

UD 1

INTRODUCCIÓN AL PARADIGMA ORIENTADO A OBJETOS

Pedro J. Ponce de León

Versión 0.9



Indice



- **El progreso de la abstracción**
 - Definición de la abstracción
 - Lenguajes de programación y niveles de abstracción
 - Principales paradigmas de programación
 - Mecanismos de abstracción en los lenguajes de programación
- El paradigma orientado a objetos
 - Lenguajes orientados a objetos (LOO). Características básicas
 - LOO: Características opcionales
 - Historia de los LOO
 - Metas de la programación orientada a objetos (POO)

El progreso de la abstracción

Definición



■ Abstracción

- *Supresión intencionada (u ocultación) de algunos detalles de un proceso o artefacto, con el fin de destacar más claramente otros aspectos, detalles o estructuras.*

- En cada nivel de detalle cierta información se muestra y cierta información se omite.
 - Ejemplo: Diferentes escalas en mapas.
- Mediante la abstracción creamos **MODELOS** de la realidad.

El progreso de la abstracción



Lenguajes de programación y niveles de abstracción

- Los diferentes niveles de abstracción ofertados por un lenguaje, dependen de los mecanismos proporcionados por el lenguaje elegido:

- Ensamblador
- Procedimientos

Perspectiva funcional

- Paquetes
- Tipos abstractos de datos (TAD)

Perspectiva de datos

- Objetos
 - TAD
 - + paso de mensajes
 - + herencia
 - + polimorfismo

Perspectiva de servicios

El progreso de la abstracción

Lenguajes de programación y niveles de abstracción



- Los lenguajes de programación proporcionan abstracciones

Espacio del problema	Espacio de la solución
<p>Lenguajes Orientados a Objetos (LOO)</p> <p>LOO Puros</p> <p>Smalltalk, Eiffel</p>	<p>Lenguajes Ensamblador</p> <p>Lenguajes Imperativos (C, Fortran, BASIC)</p> <p>Lenguajes Específicos (LISP, PROLOG)</p>

LOO Híbridos (Multiparadigma)
C++, Object Pascal, Java,...

El progreso de la abstracción

Principales paradigmas



- **PARADIGMA:**
 - Forma de entender y representar la realidad.
 - Conjunto de teorías, estándares y métodos que, juntos, representan un modo de organizar el pensamiento.
- Principales **paradigmas de programación:**
 - Paradigma *Funcional*: El lenguaje describe procesos
 - Lisp y sus dialectos (p. ej. Scheme), Haskell, ML
 - Paradigma *Lógico*
 - Prolog
 - Paradigma *Imperativo* (o procedural)
 - C, Pascal
 - Paradigma *Orientado a Objetos*
 - Java, C++, Smalltalk, ...

El progreso de la abstracción



Mecanismos de abstracción en los lenguajes de programación

■ OCULTACIÓN DE INFORMACIÓN:

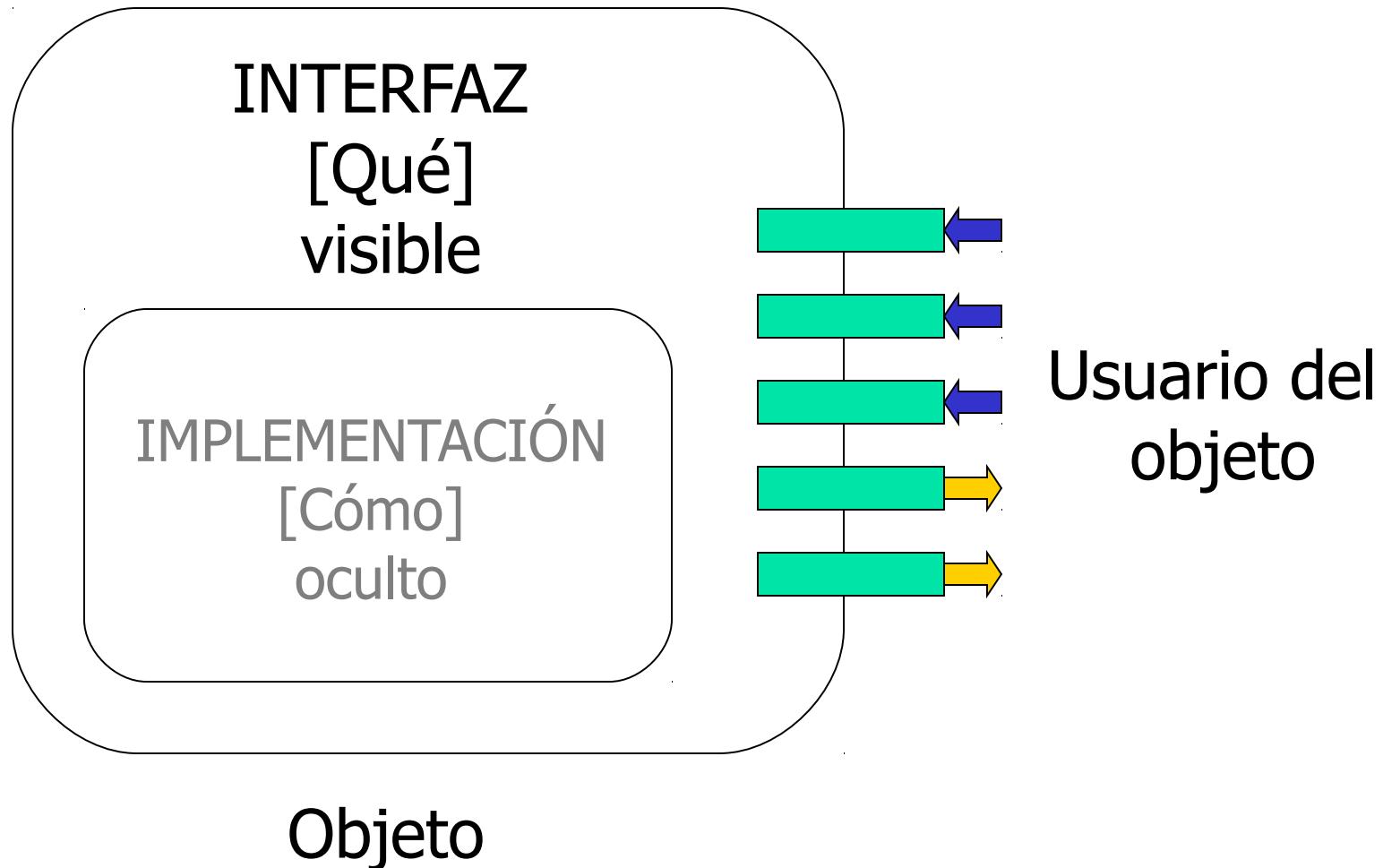
Omisión intencionada de detalles de implementación tras una interfaz simple.

- Cuando existe una división estricta entre la vista interna de un componente (objeto) y su vista externa hablamos de **ENCAPSULACIÓN**.
 - Estas dos vistas son:
 - **INTERFAZ**: QUÉ sabe hacer el objeto. Vista externa
 - **IMPLEMENTACIÓN**: CÓMO lo hace. Vista interna
 - Favorece la intercambiabilidad.
 - Favorece la comunicación entre miembros del equipo de desarrollo y la interconexión de los artefactos resultantes del trabajo de cada miembro.

El progreso de la abstracción



Mecanismos de abstracción en los lenguajes de programación





- El progreso de la abstracción
 - Definición de la abstracción
 - Lenguajes de programación y niveles de abstracción
 - Principales paradigmas de programación
 - Mecanismos de abstracción en los lenguajes de programación
- **El paradigma orientado a objetos**
 - Características básicas de los lenguajes orientados a objetos (LOO).
 - Características opcionales de los LOO
 - Historia de los LOO
 - Metas de la programación orientada a objetos (POO)

El paradigma orientado a objetos



- Metodología de desarrollo de aplicaciones en la cual éstas se organizan como colecciones cooperativas de **objetos**, cada uno de los cuales representan una **instancia** de alguna **clase**, y cuyas clases son miembros de **jerarquías de clases** unidas mediante relaciones de **herencia**. (Grady Booch)

- Cambia...
 - El modo de organización del programa:
En clases (datos+operaciones sobre datos).
 - El concepto de ejecución de programa
Paso de mensajes

- No basta con utilizar un lenguaje OO para programar orientado a objetos. Para eso hay que seguir un paradigma de programación OO.

El paradigma orientado a objetos

¿Por qué la POO es tan popular?



- POO se ha convertido durante las pasadas dos décadas en el paradigma de programación dominante, y en una herramienta para resolver la llamada crisis del software
- Motivos
 - POO escala muy bien.
 - POO proporciona un modelo de abstracción que razona con técnicas que la gente usa para resolver problemas (metáforas)
 - "*Es más fácil enseñar Smalltalk a niños que a programadores*" (Kay 77)

El paradigma orientado a objetos

Un nuevo modo de ver el mundo



- Ejemplo: Supongamos que Luis quiere enviar flores a Alba, que vive en otra ciudad.
 - Luis va a la floristería más cercana, regentada por un florista llamado Pedro.
 - Luis le dice a Pedro qué tipo de flores enviar a Alba y la dirección de recepción.
- El mecanismo utilizado para resolver el problema es
 - Encontrar un **agente** apropiado (Pedro)
 - Enviarle un **mensaje** conteniendo la petición (envía flores a Alba).
 - Es la **responsabilidad** de Pedro satisfacer esa petición.
 - Para ello, es posible que Pedro disponga de algún **método** (algoritmo o conjunto de operaciones) para realizar la tarea.
- Luis no necesita (ni le interesa) conocer el método particular que Pedro utilizará para satisfacer la petición: esa *información está OCULTA*.
- Así, la solución del problema requiere de la cooperación de varios individuos para su solución.

El paradigma orientado a objetos

Un nuevo modo de ver el mundo



Mundo estructurado en:

- Agentes y comunidades
- Mensajes y métodos
- Responsabilidades
- Objetos y clases
- Jerarquías de clases
- Enlace de métodos

El paradigma orientado a objetos

Un nuevo modo de ver el mundo



■ **Agentes y comunidades**

- Un programa OO se estructura como una comunidad de agentes que interaccionan (OBJETOS). Cada objeto juega un rol en la solución del problema. Cada objeto proporciona un servicio o realiza una acción que es posteriormente utilizada por otros miembros de la comunidad.

El paradigma orientado a objetos

Un nuevo modo de ver el mundo



■ **Mensajes y métodos**

- A un objeto se le envían mensajes para que realice una determinada acción.
- El objeto selecciona un método apropiado para realizar dicha acción.
- A este proceso se le denomina **Paso de mensajes**
- Sintáxis de un mensaje:

receptor.selector(argumentos)

```
unJuego.mostrarCarta(laCarta, 42, 47)
```

El paradigma orientado a objetos

Un nuevo modo de ver el mundo



■ **Mensajes y métodos**

- Un mensaje se diferencia de un procedimiento/llamada a función en dos aspectos:
 - En un mensaje siempre hay un receptor, lo cual no ocurre en una llamada a procedimiento.
 - La interpretación de un mismo mensaje puede variar en función del receptor del mismo.
 - Por tanto un nombre de procedimiento/función se identifica 1:1 con el código a ejecutar, mientras que un mensaje no.
 - Un ejemplo:

```
JuegoDeCartas juego = new Poker ... ó ... new Mus ... ó ...
juego.repartirCartas(numeroDeJugadores)
```

El paradigma orientado a objetos

Un nuevo modo de ver el mundo



■ **Responsabilidades**

- El comportamiento de cada objeto se describe en términos de responsabilidades
- **Protocolo:** Conjunto de responsabilidades de un objeto
- POO vs. programación imperativa

No pienses lo que puedes hacer con tus estructuras de datos.

Pregunta a tus objetos lo que pueden hacer por ti.

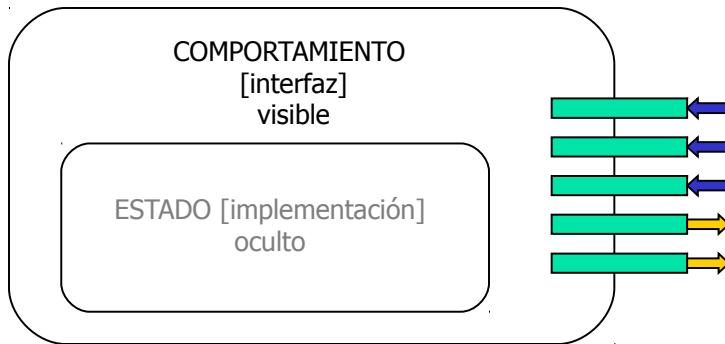
El paradigma orientado a objetos

Un nuevo modo de ver el mundo



■ Objetos y clases

- Un objeto es una **encapsulación** de un **estado** (valores de los datos) y **comportamiento** (operaciones).



- Los objetos se agrupan en categorías (**clases**).
 - Un objeto es una **instancia** de una clase.
 - El método invocado por un objeto en respuesta a un mensaje viene determinado por la clase del objeto receptor.

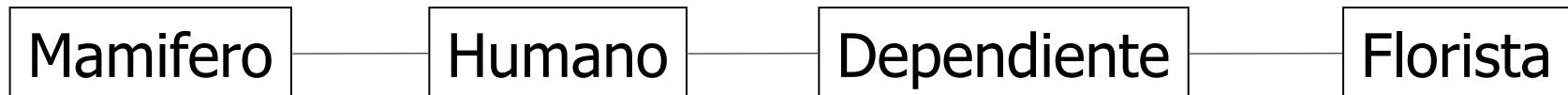
El paradigma orientado a objetos

Un nuevo modo de ver el mundo



■ **Jerarquías de clases**

- En la vida real, mucho conocimiento se organiza en términos de jerarquías. Este principio por el cual el conocimiento de una categoría más general es aplicable a una categoría más específica se denomina **generalización**, y su implementación en POO se llama **herencia**.
 - Pedro, por ser florista, es un dependiente (sabe vender y cobrar)
 - Los dependientes normalmente son humanos (pueden hablar)
 - Los humanos son mamíferos (Pedro respira oxígeno...)
- Las clases de objetos pueden ser organizadas en una estructura jerárquica de herencia. Una clase '*hijo*' **hereda** propiedades de una clase '*padre*' más alta en la jerarquía (más general):



El paradigma orientado a objetos

Un nuevo modo de ver el mundo



■ **Enlace de métodos**

Instante en el cual una llamada a un método es asociada al código que se debe ejecutar

- Enlace estático: en tiempo de compilación
- Enlace dinámico: en tiempo de ejecución
- Supongamos que en este ejemplo la asignación a la variable 'juego' depende de la interacción con el usuario (tiempo de ejecución).

```
JuegoDeCartas juego = new Poker ... ó ... new Mus ... ó ...
juego.repartirCartas(numeroDeJugadores)
```

El mensaje 'repartirCartas' deberá tener enlace dinámico.



- El progreso de la abstracción
 - Definición de la abstracción
 - Principales paradigmas de programación
 - Lenguajes de programación y niveles de abstracción
 - Mecanismos de abstracción en los lenguajes de programación
- El paradigma orientado a objetos
 - **Características básicas de los lenguajes orientados a objetos**
 - LOO: Características opcionales
 - Historia de los LOO
 - Metas de la programación orientada a objetos (POO)

Características Básicas de un LOO



- Según Alan Kay (1993), son seis:
 - (1) Todo es un **objeto**
 - (2) Cada objeto es construido a partir de otros objetos.
 - (3) Todo objeto es **instancia** de una **clase**
 - (4) Todos los objetos de la misma clase pueden recibir los mismos mensajes (realizar las mismas acciones). La clase es el lugar donde se define el **comportamiento** de los objetos y su estructura interna.
 - (5) Las clases se organizan en una estructura arbórea de raíz única, llamada **jerarquía de herencia**.
 - (6) Un programa es un conjunto de objetos que se comunican mediante el **paso de mensajes**.



