación
 - Ejemplo sistemas empotrados



Imposiciones de diseño

Reutilización

2.- Establecer las políticas y protocolos que permitan a los componentes seguir siendo accesibles

- Sistemas abiertos
- Imposible determinar un análisis estático puesto que naturaleza y estructura se desarrollan a través del tiempo
- Dominio de políticas
 - **Flujo** los componentes de tipo A envían mensajes a los componentes de tipo B
 - **Bloqueo** los métodos de tipo A siempre lanzan excepciones si el recurso R no está disponible
 - **Notificaciones** los objetos de tipo A siempre envían notificaciones de cambio a sus oyentes siempre que estén actualizados
- Ejemplo internet

Bibliografía

- Libros y manuales

- Doug Lea. Programación concurrente en Java: Principios y patrones de diseño. 2ª ed. PEARSON EDUCACION, 2000.
<http://gee.cs.oswego.edu/dl/cpj/>.
- Oracle. «Lesson: Concurrency (The Java™ Tutorials > Essential Classes)». Accedido enero 23, 2013.
<http://docs.oracle.com/javase/tutorial/essential/concurrency/index.html>.

- Enlaces de interés

- «Java Concurrent Animated | Free Home & Education software downloads at SourceForge.net». Accedido enero 23, 2013.
<http://sourceforge.net/projects/javaconcurrenta/>.