***Imagen que contiene Logotipo

Descripción generada automáticamente***

Alejandro Ortega Martínez

Grado en Ingeniería Informática

23/4/2021

Tema 1

Conceptos

Programación Concurrente

Contenido

[1. Introducción 2](#_Toc70076973)

[2. Estructuras básicas de soporte a la concurrencia en Java 3](#_Toc70076974)

[2.1. Ciclo de vida de un hilo 3](#_Toc70076975)

[2.2. Threads 3](#_Toc70076976)

[2.3. Prioridades 4](#_Toc70076977)

[2.4. Dormir hilos 4](#_Toc70076978)

[2.5. Esperar a hilos 4](#_Toc70076979)

[2.4. Interrupciones 5](#_Toc70076980)

[2.6. Depósito de hilos (pools) 6](#_Toc70076981)

[2.7. Ejecutores y depósitos 6](#_Toc70076982)

[3. Objetos y concurrencia 7](#_Toc70076983)

[3.1. Concurrencia 7](#_Toc70076984)

[3.2. Estructuras para la programación concurrente 7](#_Toc70076985)

[3.3. Concurrencia y programación OO 7](#_Toc70076986)

[3.4. Transformaciones y modelos de objetos 7](#_Toc70076987)

[Transformaciones secuenciales 8](#_Toc70076988)

[Modelo de objetos activos o modelos de actores 8](#_Toc70076989)

[Modelos mixtos 9](#_Toc70076990)

[4. Imposiciones de diseño 10](#_Toc70076991)

[4**.**1. Seguridad 11](#_Toc70076992)

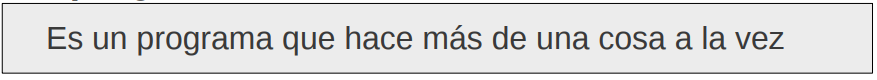
[4.2. Vivacidad 11](#_Toc70076993)

[4.3. Rendimiento 12](#_Toc70076994)

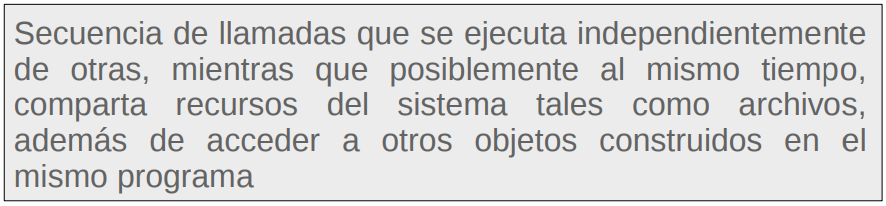
[4.4. Reutilización 12](#_Toc70076995)