- **Polimorfismo**
 - Capacidad de una entidad de referenciar elementos de distinto tipo en distintos instantes

p. ej., enlace dinámico
- **Genericidad**
 - Definición de clases parametrizadas (*templates* en C++, *generics* en Java) que definen tipos genéricos.

p. ej.: Lista<T> : donde T puede ser cualquier tipo.
- **Gestión de Errores**
 - Tratamiento de condiciones de error mediante **excepciones**
- **Aserciones**
 - Expresiones que especifican qué hace el software en lugar de cómo lo hace
 - **Precondiciones**: propiedades que deben ser satisfechas cada vez que se invoca una servicio
 - **Postcondiciones**: propiedades que deben ser satisfechas al finalizar la ejecución de un determinado servicio
 - **Invariantes**: aserciones que expresan restricciones para la consistencia global de sus instancias.



- **Tipado estático**
 - Es la imposición de un tipo a un objeto en tiempo de compilación
 - Se asegura en tiempo de compilación que un objeto entiende los mensajes que se le envían.
 - Evita errores en tiempo de ejecución
- **Recogida de basura** (*garbage collection*)
 - Permite liberar automáticamente la memoria de aquellos objetos que ya no se utilizan.
- **Concurrencia**
 - Permite que diferentes objetos actúen al mismo tiempo, usando diferentes *threads* o hilos de control.



Características opcionales de un LOO (3/3)

■ Persistencia

- Es la propiedad por la cual la existencia de un objeto trasciende la ejecución del programa.
 - Normalmente implica el uso de algún tipo de base de datos para almacenar objetos.

■ Reflexión

- Capacidad de un programa de manipular su propio estado, estructura y comportamiento.
 - En la programación tradicional, las instrucciones de un programa son 'ejecutadas' y sus datos son 'manipulados'.
 - Si vemos a las instrucciones como datos, también podemos manipularlas.

```
String instr = "System.out.println( " ;
ejecuta(instr + "27") );
Class c = Class.forName( "String" ) ;
Method m = c.getMethod( "length" , null ) ;
m.invoke(instr,null) ;
```



Características opcionales de un LOO: conclusiones

- Lo ideal es que un lenguaje proporcione el mayor número posible de las características mencionadas
 - Orientación a objetos no es una condición booleana: un lenguaje puede ser 'más OO' que otro.



- El progreso de la abstracción
 - Definición de la abstracción
 - Principales paradigmas de programación
 - Lenguajes de programación y niveles de abstracción
 - Mecanismos de abstracción en los lenguajes de programación
- El paradigma orientado a objetos
 - Características básicas de los lenguajes orientados a objetos (LOO).
 - LOO: Características opcionales
 - **Historia de los LOO**
 - Metas de la programación orientada a objetos (POO)



Historia de los L.O.O.

Año	Lenguaje	Creadores	Observaciones
1967	Simula	Norwegian Computer Center	<i>clase, objeto, encapsulación</i>
1970s	Smalltalk	Alan Kay	<i>método y paso de mensajes, enlace dinámico, herencia</i>
1985	C++	Bjarne Stroustrup	Laboratorios Bell. Extensión de C. Gran éxito comercial (1986->)
1986	1ª Conf. OOPSLA		Objective C, Object Pascal, C++, CLOS,... Extensiones de lenguajes no OO (C, Pascal, LISP,...)
'90s	Java	Sun	POO se convierte en el paradigma dominante. Java: Ejecución sobre máquina virtual
'00->	C#, Python, Ruby,...		Más de 170 lenguajes OO... Lista TIOBE (Del Top 10, 8 o 9 son OO)

Historia de los L.O.O.: Actualidad



- A partir de los 90' proliferan con gran éxito la tecnología y lenguajes OO.
- Los más implantados en la actualidad son **Java, C++ y PHP** (lista TIOBE)
- **C#, Python, Objective-C** son otros lenguajes OO muy utilizados
- Híbridos (OO, procedural): PHP, C++, Visual Basic, Javascript
- Otros LOO: Delphi, Ruby, ActionScript,...

Indice



- El progreso de la abstracción
 - Definición de la abstracción
 - Principales paradigmas de programación
 - Lenguajes de programación y niveles de abstracción
 - Mecanismos de abstracción en los lenguajes de programación
- El paradigma orientado a objetos
 - Características básicas de los lenguajes orientados a objetos (LOO).
 - LOO: Características opcionales
 - Historia de los LOO
 - **Metas de la programación orientada a objetos (POO)**



- La meta última del incremento de abstracción de la POO es
 - **MEJORAR LA CALIDAD DE LAS APLICACIONES.**
- Para medir la calidad, Bertrand Meyer define unos parámetros de calidad:
 - **PARÁMETROS EXTRÍNSECOS**
 - **PARÁMETROS INTRÍNSECOS**



- **Fiabilidad: corrección + robustez:**
 - **Corrección:** capacidad de los productos software para realizar con exactitud sus tareas, tal y como se definen en las especificaciones.
 - **Robustez:** capacidad de los sistemas software de reaccionar apropiadamente ante condiciones excepcionales.
- La corrección tiene que ver con el comportamiento de un sistema en los casos previstos por su especificación. La robustez caracteriza lo que sucede fuera de tal especificación.

Metas de la P.O.O.

Principales parámetros Intrínsecos



- **Modularidad: extensibilidad + reutilización:**
 - **Extensibilidad:** facilidad de adaptar los productos de software a los cambios de especificación.
 - **Reutilización:** Capacidad de los elementos software de servir para la construcción de muchas aplicaciones diferentes.
 - Las aplicaciones a menudo siguen patrones similares
- En definitiva: producir aplicaciones + fáciles de cambiar: **mantenibilidad**



- **El progreso de la abstracción**
 - Definición de la abstracción
 - Principales paradigmas de programación
 - Lenguajes de programación y niveles de abstracción
 - Mecanismos de abstracción en los lenguajes de programación
- **El paradigma orientado a objetos**
 - Características básicas de los lenguajes orientados a objetos (LOO).
 - LOO: Características opcionales
 - Historia de los LOO
 - Metas de la programación orientada a objetos (POO)

Bibliografía



- Cachero et. al.
 - ***Introducción a la programación orientada a Objetos***
 - Capítulo 1
- Timothy Budd
 - ***An introduction to OO Programming. 3rd Edition.***
Addison Wesley, 2002
 - Capítulos 1 y 2
- Bertrand Meyer
 - ***Object Oriented Software Construction***
- Bruce Eckel
 - ***Piensa en Java, 4^a edición***
(Thinking in C++ / Thinking in Java, online)
 - Capítulo 1

UD 2

CONCEPTOS BÁSICOS de la programación orientada a objetos

Versión 0.1
(Curso 14/15)

Pedro J. Ponce de León



Indice



- Objetos
- Clases
- Atributos
- Operaciones
- UML
- Relaciones
 - Asociación
 - Todo-parte
 - Uso
- Metaclasses

Objeto

Definición



- Un objeto es cualquier cosa a la que podamos asociar unas determinadas **propiedades** y **comportamiento**.
- Desde el punto de vista del analista: un objeto representa una entidad (real o abstracta) con un papel bien definido en el dominio del problema.
- Desde el punto de vista del programador: un objeto es una estructura de datos sobre la cual podemos realizar un conjunto bien definido de operaciones.

Objeto

Definición



- Según *Grady Booch*: Un objeto tiene un estado, un comportamiento y una identidad:
 - **Estado**: conjunto de propiedades del objeto y valores actuales de esas propiedades.
 - **Comportamiento**: modo en que el objeto actúa y reacciona ante los mensajes que se le envían (con posibles cambios en su estado). Viene determinado por la **clase** a la que pertenece el objeto.
 - **Identidad**: propiedad que distingue a unos objetos de otros (nombre único de variable)

Clase

Definición



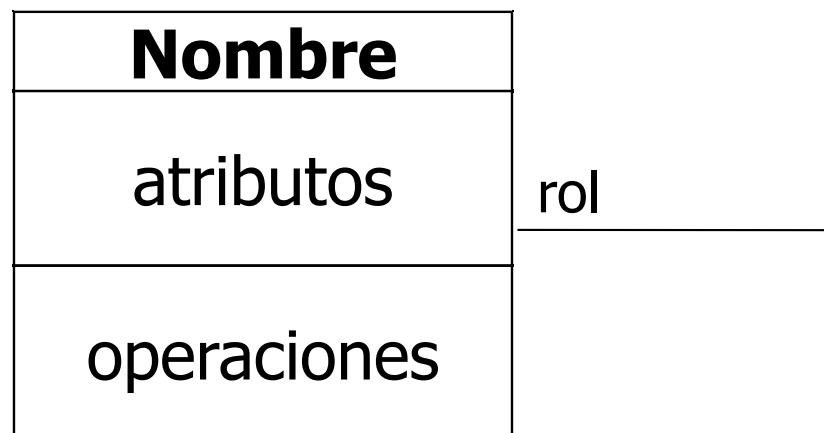
- Abstracción de los atributos (características), operaciones, relaciones y semántica comunes a un conjunto de objetos.
- Así, una clase representa al conjunto de objetos que comparten una estructura y un comportamiento comunes. Todos ellos serán **instancias** de la misma clase.



- **Identificador de Clase:** nombre

- **Propiedades**

- **Atributos** o *variables*: datos necesarios para describir los objetos (*instancias*) creados a partir de la clase.
 - La combinación de sus valores determina el estado de un objeto.
- **Roles**: relaciones que una clase establece con otras clases.
- **Operaciones, métodos, servicios**: acciones que un objeto conoce cómo ha de ejecutar.



```
class Nombre {  
    private tipo1 atributo1;  
    private tipo2 atributo2;  
    ...  
    ...  
    public tipoX operacion1() {...}  
    public tipoY operacion2(...) {...}  
    ...  
} // Java
```

¿Objetos o clases?



- Película **CLASE**
- Carrete de película **CLASE**
- Carrete con nº de serie 123456 **OBJETO**
- Pase de la película 'La vida de Brian' en el cine Muchavista a las 19:30 **OBJETO**

En general:

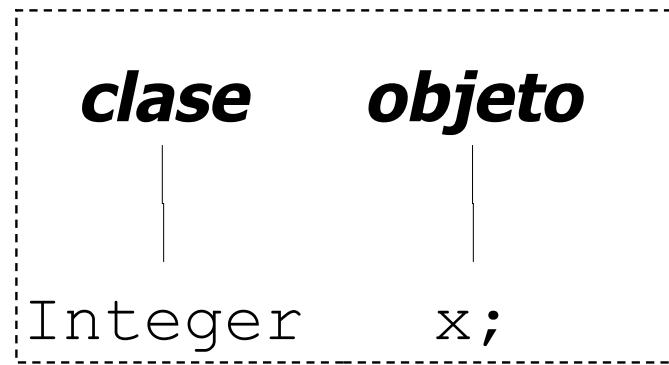
- Algo será una clase si puede tener instancias.
- Algo será un objeto si es algo único que comparte características con otras cosas similares

Objeto

Objetos y clases en un lenguaje de programación



- Clase Tipo
- Objeto Variable o constante



Clase: caracterización de un conjunto de objetos que comparten propiedades

Atributos

Definición



- **Atributo** (*dato miembro o variable de instancia*)
 - Porción de información que un objeto posee o conoce de sí mismo.
 - Suelen ser a su vez objetos
 - Se declaran como 'campos' de la clase.
 - **Visibilidad** de un atributo
 - *Indica desde donde se puede acceder a él.*
 - + Pública (interfaz) -> desde cualquier lugar
 - - Privada (implementación) -> sólo desde la propia clase
 - # Protegida (implementación) -> desde clases derivadas
 - ~ De paquete (en Java) -> desde clases definidas en el mismo paquete

Es habitual que los atributos formen parte de la implementación (parte oculta) de una clase, pues conforman el estado de un objeto.



Tipos de Atributo

■ **Constantes / Variables**

- Constante: ej., una casa se crea con un número determinado de habitaciones (característica estable):
 - `private final int numHab;`
- Variable: ej., una persona puede variar su sueldo a lo largo de su vida:
 - `private int sueldo;`

■ **De instancia / De clase**

- De instancia: atributos o características de los objetos representados por la clase. Se guarda espacio para una copia de él por cada objeto creado:
 - `private String nombre; // nombre de un Empleado`
- De clase: características de una clase comunes a todos los objetos de dicha clase:
 - `private static String formatoFecha; // de la clase Fecha`



Atributos de clase

- Implican una sola zona de memoria reservada para todos los objetos de la clase, y no una copia por objeto, como sucede con las variables de instancia.
 - **Sirven para:**
 - **Almacenar características comunes (constantes) a todos los objetos**
 - Número de ejes de un coche
 - Número de patas de una araña
 - **Almacenar características que dependen de todos los objetos**
 - Número de estudiantes en la universidad
 - Un atributo estático puede ser accedido desde cualquier objeto de la clase, ya que es un dato miembro de la clase.

Operaciones

Definición



- **Operación** (función miembro, método o servicio de la clase)
 - Acción que puede realizar un objeto en respuesta a un mensaje. Definen el **comportamiento** del objeto.
 - Tienen asociada una **visibilidad** (como los atributos)
 - Pueden ser **de clase o de instancia** (como los atributos)
 - Pueden modificar el estado del sistema (**órdenes**) o no (**consultas**)
 - **Signatura de una operación en :**
TipoRetorno
NombreClase.NombreFuncionMiembro(parametros)

Operaciones

Tipos de Operaciones



■ De instancia/De clase

■ ***Operaciones de instancia:***

- Operaciones que pueden realizar los objetos de la clase.
- Pueden acceder directamente a atributos tanto de instancia como de clase.
- Normalmente actúan sobre el objeto receptor del mensaje.

```
Circulo c = new Circulo();  
  
c.setRadio(3);  
double r = c.getRadio();  
c.pintar();
```

```
void setRadio(double r) {  
    if (r > 0.0) radio = r;  
    else radio = 0.0;  
}
```



- **De instancia/De clase**
 - ***Operaciones de clase:***

- Operaciones que acceden exclusivamente a atributos de clase.
- No existe receptor del mensaje (a menos que se pase explícitamente como parámetro).
- Se pueden ejecutar sin necesidad de que exista ninguna instancia.

```
class Circulo {  
  
    private static final double pi=3.141592;  
  
    public static double getRazonRadioPerimetro()  
    {  
        return 2*pi;  
    }  
    ...  
};
```

Operaciones

Tipos de Operaciones



■ Órdenes

- Pueden modificar el estado del objeto receptor.

```
c.setRadio(3); // modifica el radio de 'c'
```

■ Consultas

- No modifican al objeto receptor.

```
c.getRadio(); // consulta el radio de 'c'
```

Operaciones sobrecargadas



- Algunos LOO soportan la **sobrecarga** de operaciones.
 - Consiste en la existencia, dentro de un mismo ámbito, de más de una operación definida con el mismo nombre (selector), pero diferente número y/o tipo de argumentos.

```
class Circulo {  
    // pinta sin relleno  
    public void pintar() {...}  
    // pinta con relleno  
    public void pintar(Color) {...}  
}
```

```
Circulo c = new Circulo();  
c.pintar();  
c.pintar(azul);
```

Operaciones

Referencia al objeto receptor



- En muchos LOO, en los métodos el receptor es un argumento implícito.
- Para obtener una referencia a él dentro de un método de instancia, existe una *pseudo-variable*:
 - En C++ y Java, se llama **this**.
 - En otros lenguajes, es **self**
- Ejemplo en Java:

receptor.selector(this , <argumentos>)

```
class Autoref {  
    private int x;  
  
    public Autoref auto() { return this; }  
    public int getX() { return x; }  
    public int getX2() { return this.x; }  
    public int getX3() { return getX(); }  
    public int getX4() { return this.getX(); }  
}
```

Constructor



- Operación cuyo objetivo es crear e inicializar objetos.
 - Se invoca siempre que se crea un objeto, mediante el operador **new** (en Java).
 - El enlazado de creación e inicialización asegura que un objeto nunca puede ser utilizado antes de que haya sido correctamente inicializado.
 - En Java y C++ tienen el mismo nombre que la clase y no devuelven nada (ni siquiera void).

```
class Circulo {  
  
    public Circulo() {...}; // Constructor por defecto  
    public Circulo(double r) {...}; // Constructor sobrecargado  
}
```

```
Circulo c = new Circulo();  
Circulo c2 = new Circulo(10);
```

Constructor



- **Constructor por defecto:**

- Es conveniente definir siempre uno que permita la inicialización **sin parámetros** de un objeto, donde los atributos de éste se inicialicen con valores por defecto.

```
public Circulo()
{
    super(); // llamada al constructor de Object (automático)
    radio = 1.0;
}
```

- En Java y C++, si no se define ninguno de manera explícita, el compilador genera uno con visibilidad pública, llamado **constructor de oficio**.

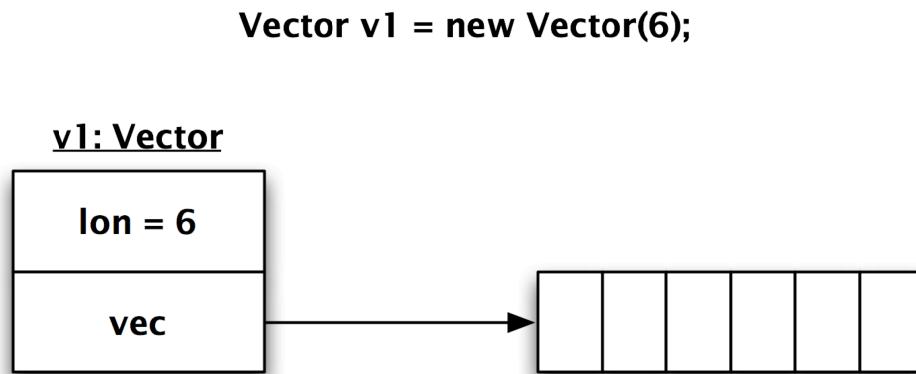
```
public Circulo()
{
    super();
    /* Java: todos los atributos = 0 ó null */
}
```

Copia de objetos



- Existen dos formas de 'copiar' o 'clonar' objetos
 - Shallow copy (copia superficial)
 - Copia bit a bit de los atributos de un objeto
 - Deep copy (copia completa)
- Supongamos una clase Vector:

```
class Vector {  
    public Vector(int lo) {  
        lon = lo;  
        vec = new int[lo];  
    }  
    private int lon;  
    private int[] vec;  
}
```



Copia de objetos



- Shallow copy
 - En C++ y Java, es el modo de copia por defecto

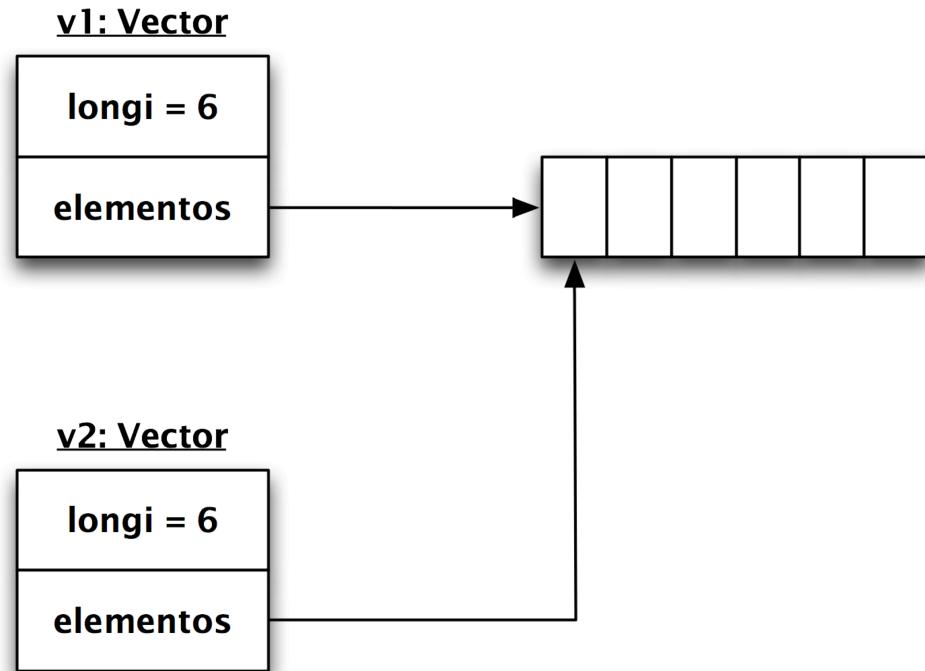
C++:

```
Vector v1(6);  
Vector v2(v1);
```

Java:

```
Vector v1 = new Vector(6);  
Vector v2 = v1.clone();
```

```
Vector v1 = new Vector(6);  
Vector v2 = v1.clone();
```



Copia de objetos

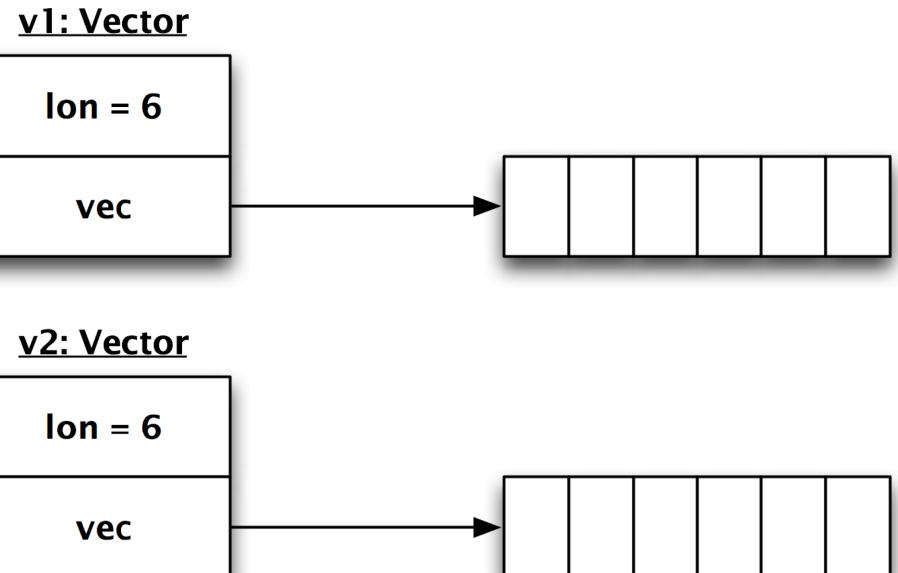


- Deep copy
 - Es necesario implementar explícitamente
 - C++: un **constructor de copia**
 - Java: un ctor. de copia ó el método **clone()**

Java: constructor de copia

```
class Vector
...
public Vector(Vector v) {
    this(v.lon); // llama a Vector(int)
    for (int i=0; i<lon; i++)
        vec[i] = v.vec[i];
}
```

```
Vector v1 = new Vector(6);
Vector v2 = new Vector(v1);
```



Copia de objetos



- Deep copy con **clone()**

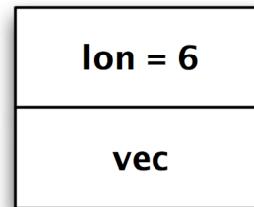
La diferencia con el ctor. de copia es que el método `clone()` también copia la parte del objeto definida en la clase `Object` (esto quedará más claro al estudiar herencia).

Java:

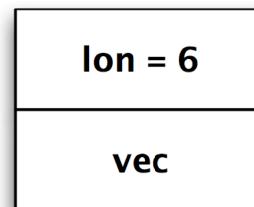
```
class Vector implements Cloneable  
...  
public Vector clone() throws  
CloneNotSupportedException {  
  
    Vector clon = (Vector) super.clone();  
    clon.vec = new int[vec.length];  
  
    for (int i=0; i<vec.length; i++)  
        clon.vec[i] = vec[i];  
  
    return clon;  
}
```

```
Vector v1 = new Vector(6);  
Vector v2 = v1.clone();
```

v1: Vector



v2: Vector



Destrucción de objetos

Java: Recolector de basura y finalize()



- La mayoría de lenguajes OO disponen de alguna forma de ejecutar código de usuario cuando un objeto es destruido.
 - C++: Destructores
 - Java: métodos `finalize()`
- En Java, el recolector de basura se encarga de destruir los objetos para los cuales ya no existen referencias en la aplicación.
Implica que el programador no tiene control sobre cuándo exactamente se libera la memoria de un objeto.

En Java y Eiffel podemos definir métodos `finalize()`, que serán ejecutados justo antes de que el objeto sea destruido.

- `protected void finalize() {} // def. en clase Object`

Los métodos `finalize()` se usan para liberar recursos que el objeto haya podido adquirir (cerrar ficheros abiertos, conexiones con bases de datos,...).

→ Normalmente no es necesario definirlos.

Forma canónica de una clase



- Es el conjunto de métodos que toda clase, independientemente de su funcionalidad específica, debería definir. Suelen existir una definición 'de oficio' proporcionada por el compilador y/o máquina virtual.
- En C++, p. ej. son éstos:
 - Constructor por defecto
 - Constructor de copia
 - Operador de asignación
 - Destructor
- En Java, son
 - Constructor por defecto (constructor de oficio)
 - `public String toString()` - Representación del objeto en forma de cadena
 - `public boolean equals(Object o)` – comparación de objetos
 - `public int hashCode()` - identificador numérico de un objeto

Forma canónica de una clase en Java



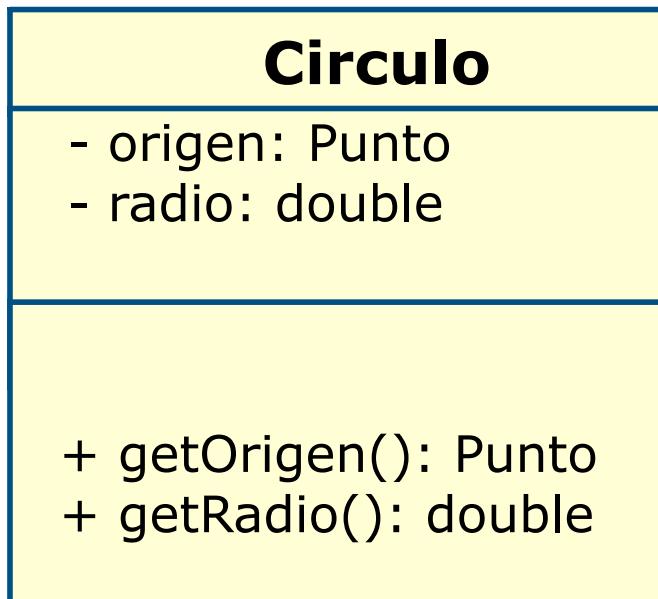
```
public class Nombre
{
    public Nombre() { ... }

    public String toString() { return ... }

    public boolean equals(Object o)
    { ... Define una relación de equivalencia:
        Reflexiva: a.equals(a) debe devolver true.
        Simétrica: si a.equals(o) entonces o.equals(a).
        Transitiva: si a.equals(b) y b.equals(c), entonces a.equals(c).
        (Por defecto devuelve true si ambos objetos son el mismo objeto
        en memoria.)
    }
    public int hashCode()
    { ... Consistente con 'equals': Dos objetos "equals"
        deben tener el mismo código hash.
    }
}
```



- Distintos LOO poseen distinta sintaxis para referirse a los mismos conceptos.
 - Las líneas de código no son un buen mecanismo de comunicación
- UML resuelve este problema y homogeneiza el modo en que se comunican conceptos OO



```
// C++

class Circulo{
    public:
        Punto getOrigen();
        double getRadio();

    private:
        Punto origen;
        double radio;
}
```

Notación UML de Clases, Atrib. y Oper.

Otras equivalencias



Círculo

- origen: Punto
- radio: double

- + getOrigen(): Punto
- + getRadio(): double

// Java y C#

```
class Circulo{
    public Punto getOrigen()
        {return origen;}
    public double getRadio()
        {return radio;}
    private Punto origen;
    private double radio;
}
```

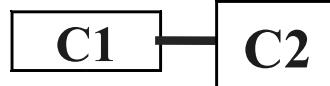
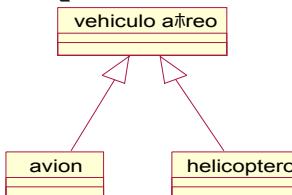


Relaciones entre Clases y Objetos

- Booch: Un objeto por sí mismo es soberanamente aburrido.
- La resolución de un problema exige la colaboración de objetos.
 - Esto exige que los agentes **se conozcan**
- El conocimiento entre objetos se realiza mediante el establecimiento de **relaciones**.
- Las relaciones se pueden establecer **entre clases** o **entre objetos**.
- Además, a nivel de objetos, podemos encontrar dos tipos de relaciones:
 - **Persistentes:** recogen caminos de comunicación entre objetos que se almacenan de algún modo y que por tanto pueden ser reutilizados en cualquier momento.
 - **No persistentes:** recogen caminos de comunicación entre objetos que desaparecen tras ser utilizados.



Relaciones entre Clases y Objetos

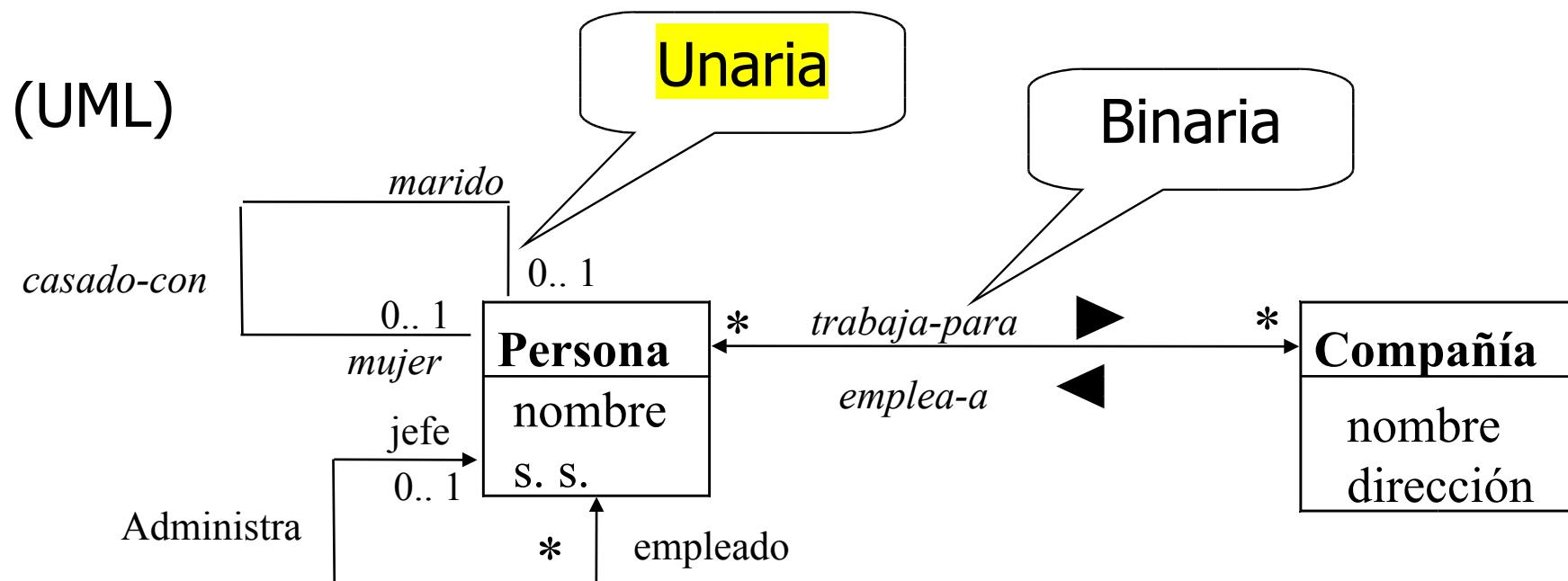
	Persistente	No persist.
Entre objetos	<ul style="list-style-type: none">▪ Asociación▪ Todo-Parte<ul style="list-style-type: none">▪ Agregación▪ Composición   	<ul style="list-style-type: none">▪ Uso (dependencia) 
Entre clases	Generalización (Herencia) 	