# 1. Introducción

Un **programa concurrente:**

****

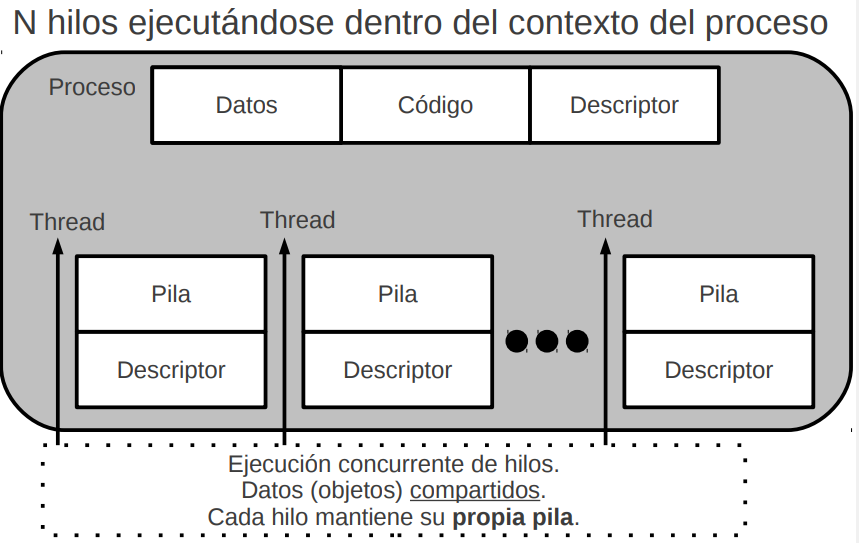
Una definición de **proceso o hilo:**

****

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

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

Diagrama de procesos e hilos



# 2. Estructuras básicas de soporte a la concurrencia en Java

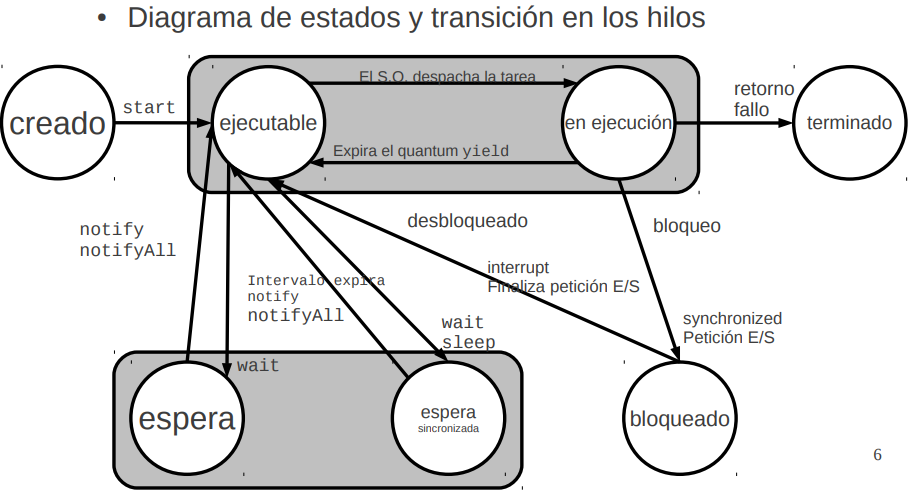
A un lenguaje secuencial se le añade el soporte de hebras: “procesos ligeros”, hilos o threads.

No crea procesos paralelos (aunque también se puede), sino **tareas dentro del mismo proceso**:

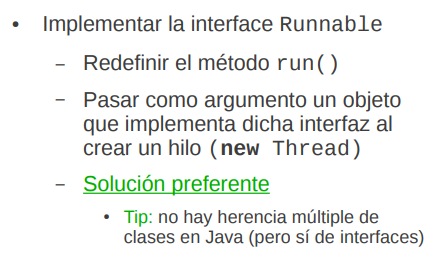
* Menos costoso que crear y destruir **procesos.**
* Todo proceso tiene, como mínimo uno, el **hilo principal**

Es transparente al sistema operativo con planificación **no determinista**. Aparecen **problemas asociados a la compartición** de memoria y ficheros.

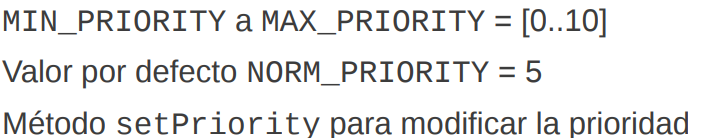
## 2.1. Ciclo de vida de un hilo



## 2.2. Threads



## 2.3. Prioridades



Método **setPriority** para modificar la prioridad. Cada hilo hereda la prioridad del hilo que lo crea.

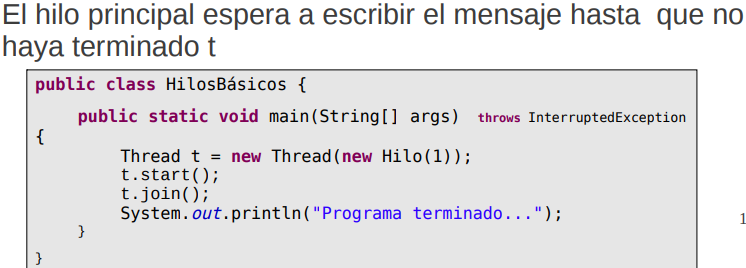
No garantizan un orden de ejecución.

## 2.4. Dormir hilos

Los hilos se pueden “dormir” o parar por un determinado tiempo. Esto suspende el hilo actual. Para dormir tareas, usamos TimeUnit.

## 2.5. Esperar a hilos

El método de **Thread.join()** permite a 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.



## 2.4. Interrupciones

Una **interrupción**  es una indicación a un hilo de que debería dejar 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 como un hilo responde a una interrupción. Los hilos, al ser interrumpidos, o bien lanzan InterruptedException, o si no consultar Thread.interrupted().