Relaciones entre Objetos

Asociación



- Expresa una relación (unidireccional o bidireccional) entre los objetos instanciados a partir de las clases conectadas.
- El sentido en que se recorre la asociación se denomina **navegabilidad** de la asociación:



Relaciones entre Objetos

Asociación



- Cada extremo de la asociación se caracteriza por:
 - **Rol:** papel que juega el objeto situado en cada extremo de la relación en dicha relación
 - Implementación: nombre de la referencia
 - **Multiplicidad:** número de objetos **mínimo** y **máximo** que pueden relacionarse con un objeto del extremo opuesto de la relación.
 - Por defecto 1
 - Formato: (*mínima..máxima*)
 - Ejemplos (notación UML)
- | | |
|--------------|-------------------------------------|
| 1 | Uno y sólo uno (por defecto) |
| 0..1 | Cero a uno. También (0,1) |
| M..N | Desde M hasta N (enteros naturales) |
| * | Cero a muchos |
| 0..* | Cero a muchos |
| 1..* | Uno a muchos (al menos uno) |
| 1,5,9 | Uno o cinco o nueve |



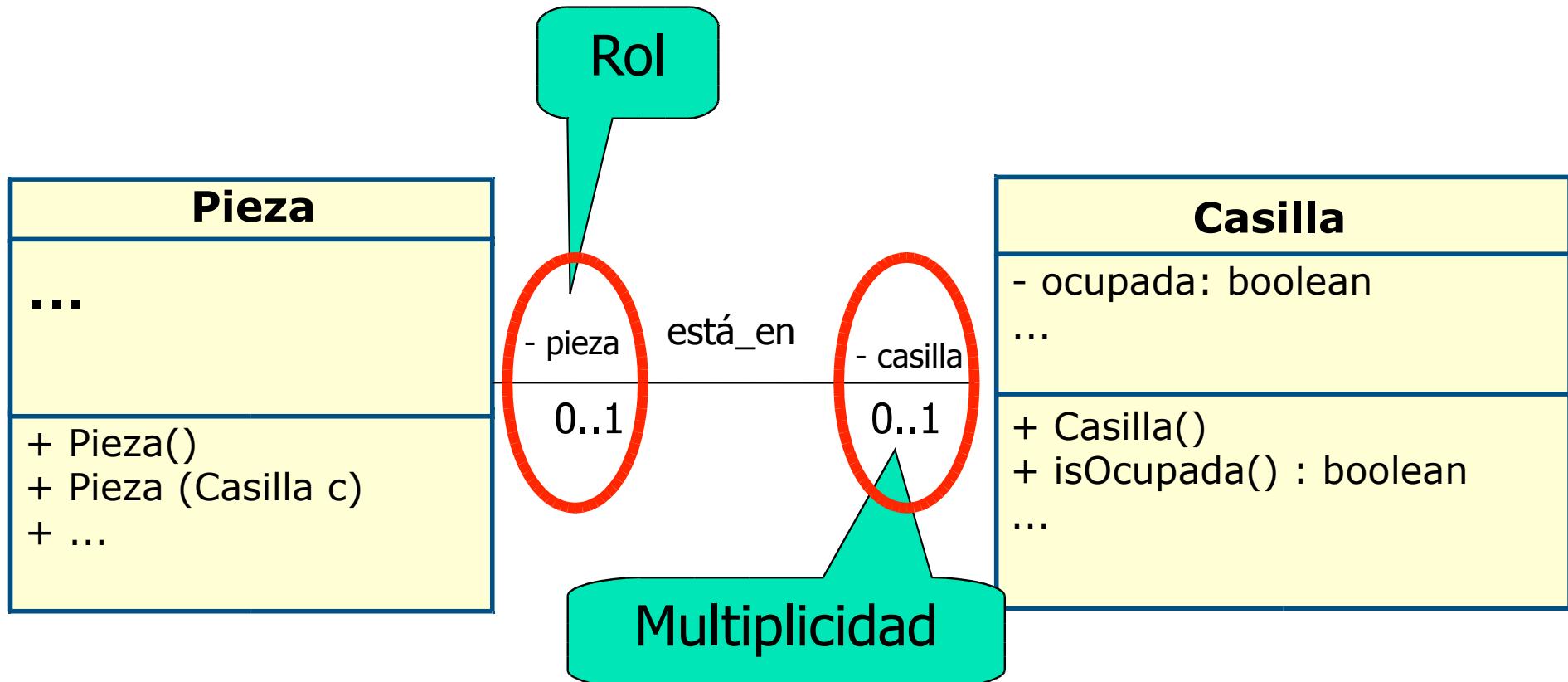
- En una asociación, dos objetos A y B asociados entre sí existen de forma independiente
 - La creación o desaparición de uno de ellos implica únicamente la creación o destrucción de la relación entre ellos y nunca la creación o destrucción del otro objeto.
- Implementación
 - Una sola referencia o un vector de referencias del tamaño indicado por la cardinalidad *máxima*.
 - La decisión sobre en qué clase se introduce el nuevo dato miembro depende de la navegabilidad de la asociación.
 - Si el máximo es *: array dinámico de referencias (Java: Vector, ArrayList, LinkedList,...).

Relaciones entre Objetos

Asociación: Ejemplo



- A partir del dibujo de la Fig., define la clase Pieza
 - Una pieza se relaciona con 0..1 casillas
 - Una casilla se relaciona con 0..1 piezas



Relaciones entre Objetos

Asociación: Ejemplo



```
class Pieza{  
    public Pieza() {casilla=null;} // Constructor por def.  
    public Pieza(Casilla c) { // Ctor. sobrecargado  
        casilla=c;  
    }  
  
    public Casilla getCasilla() { return casilla; }  
    public void setCasilla(Casilla c) { casilla = c; }  
  
private Casilla casilla; // 'casilla': nombre del rol  
    ...  
}
```

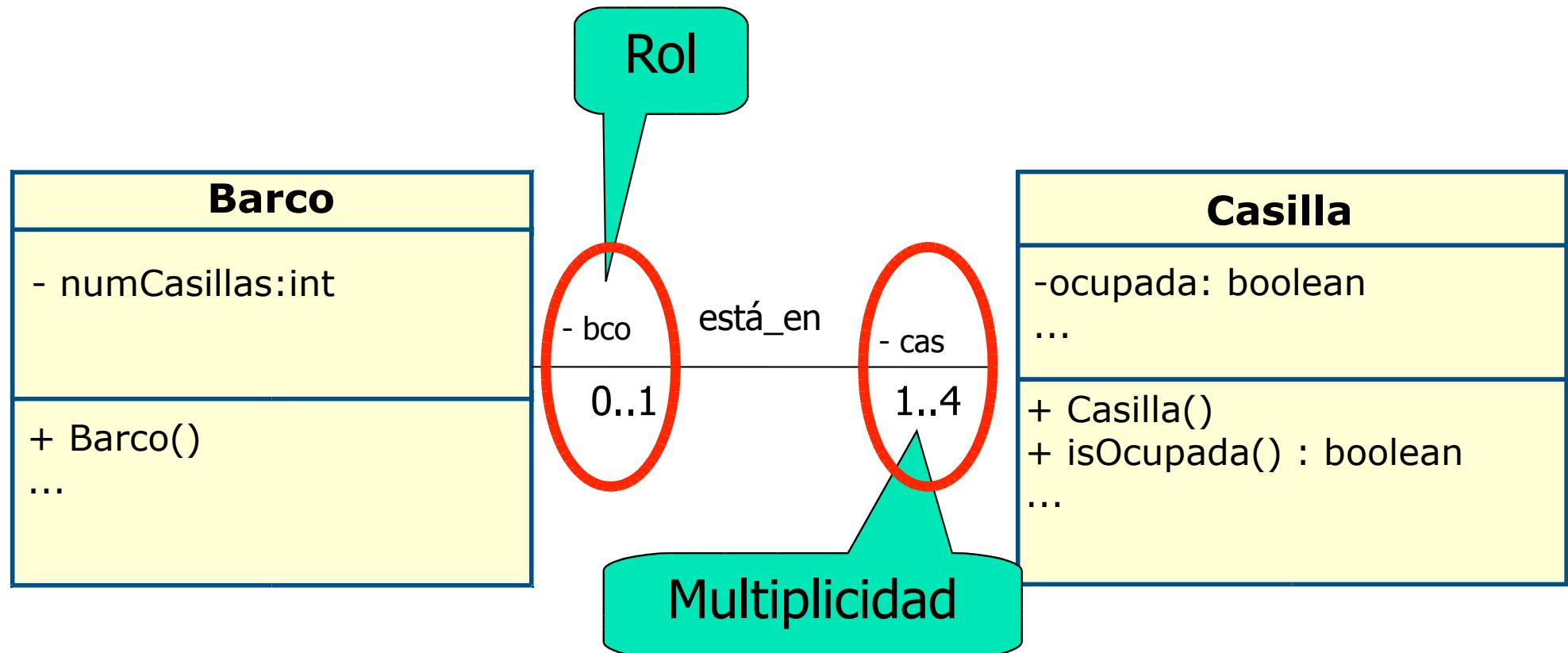
(Implementa la clase Casilla de forma similar)

Relaciones entre Objetos

Asociación: Ejemplo



- A partir del dibujo de la Fig., define la clase Barco
 - Un barco se relaciona con 1..4 casillas
 - Una casilla se relaciona con 0..1 barcos



Relaciones entre Objetos

Asociación: Ejemplo



```
class Barco{  
    private static final int MAX_CAS=4;  
    private Casilla cas[] = new Casilla[MAX_CAS];  
    private int numCasillas;  
  
    public Barco() {  
        numCasillas=0;  
        for (int x=0;x<MAX_CAS;x++)  
            cas[x]=null;  
    }  
}
```

- ¿Detectáis alguna posible inconsistencia que no controle este código? Modifica lo que sea necesario.
- ¿Cambiaría en algo el código de la clase Casilla definida en el ejercicio anterior (aparte del nombre de la referencia a Barco)?

Relaciones entre Objetos

Asociación: Ejemplo



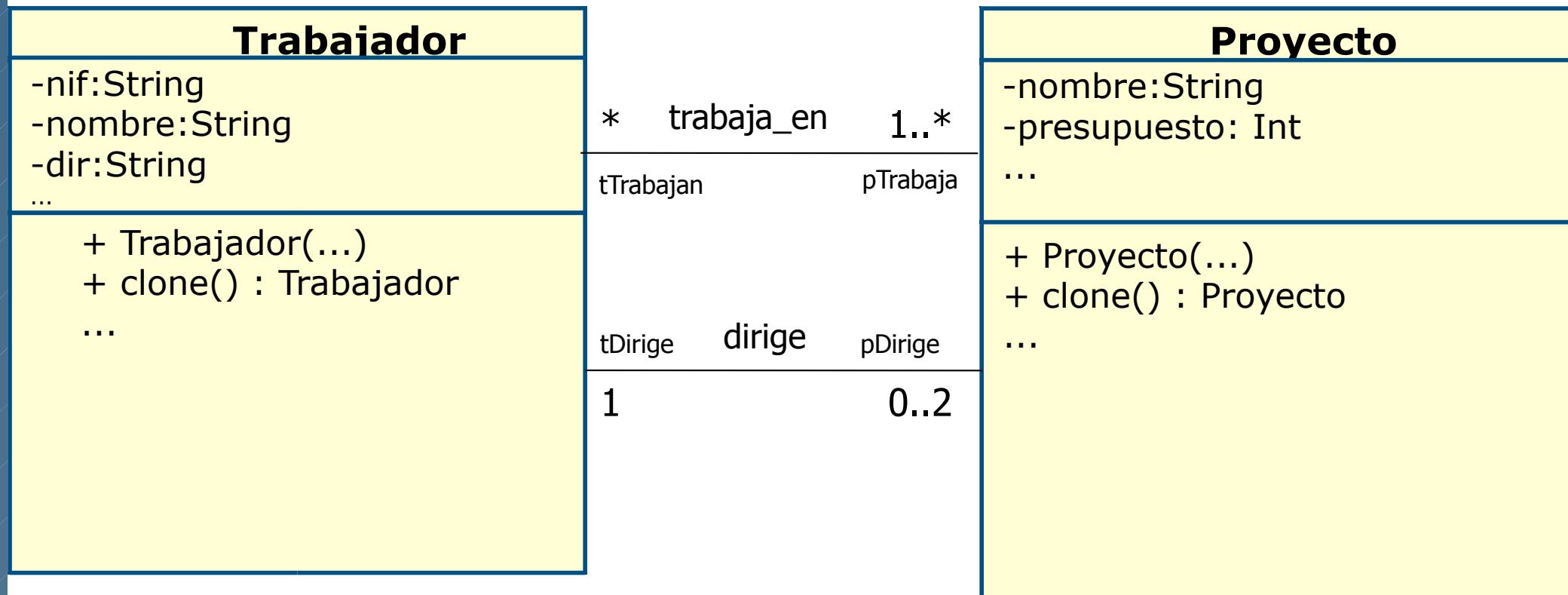
```
class Barco{  
    private static final int MAX_CAS=4;  
    private Vector<Casilla> cas;  
    // usando Vector del API java.util.*  
  
    public Barco(Vector<Casilla> c) {  
        cas=null;  
        if (c.size() <= MAX_CAS && c.size() > 0)  
            cas = c;  
    }  
    // nota: lo correcto sería lanzar una excepción (tema 3)  
    // si el número de casillas no es correcto  
}
```

Relaciones entre Objetos

Asociación: Ejercicio



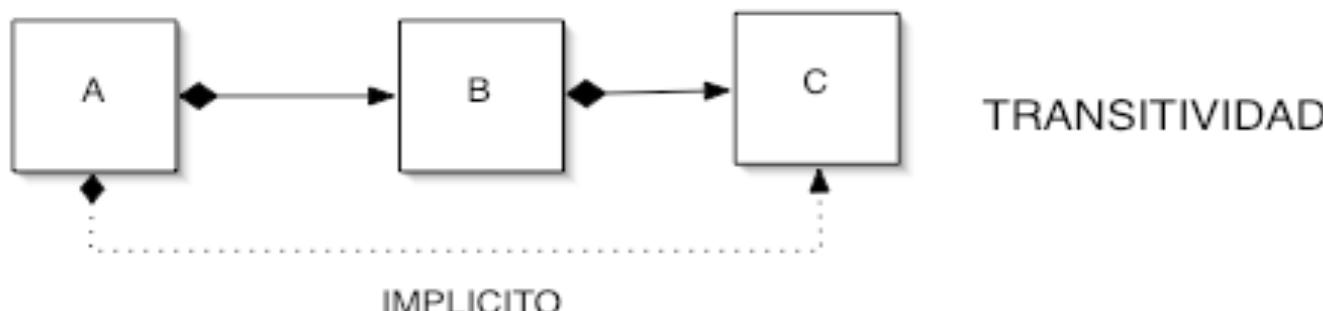
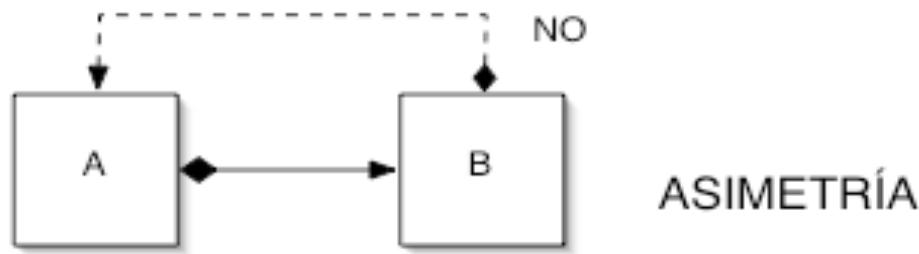
- A partir del dibujo de la Fig., define las clases Trabajador y Proyecto
 - Un trabajador debe trabajar siempre como mínimo en un proyecto, y dirigir un máximo de dos
 - Un proyecto tiene n trabajadores, y siempre debe tener un director



Todo-Parte



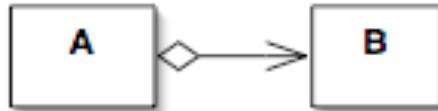
- Una relación Todo-Parte es una relación en la que un objeto forma parte de la naturaleza de otro.
 - Se ve en la vida real: 'A está compuesto de B', 'A tiene B'
 - (en inglés, relaciones 'has-a')
- Asociación vs. Todo-Parte
 - La diferencia entre asociación y relación todo-parte radica en la **asimetría** y **transitividad** presentes en toda relación todo-parte.





- Se distinguen dos tipos de relación todo-parte:

- **Agregación**



- Asociación binaria que representa una relación todo-parte ('tiene-un', 'es-parte-de', 'pertenece-a')
 - Ejemplo: Un equipo y sus miembros

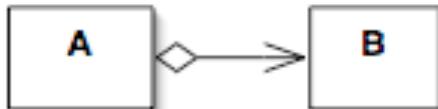
- **Composición**



- Agregación 'fuerte':
 - Una instancia 'parte' está relacionada, como máximo, con una instancia 'todo' en un instante dado.
 - Cuando un objeto 'todo' es eliminado, también son eliminados sus objetos 'parte' (Es responsabilidad del objeto 'todo' disponer de sus objetos 'parte')
 - Ejemplo: Un libro y sus capítulos



¿Agregación o composición?



¿Puede el objeto parte ser compartido por más de un objeto agregado?

- No => **disjunta** (Composición)
- Sí => no disjunta (Agregación)

¿Puede existir un objeto parte sin ser componente de un objeto agregado?

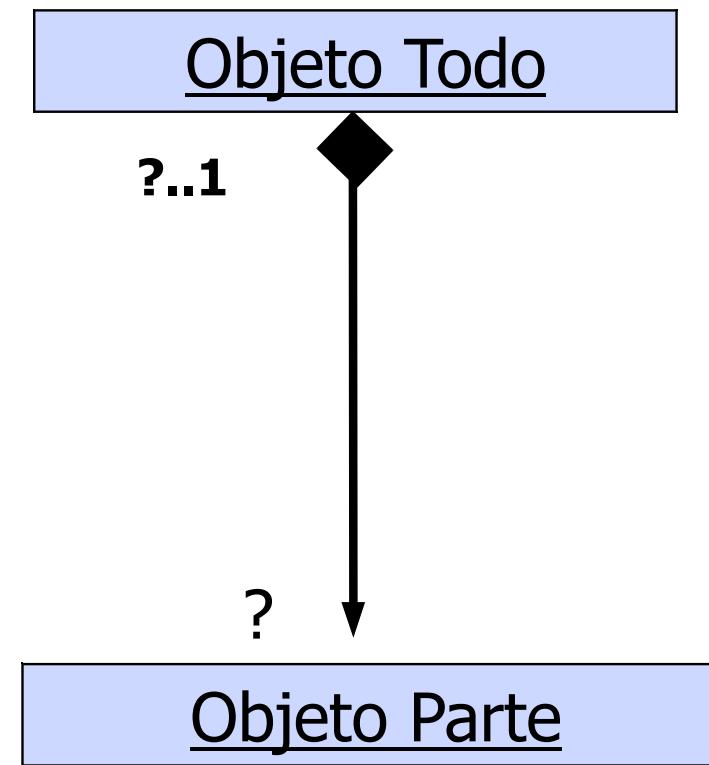
- Sí => **flexible** (Agregación)
- No => **estricta** (Composición)

Relaciones entre Objetos

Caracterización composición



- Si tenemos en cuenta lo anterior, una composición se caracteriza por una cardinalidad máxima de 1 en el objeto compuesto (el todo).

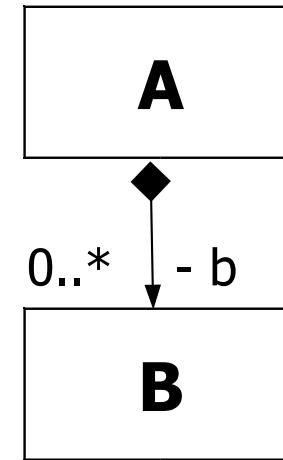
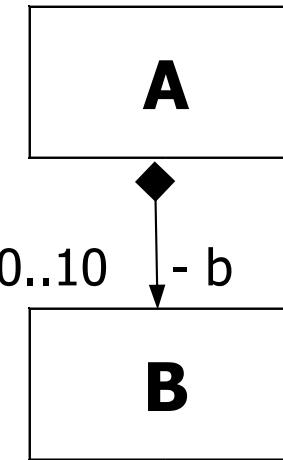
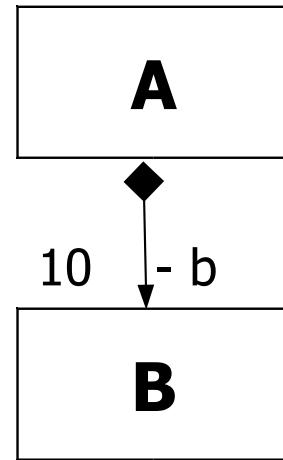
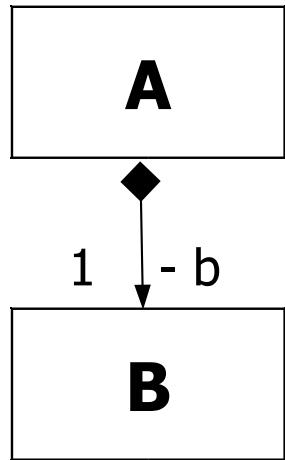




- Agregación: Se implementa como una asociación unidireccional
 - El objeto 'todo' mantiene referencias (posiblemente compartidas con otros objetos) a sus partes agregadas.
- Composición: Dijimos que
 - Es responsabilidad del objeto 'todo' disponer de sus objetos 'parte'...

Relaciones todo-parte

Implementación (Java)



```
class A {  
    private B b;  
    // b es un  
    // subobjeto  
...}
```

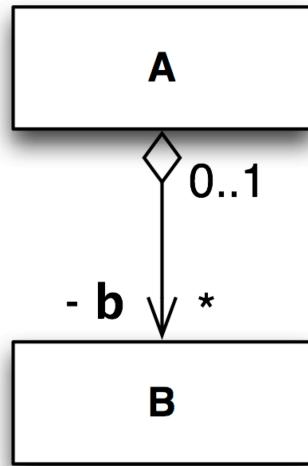
```
class A {  
    private static final int MAX_B = 10;  
    private B b[] = new B[MAX_B];  
...}
```

```
class A {  
    private  
    Vector<B> b;  
...};
```

La declaración de atributos es la misma para una agregación o una composición.

Relaciones todo-parte

Implementación de la agregación



A) El objeto B puede ser creado fuera de A, de forma que pueden existir referencias externas ('objB') al objeto agregado.

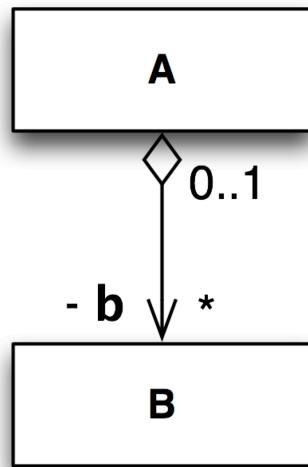
```
class A {  
    private Vector<B> b = new Vector<B>();  
  
    public A() {}  
    public addB(B unB) {  
        b.add(unB);  
    }  
    ...}
```

```
// En otro lugar (código cliente),  
// quizás fuera de A...  
B objB = new B();  
if (...) {  
    A objA = new A();  
    objA.addB(objB);  
}
```

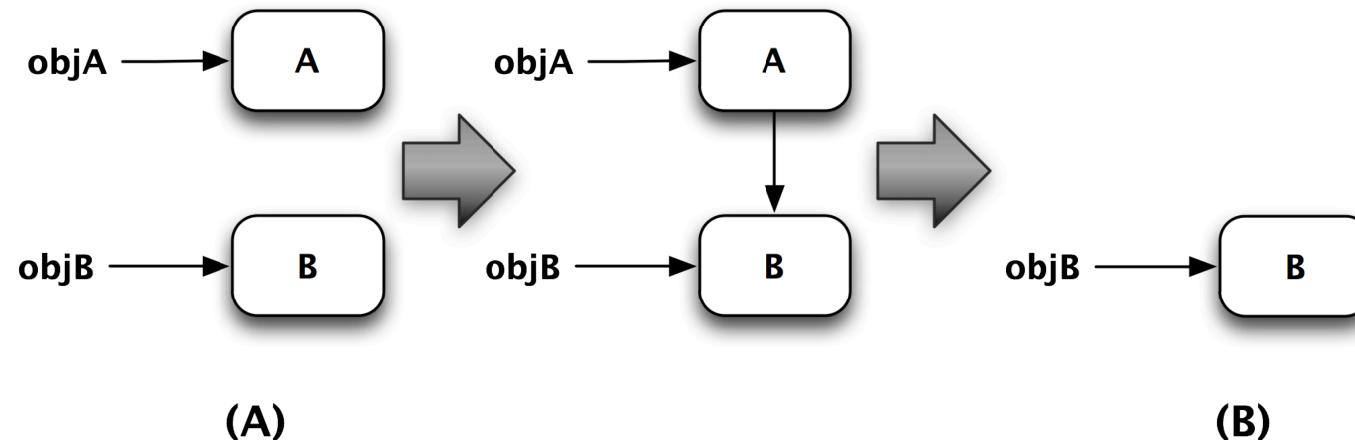
B) Cuando 'objA' desaparece, 'objB' sigue existiendo

Relaciones todo-parte

Implementación de la agregación

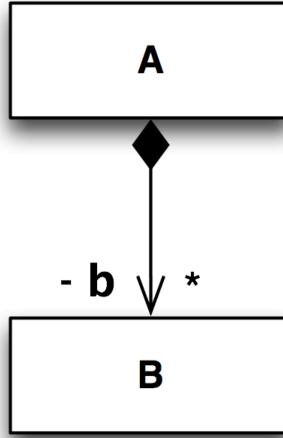


- A) El objeto B puede ser creado fuera de A, de forma que pueden existir referencias externas ('objB') al objeto agregado.
- B) Cuando 'objA' desaparece, el objeto B sigue existiendo, pues aún hay referencias a él ('objB').



Relaciones todo-parte

Implementación de la composición



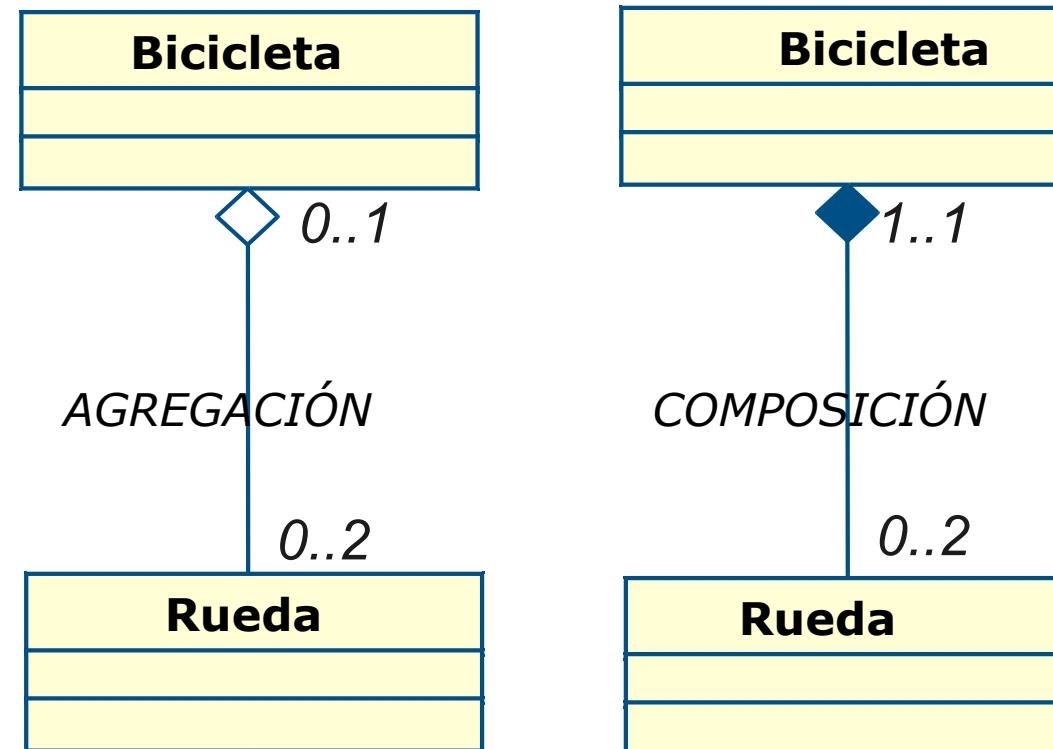
A) El objeto B es creado 'dentro' de A, de forma que A es el único que mantiene referencias a su componente B.

```
class A {  
    private Vector<B> b = new Vector<B>();  
  
    public A() {}  
    public addB(...) {  
        b.add(new B(...));  
    } '...' es la información necesaria  
    para crear B  
}  
...  
// En otro lugar (código cliente),  
// fuera de A...  
  
if (...) {  
    A objA = new A();  
    objA.addB(...);  
}
```

B) Cuando 'objA' desaparece, también desaparecen los objetos B que forman parte de él

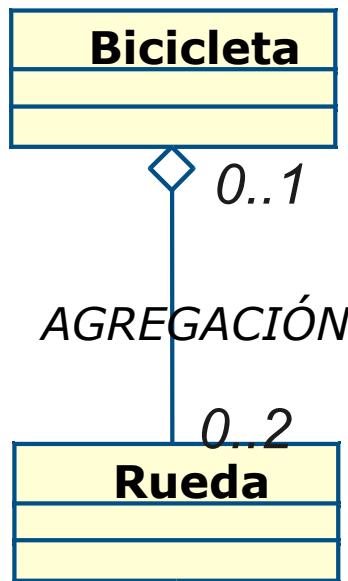


Algunas relaciones pueden ser consideradas agregaciones o composiciones, en función del contexto en que se utilicen



Relaciones todo-parte

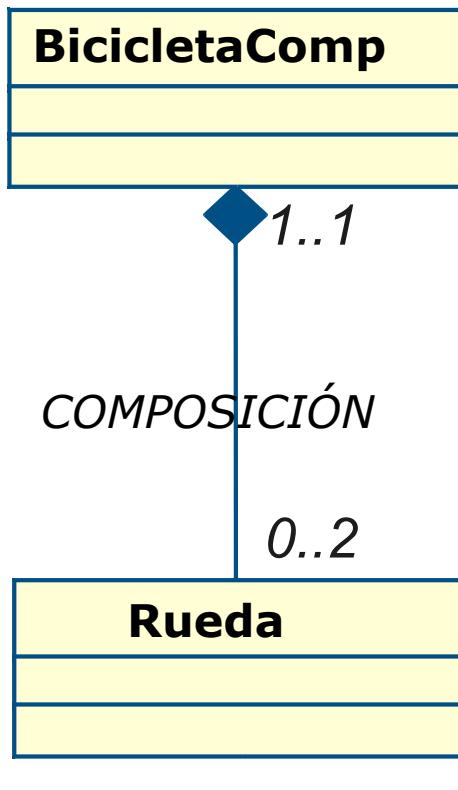
Ejemplo Bicicleta



```
class Rueda {  
    private String nombre;  
    public Rueda(String n) {nombre=n; }  
}  
  
class Bicicleta {  
    private Vector<Rueda> r;  
    private static final int MAXR=2;  
    public Bicicleta(Rueda r1, Rueda r2) {  
        r.add(r1);  
        r.add(r2);  
    }  
    public void cambiarRueda(int pos, Rueda raux) {  
        if (pos>=0 && pos<MAXR)  
            r.set(pos,raux);  
    }  
  
    public static final void main(String[] args)  
    {  
        Rueda r1=new Rueda("1");  
        Rueda r2=new Rueda("2");  
        Rueda r3=new Rueda("3");  
        Bicicleta b(r1,r2);  
        b.cambiarRueda(0,r3);  
    }  
}
```