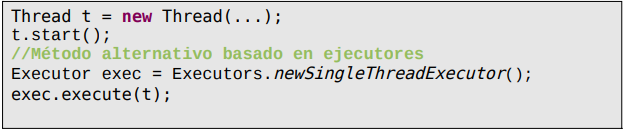
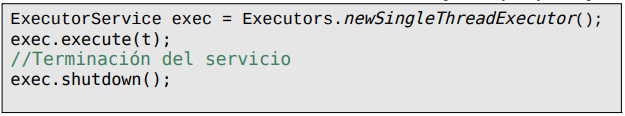
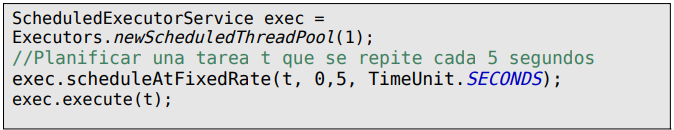
2.5. Ejecutores

Los ejecutores **crean y administran la ejecución de un objeto Runnable**.

Ventajas:

* Reutilizan hilos existentes para eliminar la sobrecarga de creación.
* Optimizan el rendimiento definiendo un numero de hilos adecuados al entorno de ejecución en explotación.

Existen una jerarquía de tres niveles de interfaces, cada uno incrementa la funcionalidad del anterior.

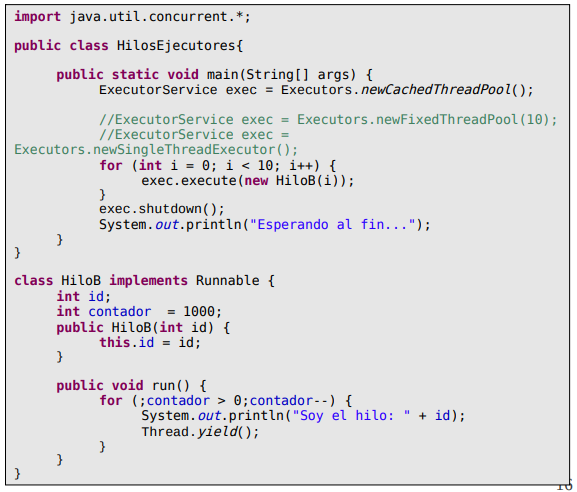
* **Executor.execute()** lanza un hilo con un objeto Runnable
* **ExecutorService** Añade características que ayudan a administrar el ciclo de vida, tanto de las tareas individuales como del propio ejecutor. 
* **ScheduledExecutorService:** Permite planificar con un cierto retardo (delay)

## 2.6. Depósito de hilos (pools)

Minimizan el costo de creación de hilos. Aunque son menos costosos que los procesos, siempre acarrea un coste adicional.

* **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.

## 2.7. Ejecutores y depósitos



# 3. Objetos y concurrencia

## 3.1. Concurrencia

Un programa concurrente es un programa que **hace mas de una cosa a la vez.** La Maquina Virtual Java y el sistema operativo subyacente hacen posible la simultaneidad aparente.

La **programación concurrente** (PRGC) en java se restringen a estructuras que afectan a una sola JVM, mientras que la **programación distribuida** (PRGD) afecta a múltiples JVM en distintos sistemas informáticos.

## 3.2. Estructuras para la programación concurrente

Sus características son las siguientes:

Autonomía e independencia, Administración de la estructura, capacidad para compartir recursos subyacentes, y comunicación entre las mismas estructuras.

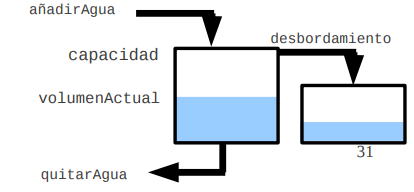
## 3.3. Concurrencia y programación OO

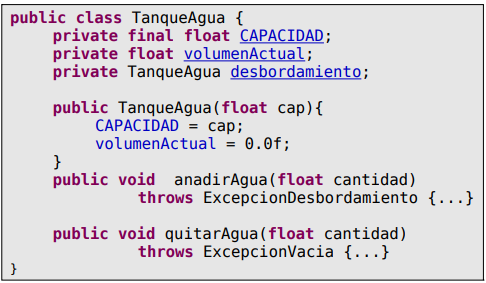
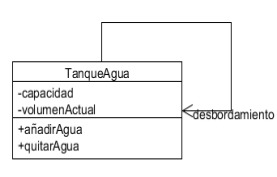
Las diferencias con la programación secuencial son las siguientes:

* **Ejecuciones no deterministas**
* **Comprensión de código** no secuencial
* Las **interferencias entre las actividades** necesitan diseño conservador.

Es una programación **basada en eventos**, en la que puede haber múltiples diseños. Hay que tener en cuenta también los efectos secundarios de interferencias y coordinación entre actividades concurrentes.