Relaciones todo-parte

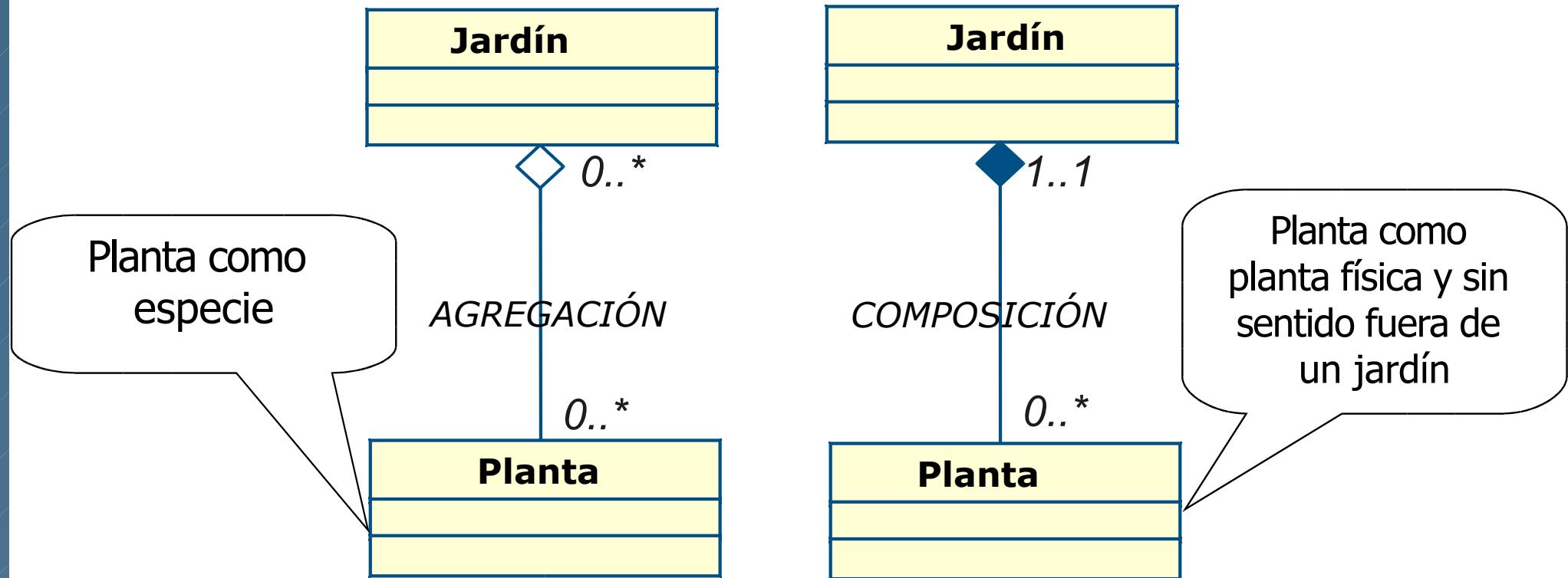
Ejemplo Bicicleta



```
class BicicletaComp{  
    private static const int MAXR=2;  
    private Vector<Rueda> r;  
    public BicicletaComp(String p, String s) {  
        r.add(new Rueda(p));  
        r.add(new Rueda(s));  
    }  
    public static final void main(String[] args) {  
        BicicletaComp b2 = new BicicletaComp("1", "2");  
        BicicletaComp b3 = new BicicletaComp("1", "2");  
        //son ruedas distintas aunque con el mismo nombre  
    }  
}
```



Observad el diferente significado de la clase Planta en función del tipo de relación que mantiene con Jardín



Relaciones todo-parte

Ejemplo Jardín



Supongamos que tenemos el código

```
class Planta {  
    public Planta(String n, String e)  
    {...}  
  
    public String getNombre() {...}  
    public String getEspecie() {...}  
    public String getTemporada() {...}  
    public void setTemporada(String t)  
    {...}  
  
    private String nombre;  
    private String especie;  
    private String temporada;  
}
```

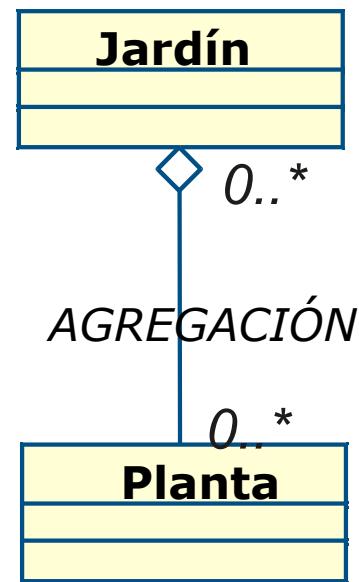
```
class Jardin {  
    public Jardin(String e) {...}  
  
    public void plantar(String n, String e,  
                        String t)  
    {  
        Planta lp = new Planta(n, e);  
        lp.setTemporada(t);  
        p.add(lp);  
    }  
  
    private Vector<Planta> p  
        = new Vector<Planta>();  
    private String emplazamiento;  
}
```



¿Qué relación existe entre Jardín y planta?



¿Cómo cambiaría el código si decidiésemos implementar el jardín como una agregación de plantas?

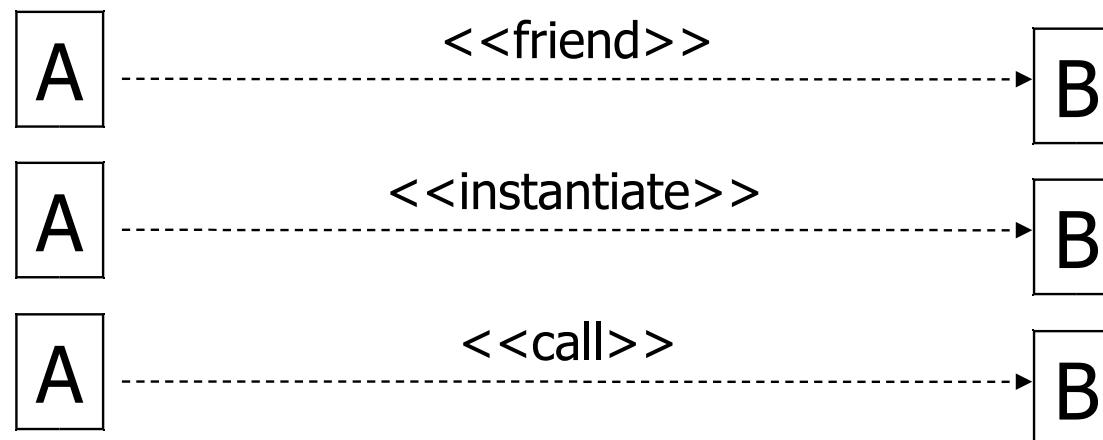


Relaciones entre Objetos

Relación de Uso (Dependencia)



- Una clase A **usa** una clase B cuando no contiene datos miembros del tipo especificado por la clase B pero:
 - Utiliza alguna **instancia de la clase B como parámetro** (o variable local) en alguno de sus métodos para realizar una operación.
 - **Accede a sus variables privadas** (clases amigas)
 - Usa algún **método de clase** de B.
- En UML este tipo de relaciones se diseñan mediante **dependencias**.



Relaciones entre Objetos

Relación de Uso (Dependencia): Ejemplo



- Supongamos que no nos interesa guardar las gasolineras en las que ha repostado un coche: no es asociación.
- Sin embargo sí existe una interacción:

```
class Coche {  
    Carburante tipoC;  
    float lGasos;  
    float repostar(Gasolinera g, float litros) {  
        float importe=g.dispensarGasos(litros, tipoC);  
        if (importe>0.0) //si éxito dispensar  
            lGasos=lGasos+litros;  
        return (importe);  
    }  
}
```

Bibliografía



- Cachero et. al.
 - ***Introducción a la programación orientada a Objetos***
 - Capítulo 2
- T. Budd.
 - ***An Introduction to Object-oriented Programming, 3rd ed.***
 - Cap. 4 y 5; cap. 6: caso de estudio en varios LOO
- G. Booch.
 - ***Object Oriented Analysis and Design with Applications***
 - Cap. 3 y 4
- G. Booch et. al.
 - ***El lenguaje unificado de modelado.*** Addison Wesley. 2000
 - Sección 2 (cap. 4-8)

UD 3

INTRODUCCIÓN AL DISEÑO ORIENTADO A OBJETOS

Pedro J. Ponce de León

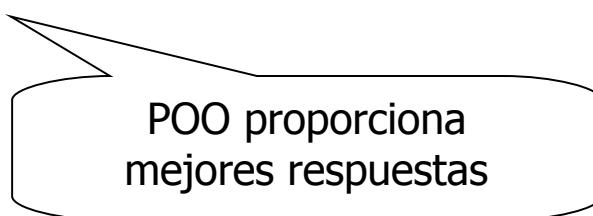
Versión 0.1
(Curso 11/12)



- Diseño O.O.
 - Principios de Parnas
 - Acoplamiento y cohesión
 - Diseño dirigido por responsabilidades (RDD)
 - Principios y artefactos
 - Modelado de objetos
 - Tarjetas CRC
 - Ejercicios de diseño con CRC

- Objetivo principal: conseguir crear un universo de objetos lo más independientes posible entre sí.
 - Una posible técnica a utilizar es la denominada **Diseño Dirigido por Responsabilidades** (*responsibility-driven design*) [Wirfs-Brock].

- Pequeños proyectos : 'Programming in the small'
 - Pocos programadores. Un individuo puede abarcar todos los aspectos del proyecto.
 - El mayor problema es el diseño y desarrollo de algoritmos para resolver el problema actual.
- Grandes proyectos: 'Programming in the large' :
 - Gran equipo de programadores. Un individuo no puede hacerse responsable ni es capaz de entender todo el proyecto
 - El mayor problema en el proceso de desarrollo es el manejo de detalles y la comunicación entre las distintas porciones del proyecto



POO proporciona mejores respuestas

El diseño de aplicaciones OO

Interfaz e Implementación



- El énfasis en caracterizar un componente software por su comportamiento tiene una consecuencia fundamental: separación de interfaz (qué) e implementación (cómo).
- ***Principios de Parnas***
 - El desarrollador de un componente sw C debe proporcionar al usuario de C toda la info necesaria para hacer un uso efectivo de los servicios de C y no debería proporcionar ninguna otra información.
 - El desarrollador de un componente sw C debe recibir toda la información necesaria para realizar las responsabilidades necesarias asignadas al componente y ninguna otra información.

El diseño de aplicaciones OO

Métricas de calidad



- **Acoplamiento:** relación entre componentes software
 - Interesa bajo acoplamiento. Éste se consigue moviendo las tareas a quién ya tiene habilidad para realizarlas.
-
- **Cohesión:** grado en que las responsabilidades de un solo componente forman una unidad significativa.
 - Interesa alta cohesión. Ésta se consigue asociando a un solo componente tareas que están relacionadas en cierta manera, p. ej., acceso a los mismos datos.

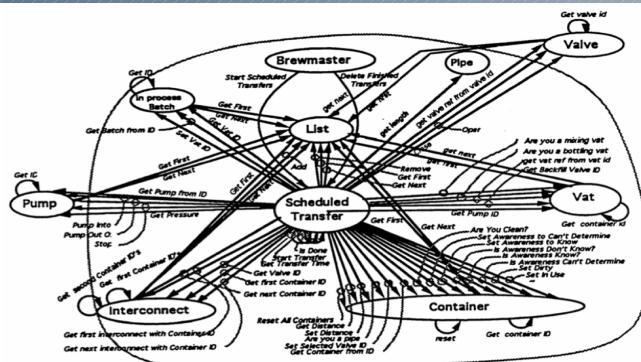
El diseño de aplicaciones OO

Métricas de calidad



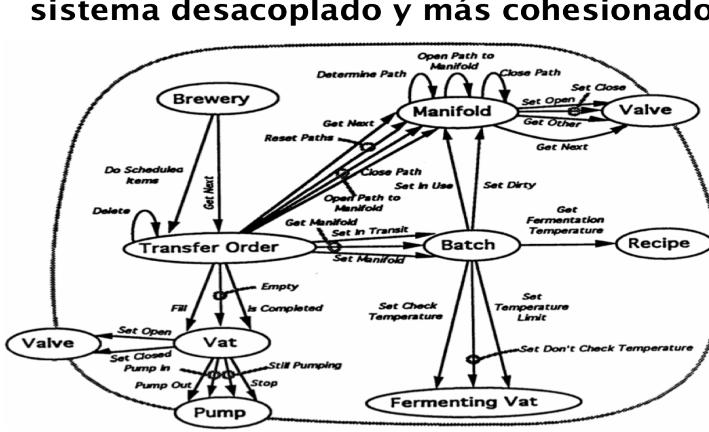
- Uno de los principios básicos de la ingeniería del software es **“incrementar la cohesión, reducir el acoplamiento”**.
- Componentes con bajo acoplamiento y alta cohesión facilitan su utilización y su interconexión.

Acoplamiento vs. cohesión



sistema demasiado acoplado

Acoplamiento

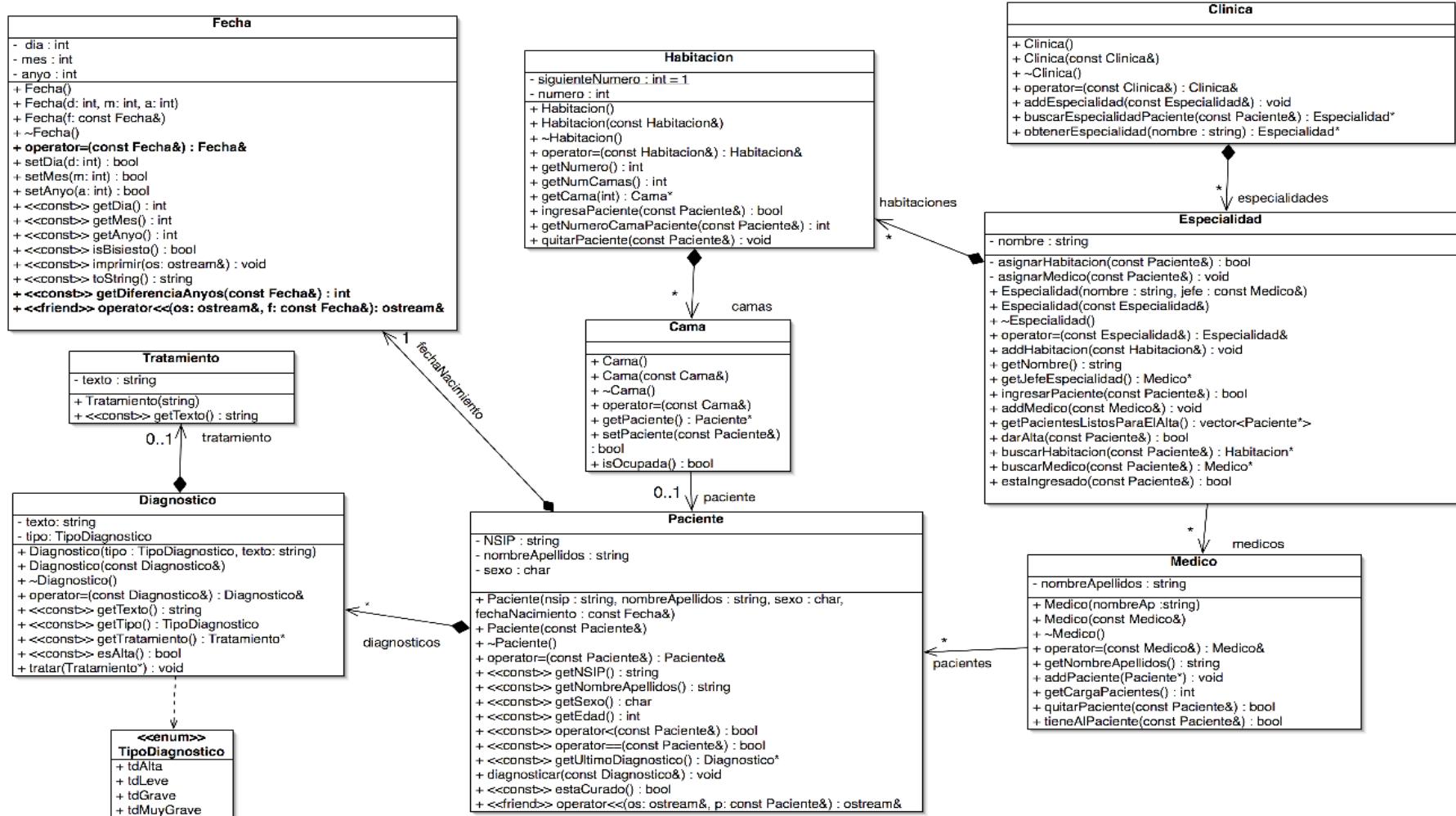


sistema desacoplado y más cohesionado

- Un diagrama de clases define las clases, sus propiedades y cómo se relacionan unas con otras.
- Proporciona una vista **estática** de los elementos que conforman el software.
 - Se ven las partes que componen la aplicación y cómo se ensamblan, pero no cómo se comportan cuando el sistema se ejecuta.
- El diagrama de clases es el diagrama principal de análisis y diseño.