## 3.4. Transformaciones y modelos de objetos

Objetos software vs Objetos reales: Solo se consideran las características en el campo de la computación: Atributos, restricciones de estado invariantes, operaciones, conexiones con otros objetos…

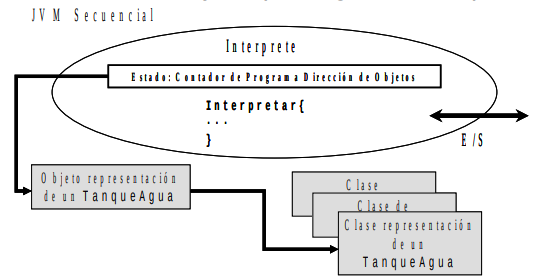


### Transformaciones secuenciales

El computador de propósito general aparenta ser cualquier objeto

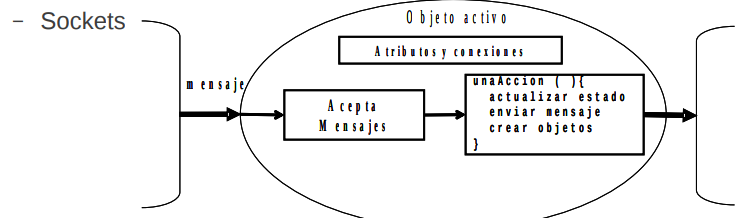
1. Carga el descriptor de la clase .class
2. Construir representación pasiva de una instancia
3. Interpretar las operaciones

JVM es en si misma un objeto que finge ser cualquier objeto



### Modelo de objetos activos o modelos de actores

Cada objeto es autónomo, y es tan potente como una JVM secuencial.

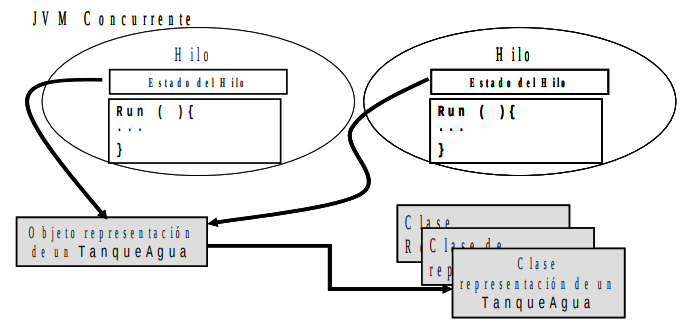


### Modelos mixtos

Una JVM puede estar compuesta de múltiples hilos. Los hilos comparten las representaciones subyacentes.

Se separan **objetos** normales/**pasivos** de los **activos**/hilos

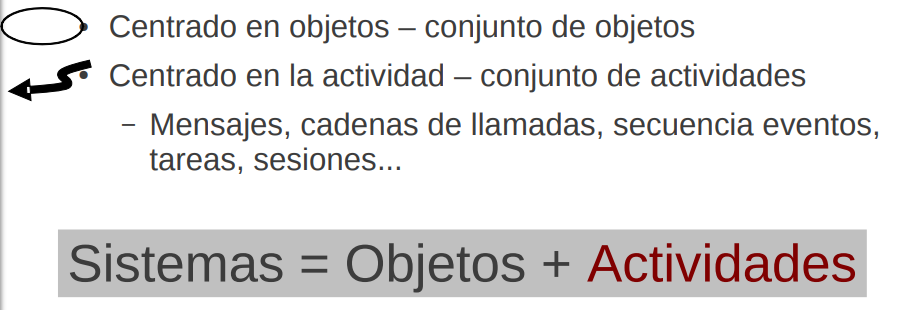
* Protección de objetos pasivos de coincidencia de hilos mediante cerrojos
* Objetos activos mas simples sólo soportan una operación run()

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.

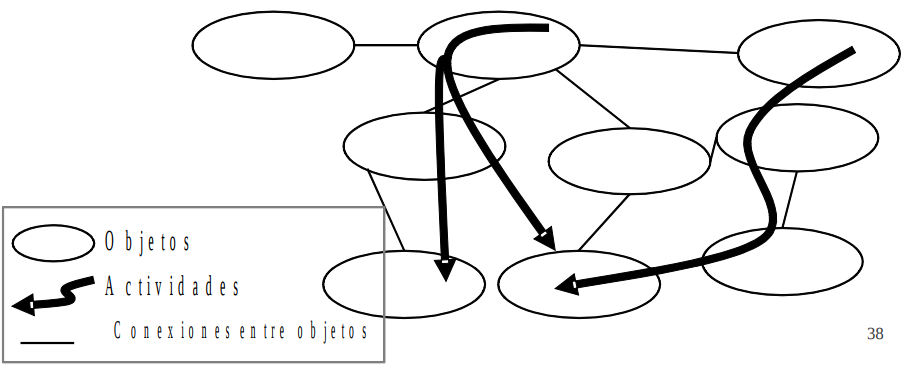
# 4. Imposiciones de diseño

Programación secuencial vs Programación concurrente: aparecen nuevos problemas de diseño.