El diseño de Aplicaciones OO

Diagrama de Clases (UML)



Responsibility-Driven Design (RDD)

- Proporciona técnicas informales para el modelado de roles, responsabilidades y colaboraciones de objetos.

Ejemplo: diseñar un caballo

- Objetos: Cabeza, cuerpo, cola, patas (4)
- Comportamiento: Arrancar, parar, acelerar, decelerar, ...

Diseñar un caballo con responsabilidad:

- ¿Para qué queremos al caballo?
 - Para pasear
 - Para competir
 - Para rejonear

- **Maximizar la abstracción**

Pensar en las responsabilidades de los objetos de ‘conocer/saber’, ‘hacer’ y ‘decidir’: ¿quién es responsable de saber tal o cual cosa? ¿Quién es responsable de hacer esto o lo otro? Para esto necesitamos a un experto en el dominio (podemos ser nosotros mismos)

- **Distribuir el comportamiento**

Promueve una arquitectura de control delegado (que una tarea determinada la haga quien sabe hacerlo. Por ejemplo, ¿quién es responsable de imprimir una fecha en formato DD/MM/AAAA?)

- **Crear objetos ‘inteligentes’**,
que sepan hacer ciertas cosas, que no sean sólo un capazo donde guardar datos.

- **Preservar la flexibilidad**
Flexibilidad: capacidad de algo de ser modificado y adaptado a los cambios del entorno.

Diseña objetos de forma que los detalles internos se puedan modificar fácilmente, sin afectar a otros objetos o al sistema.

- **Aplicación** = un conjunto de objetos interactivos
- **Objeto** = una implementación de uno o más roles
- **Rol** = un conjunto de responsabilidades
- **Responsabilidad** = la obligación de realizar una tarea o conocer cierta información
- **Colaboración** = una interacción entre objetos o roles (o ambos)

- **Modelado de objetos:**

1. Identificar las clases de objetos que forman parte del sistema
2. Asignar responsabilidades a cada clase de objeto

- Consejo:

Describe el sistema a modelar como si contaras una historia/escribieras un artículo

- Identifica los temas importantes
- Encuentra candidatos/protagonistas
- Identifica los **sustantivos** y **verbos** de la descripción del sistema que pueden corresponderse con objetos y su comportamiento.

- **Pautas para modelar objetos**

- Busca los conceptos clave del dominio: conceptos familiares para los expertos en el dominio/problema a resolver
- Elige nombres con sentido
- Distingue objetos con diferentes comportamientos.
- Coloca a los objetos en su contexto

El diseño de Aplicaciones OO

RDD: Modelado de objetos (CRC)



Asignar NOMBRES a los objetos

¡Esto es importante!

- El nombre debería encajar con el rol del objeto en el sistema.
Por ej. un proveedor de servicio es algo que realiza un trabajo: StringTokenizer, ClassLoader, Authenticator.
- El nombre debe dar al programador que utilizará esos objetos una idea suficientemente clara de lo que el objeto sabe hacer (sin entrar en detalles):
TemporizadorConUnaPrecisionDeMasMenosDosMilisegundos → (mejor simplemente Temporizador)
- Capitalización o subrayado para separar palabras:
LectorDeTarjetas o Lector_de_tarjetas
 - Ojo con las abreviaturas, no todo el mundo las entiende y pueden confundir:
TermProcess

Asignar NOMBRES a los objetos

- Evita nombres con varias interpretaciones posibles:
`vacio()` : ¿nos dice si el objeto está vacío o lo vacía?
- No uses dígitos
- Booleanos: usa nombres que permitan interpretar claramente su valor:
`ImpresoraPreparada` es mejor que `EstadoImpresora`.
- Un buen lugar para ver ejemplos de nombres informativos es el API de Java.
Úsalos como modelo.

El diseño de Aplicaciones OO

RDD: Modelado de objetos (CRC)



Ejercicio: Intenta averiguar (sin consultar el API de Java) qué hacen estos tipos de objetos :

ProcessBuilder
SecurityManager
StringBuffer
ArithmaticException
NullPointerException
NoClassDefFoundError
Comparator
EventListener
SortedMap
PropertyPermission
ImageReader
ImageWriter
ContainerOrderFocusTraversalPolicy (AWT)
MenuShortcut

Selecciona candidatos que puedes

- Nombrar
- Definir su propósito
- Asignarle una o dos responsabilidades
- Comprender cómo los demás lo ven

El diseño de Aplicaciones OO

RDD: Asignación de responsabilidades (CRC)



Una **responsabilidad** es algo que una clase conoce o hace.

P. ej.: Un Estudiante conoce su nombre, su DNI, su domicilio. Un Estudiante se matricula en asignaturas, se presenta a exámenes, solicita certificados, ...

Las responsabilidades son algo más que una sola operación o atributo: a menudo involucran varias operaciones y/o manejan varios atributos o items.

Usa descripciones claras y 'contundentes': Usa verbos con un significado claro: 'eliminar', 'mezclar', 'calcular', 'activar', en lugar de otros como 'organizar', 'mantener', 'procesar', 'aceptar'...

El diseño de Aplicaciones OO

RDD: Asignación de responsabilidades (CRC)



Consejos para asignar responsabilidades

- Mantener el comportamiento junto a la información que maneja en el mismo objeto
- No crear objetos (roles) demasiado grandes. Esto los hace más comprensibles
- Distribuye la inteligencia del sistema
- Una misma responsabilidad puede ser implementada por uno o más métodos
- DELEGA: si es posible, haz que otros objetos hagan parte del trabajo.

El diseño de Aplicaciones OO

RDD: Identificación de colaboradores (CRC)



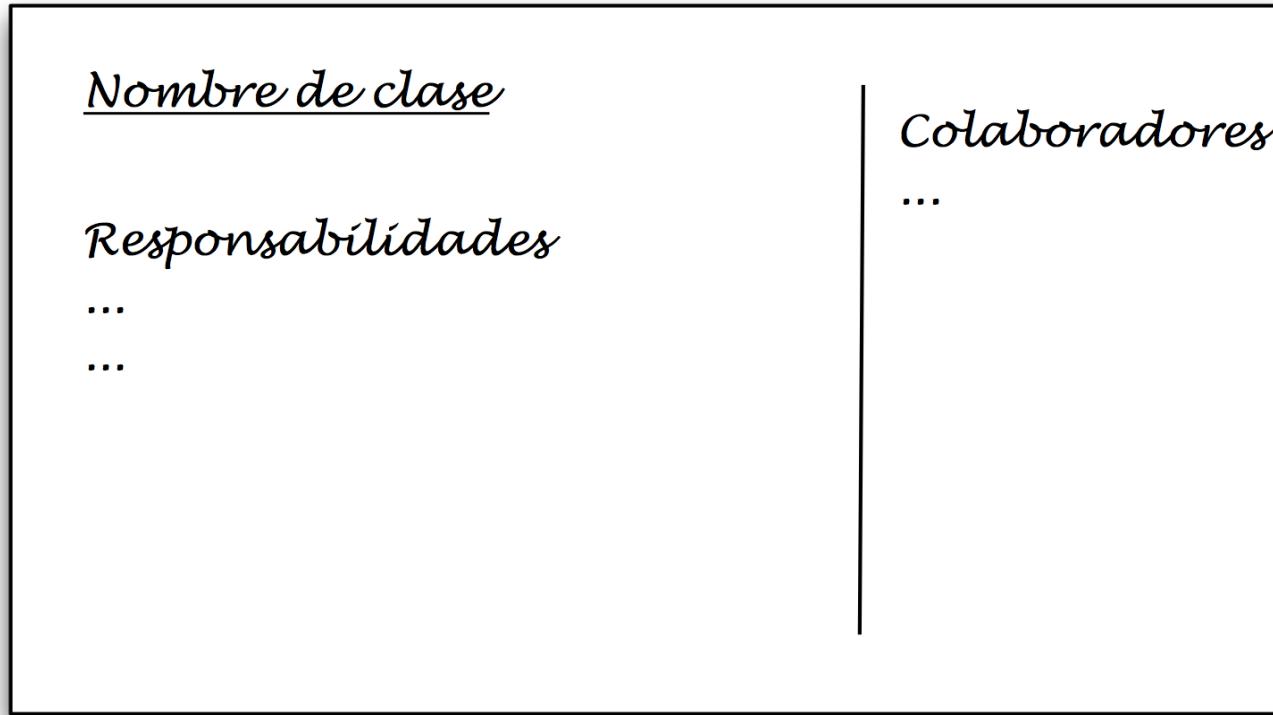
Colaboración:

Como un grupo de objetos trabaja conjuntamente para realizar una tarea.

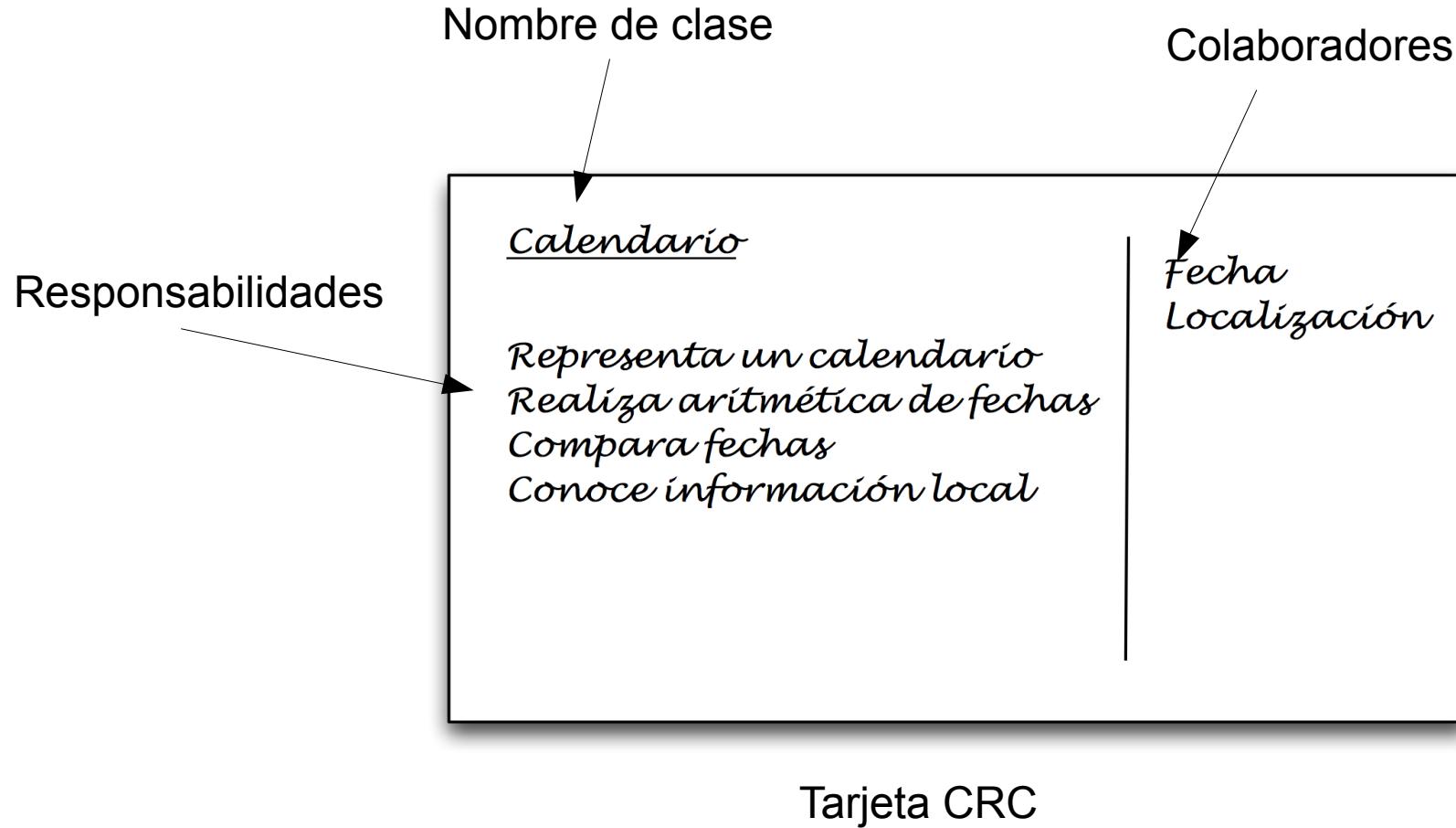
La lista de colaboradores de una clase debe incluir todas aquellas clases que necesita conocer.

- Obligatoriamente, las que suministran servicios que la clase actual necesita para llevar a cabo alguna de sus responsabilidades
- Opcionalmente, aquellas que utilizan servicios proporcionados por la clase actual.

CRC: Class, Responsibility, Collaborators (Clase, Responsabilidades, Colaboradores)

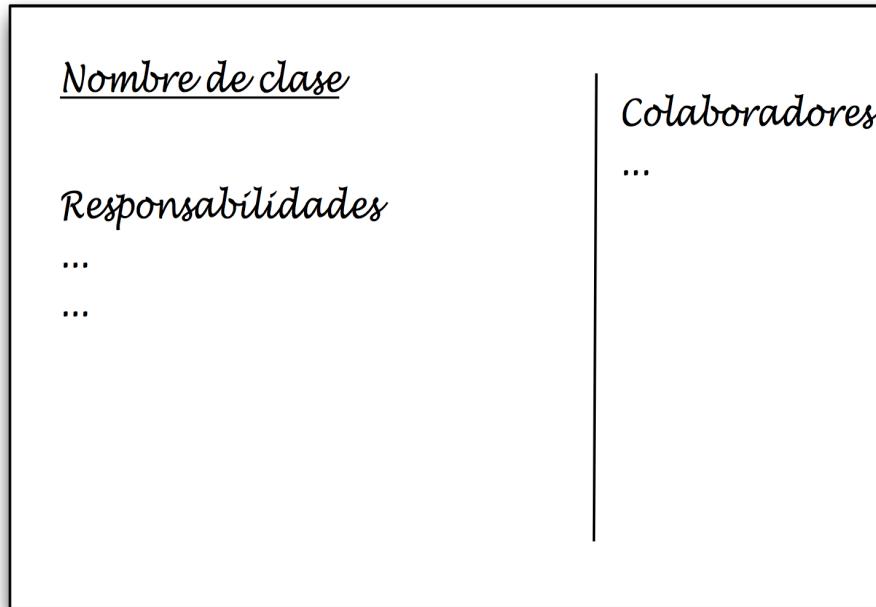


Toda la información sobre una clase de objetos es escrita sobre una tarjeta índice, de las usadas (antiguamente) en las bibliotecas.



El diseño de Aplicaciones OO

Tarjetas CRC



Nombre de la clase: crea un vocabulario para discutir un diseño. Debemos encontrar un conjunto nombres consistente y evocativo que describa a nuestros objetos en el contexto de nuestra aplicación.

Las **responsabilidades** identifican tareas que se deben resolver. Se expresan mediante un puñado de frases simples, cada una con un verbo activo.