Hay **dos perspectivas** complementarias de un sistema orientado a objetos



Las dos perspectivas son complementarias: Un objeto se puede implicar en varias actividades, y una actividad puede atravesar múltiples objetos:



Las estrategias de diseño también son complementarias:

Diseño respecto a la **corrección:**

* **Seguridad**: Nunca sucede nada malo a un objeto
* **Vivacidad**: Algo sucede eventualmente dentro de una actividad

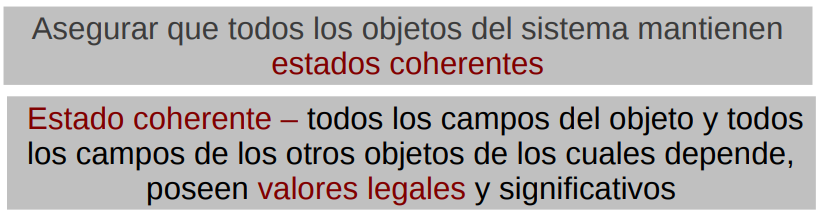
La elección de la estrategia complementaria depende del balance de efectos de fallos. Por defecto se prima la seguridad. Un ejemplo en el que se prime la vivacidad es una barra de progreso.

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 rápido.

## 4**.**1. Seguridad

A la seguridad en programación concurrente se agrega una **dimensión temporal**. El objetivo en la preservación de la seguridad es:



**Valores legales** implica cumplir invariantes a nivel conceptual (VolumenActual>=0 AND VolumenActual<=Capacidad)

Un método publico puede cambiar el estado del objeto (añadirAgua, quitarAgua)

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

Algunas técnicas de protección para **garantizar valores legales** (cumplir invariantes):

* Procesamiento de excepciones
* **Técnicas de exclusión** para garantizar la atomicidad de las acciones públicas (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 Escritura/Escritura)

Es importante la **comprensión de atributos y restricciones conceptuales** (Un contador tiene un valor entro no negativo, un termostato tiene una temperatura igual a la lectura más reciente del sensor, etc).

Hay que tener en cuenta que **pueden aparecer nuevos invariantes adicionales por decisiones de implementación**.

## 4.2. Vivacidad

Los problemas de seguridad se tienen que equilibrar con problemas de vivacidad. En **sistemas vivaces**, cada actividad **progresa hacia la terminación**. Sin embargo, el ciclo de vida de un hilo puede incluir un número de bloqueos transitorios (bloqueos synchronized, esperas, etc.)

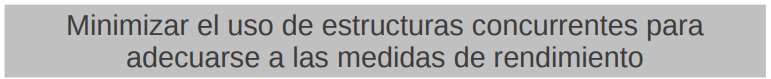
Los problemas de vivacidad son los siguientes:

* **Interbloqueo**
* **Señales perdidas:** Un hilo empieza a esperar después de la notificación para que despierte
* **Cerrojos anidados en un monitor**
* **Falta de vivacidad:** Una acción repetida falla continuamente
* **Inanición**
* **Falta de recursos**
* **Fallo distribuido**

## 4.3. Rendimiento

Las imposiciones de rendimiento amplían los problemas de vivacidad. A demás de exigir que cada método se ejecute, los objetivos del rendimiento es que se ejecute pronto y rápido.

En los diseños concurrentes multihilo se **empeora la eficiencia para mejorar la latencia**.



## 4.4. Reutilización

Una clase o un objeto es reutilizable cuando puede ser utilizado fácilmente en distintos **contextos**. La dualidad entre seguridad y vivacidad dificultan la reutilización (Un componente seguro no puede reutilizarse por no ser vivaz en todos los contextos).

Tendencias para abordar la **dependencia del contexto**:

* Reducir incertidumbre aislando parte del sistema
* Establecer las políticas y protocolos que permitan a los componentes seguir siendo accesibles