Colaboradores: "no object is an island" Todos los objetos establecen relaciones con otros. Llamamos objetos colaboradores a aquellos que recibirán mensajes o enviarán mensajes al objeto actual, con objeto de satisfacer una responsabilidad.

Modelado de objetos con tarjetas CRC

Las tarjetas CRC enfatizan la cohesión y el (des)acoplamiento. El tamaño de la tarjeta es una buena apoximación a la complejidad que debería tener un objeto.

Las tarjetas se pueden

Apilar: partes debajo del todo

Disponer unas cerca de otras (colaboradores cercanos)

Disponerlas por capas

...

En definitiva, hacer lo que consideres conveniente para ilustrar con ellas cómo se comportan los objetos

Modelado de objetos con tarjetas CRC

Es muy útil plantearse escenarios de ejecución, donde imaginamos al sistema a diseñar en ejecución

- 1. ¿Qué es lo que hay que hacer?**
- 2. ¿Quién tiene que hacerlo?**

El diseño de Aplicaciones OO

Ejercicio de diseño con tarjetas CRC



Sistema de venta por catálogo.

Queremos crear una aplicación para un sistema de venta por catálogo. El catálogo contiene una lista de productos con su nombre, una descripción, su precio y una referencia de catálogo. Nos interesa que el sistema lleve un control del stock actual, para saber si un producto determinado se puede vender en el momento que se pide o no. Los clientes acuden a nuestra tienda, consultan el catálogo y nos indican qué productos del catálogo quieren comprar y en qué cantidad. Una vez comprobado que hay stock suficiente de cada producto, emitimos un ticket con el detalle de la compra y su importe total, el cual entregamos al cliente junto a la lista de productos solicitados.

El diseño de Aplicaciones OO

Ejercicio de diseño con tarjetas CRC



Cajero automático (dispensador de billetes)

Queremos crear una aplicación que gestione un cajero automático en el que únicamente se puede sacar dinero mediante tarjeta de débito. Cuando un cliente usa el cajero, lo primero que hace es introducir su tarjeta bancaria. El sistema le pide la contraseña y, si es correcta, le solicita a continuación la cantidad de dinero que desea retirar (sólo se permiten múltiplos de 5). Cada tarjeta tiene asociado un límite diario para sacar dinero con ella. Si la cantidad solicitada es inferior a dicho límite, el cajero comprueba que el saldo de la cuenta bancaria asociada a la tarjeta es suficiente para deducir de ella el dinero a retirar. En caso afirmativo, se deduce la cantidad solicitada del saldo de la cuenta y se le entrega al cliente la cantidad en billetes de 50,20,10 y/o 5 euros, a través del dispensador de billetes, y un justificante de la operación, que muestra, además de la cantidad retirada, el saldo restante en la cuenta y en el límite diario de la tarjeta. Una vez el cliente ha retirado el dinero y el justificante, se le devuelve la tarjeta. Si el cliente tarda más de un minuto en retirar el dinero, este se vuelve a introducir en el cajero y se anula la operación. Si, una vez retirado el dinero, el cliente tarda más de un minuto en retirar la tarjeta, esta se vuelve a introducir en el cajero, quedando custodiada en él para su posterior reclamación por parte del cliente.

Bibliografía



-
- T. Budd. ***An Introduction to Object-oriented Programming, 3rd ed.***
 - Cap. 3
- Rebecca Wirfs-Brock, **A Brief Tour of Responsibility-Driven Design**
 - http://wirfs-brock.com/PDFs/A_Brief-Tour-of-RDD.pdf
- Kent Beck, **A Laboratory for Teaching Object-Oriented Thinking**
 - <http://c2.com/doc/oopsla89/paper.html>

PROG3

UD 4

GESTIÓN DE ERRORES

Pedro J. Ponce de León

Versión 20111005



Gestión Errores

Objetivos

- **Saber utilizar las sentencias de control de excepciones para observar, indicar y manejar excepciones, respectivamente.**
- **Comprender las ventajas del manejo de errores mediante excepciones frente a la gestión de errores tradicional de la programación imperativa.**
- **Comprender la jerarquía de excepciones estándar.**
- **Ser capaz de crear excepciones personalizadas**
- **Ser capaz de procesar las excepciones no atrapadas y las inesperadas.**
- **Comprender la diferencia entre excepciones verificadas y no verificadas**

Indice

- Motivación
- Excepciones
 - Concepto
 - Lanzamiento y captura
 - Especificación de excepciones
- Excepciones estándar en Java
- Excepciones de usuario
- Particularidades de las excepciones
- Regeneración de excepciones

Gestión de Errores

Motivación

- Gestión de errores 'tradicional' (o al estilo de C)

```
int main (void)
{
    int res;
    if (puedo_fallar () == -1)
    {
        cout << "¡Algo falló!" << endl;
        return 1;
    }
    else cout << "Todo va bien..." << endl;
    if (dividir (10, 0, res) == -1)
    {
        cout << "¡División por cero!" << endl;
        return 2;
    }
    else cout << "Resultado: " << res << endl;
    return 0;
}
```

Flujo normal
Flujo de error

Gestión de Errores

Motivación

- Nos obliga a definir un esquema de programación similar a :
 - Llevar a cabo tarea 1
 - *Si se produce error*
 - *Llevar a cabo procesamiento de errores*
 - Llevar a cabo tarea 2
 - *Si se produce error*
 - *Llevar a cabo procesamiento de errores*
- A esto se le llama código ***espaguetti***
- Problemas de esta estrategia
 - Entremezcla la lógica del programa con la del tratamiento de errores (disminuye legibilidad)
 - El código cliente (llamador) no está obligado a tratar el error
 - Los 'códigos de error' no son consistentes.

Gestión de Errores

Motivación

- ¿Qué podemos hacer en lugar de crear código *spaghetti*?
 - Abortar el programa
 - ¿Y si el programa es crítico?
 - Usar indicadores de error globales
 - El código cliente (llamador) no está obligado a consultar dichos indicadores.
 - USAR EXCEPCIONES
 - La idea básica es
 - Cuando no disponemos de información suficiente para resolver el problema en el contexto actual (método o ámbito), lo indicamos a nuestro llamador mediante una excepción.

Gestión de Errores

Excepciones: Concepto

- Una **excepción** es un evento que ocurre durante la ejecución del programa que interrumpe el flujo normal de las sentencias.
- Son objetos (instancias de clases) que contienen información sobre el error.
- Las excepciones se tratan mediante sentencias de control del flujo de error que separan el código para manejar errores del resto mediante :
 - **throw, try, catch, finally**
- Por defecto, una excepción no se puede ignorar: hará que el programa aborte.
 - Una excepción pasará sucesivamente de un método a su llamador hasta encontrar un bloque de código que la trate.

Gestión de Errores

Excepciones: **Comportamiento**

- Las excepciones son **lanzadas** (`throw`) por un método cuando éste detecta una condición excepcional o de error.
- Esto interrumpe el control normal de flujo y provoca (si el propio método no la trata) la finalización prematura de la ejecución del método y su retorno al llamador.
- Las excepciones pueden ser **capturadas** (`try/catch`), normalmente por el código cliente (llamador).
- Si el llamador no captura la excepción, su ejecución también terminará y la excepción 'saldrá' de nuevo hacia un ámbito más externo y así sucesivamente hasta encontrar un lugar donde es capturada.
- Una excepción no capturada provocará que el programa aborte.

Gestión de Errores

Excepciones: Sintaxis Java

```
void Func() {  
  
    if(detecto_error1) throw new Tipo1();  
    ...  
    if(detecto_error2) throw new Tipo2();  
    ...  
}
```

constructor

```
try  
{  
    // Código de ejecución normal  
    Func(); // puede lanzar excepciones  
}  
catch (Tipo1 ex)  
{  
    // Gestión de excepción tipo 1  
}  
catch (Tipo2 ex)  
{  
    // Gestión de excepción tipo 2  
}  
finally {  
    // se ejecuta siempre  
}  
// continuación del código
```

Gestión de Errores

Excepciones: Sintaxis C++

```
void Func() {  
  
    if (detecto_error1) throw Tipol();  
    ...  
    if (detecto_error2) throw Tipo2();  
    ...  
}
```

constructor

```
try  
{  
    // Código de ejecución normal  
    Func(); // puede lanzar excepciones  
    ...  
}  
catch (Tipol &ex)  
{  
    // Gestión de excepción tipo 1  
}  
catch (Tipo2 &ex)  
{  
    // Gestión de excepción tipo 2  
}  
catch (...)  
{  
    /* Gestión de cualquier excepción no  
       capturada mediante los catch  
       anteriores */  
}  
//Continuación del código
```

Gestión de Errores

Excepciones: **Sintaxis**

- En C++ y en Java:
 - El bloque **try** contiene el código que forma parte del funcionamiento normal del programa.
 - El bloque **catch** contiene el código que gestiona los diversos errores que se puedan producir.
- Sólo en JAVA:
 - Se utiliza el bloque **finally** para cerrar ficheros o liberar otros recursos del sistema. El bloque finally se puede ejecutar:
 - (1) tras finalizar el bloque try o
 - (2) después de las cláusulas catch.

Gestión de Errores

Excepciones: Funcionamiento

- Funcionamiento:
 1. Ejecutar instrucciones try
 - Si hay error, interrumpir el bloque try e ir a bloque catch correspondiente
 2. Continuar la ejecución después de los bloques catch
- La excepción es capturada por el bloque **catch** cuyo argumento coincide con el tipo de objeto lanzado por la sentencia **throw**. La búsqueda de coincidencia se realiza sucesivamente sobre los bloques **catch** en el orden en que aparecen en el código hasta que aparece la primera concordancia.
 -
- En caso de no existir un manejador adecuado a una excepción determinada, se desencadena un protocolo que, por defecto, produce sin más la finalización del programa. (En Java la excepción es capturada y mostrada por la máquina virtual).

Gestión de Errores

Excepciones: **Lanzamiento**

- La cláusula **throw** lanza una excepción
 - Una excepción es un objeto.
 - La clase **Exception** (Java) representa una excepción general.

Lanzamiento

```
int LlamameConCuidado(int x) {  
    if (condicion_de_error(x) == true)  
        throw new Exception("valor "+x+" erroneo");  
    //... código a ejecutar si no hay error ...  
}
```

Llamada
a constructor

Gestión de Errores

Excepciones: **Captura**

- La instrucción **catch** es como una llamada a función: recibe un argumento.

Captura

```
public static void main() {  
    try {  
        LlamameConCuidado(-0);  
    } catch (Exception ex) {  
        System.err.println("No tuviste cuidado: " + ex);  
        ex.printStackTrace();  
    }  
}
```

Gestión de Errores

Excepciones: **especificación de excepción**

- En Java y C++ un método puede indicar, en su declaración, qué excepciones puede lanzar (directa o indirectamente) mediante la **especificación de excepción**.
- Este especificador se utiliza en forma de sufijo en la declaración de la función y tiene la siguiente **sintaxis** en Java:

```
throws (<lista-de-tipos>)
```

- <lista-de-tipos> indica qué excepciones puede lanzar el método.

```
public int f() throws E1, E2 { ... }
```

f() puede lanzar excepciones de tipo E1 o E2.

Gestión de Errores

Excepciones: Uso correcto

```
void f() {  
    try { g(); } catch(Exception ex) {  
        System.err.println(ex.queHaPasado());  
    }  
    // sigue...  
}
```

f() captura el tipo de excepciones que puede lanzar h()

```
void g() {  
    h();  
    // sigue...  
}
```

g() no la captura...

```
void h() {  
    if (algo_fallar)  
        throw new Exception("¡Mecachis!");  
    // sigue...  
}
```

h() puede lanzar una excepción...

Uso correcto de las excepciones

- Un método que lanza excepciones se limita a señalar que se ha producido algún tipo de error, pero, por regla general, no debe tratarlo. Delegará dicho tratamiento en quienes invoquen a dicho método.

Gestión de Errores

Excepciones estándares en Java

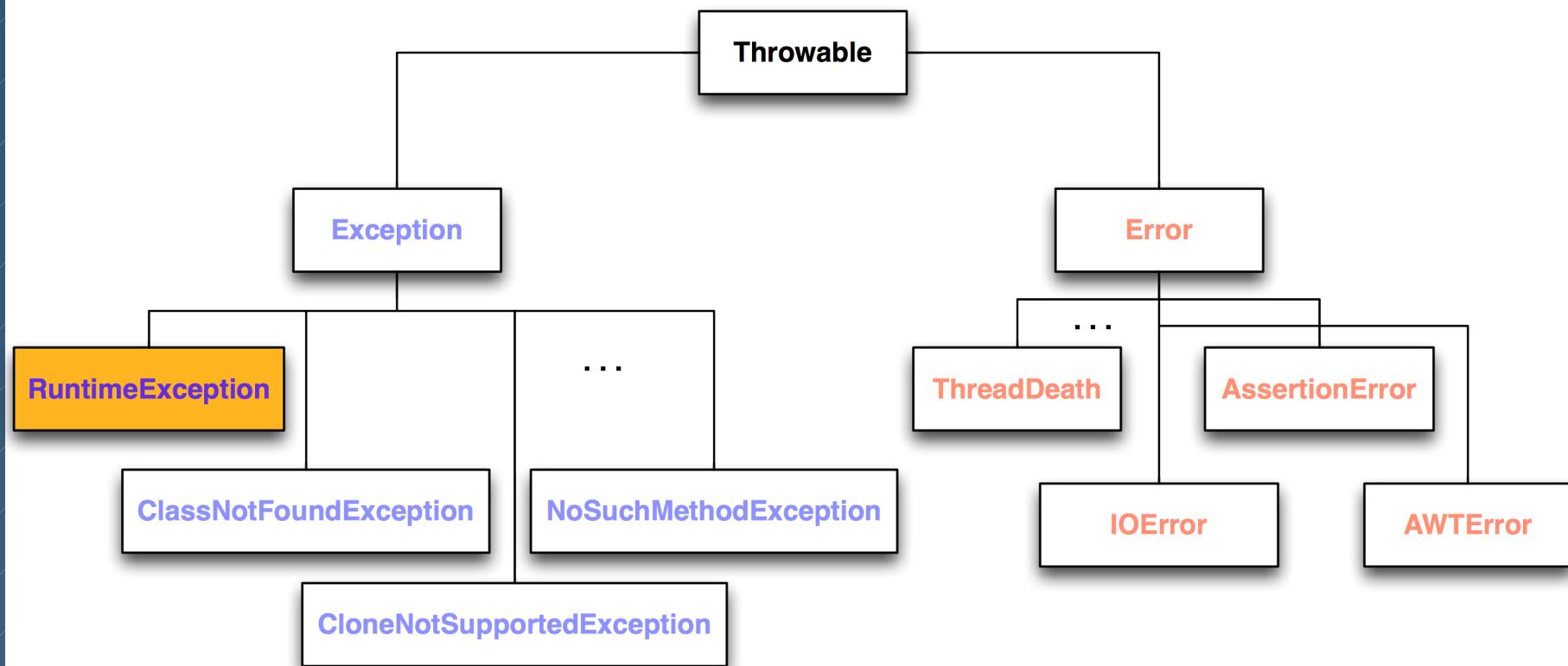
- Todas las excepciones lanzadas por componentes de la API de Java son tipos derivados de la clase **Throwable** (definida en `java.lang`), que tiene, entre otros, los siguientes métodos:

```
class Throwable {  
    ...  
    public Throwable();  
    public Throwable(String message);  
    ...  
    public String getMessage();  
    public void printStackTrace();  
    public String toString(); // nombre clase + message  
}
```

- Estos métodos están disponibles cuando creamos excepciones de usuario.

Gestión de Errores

Excepciones estándar en Java



Excepciones verificadas (checked exceptions):

Si un método lanza alguna de estas excepciones directa o indirectamente, excepto las de tipo `RuntimeException`, Java obliga a que lo declare en su especificación de excepciones, en tiempo de compilación.

Gestión de Errores

Excepciones estándar en Java

Exception:

Indican errores de ejecución del API de Java o nuestros.

Error:

Indican errores de compilación o del sistema. Normalmente no se capturan.

RuntimeException:

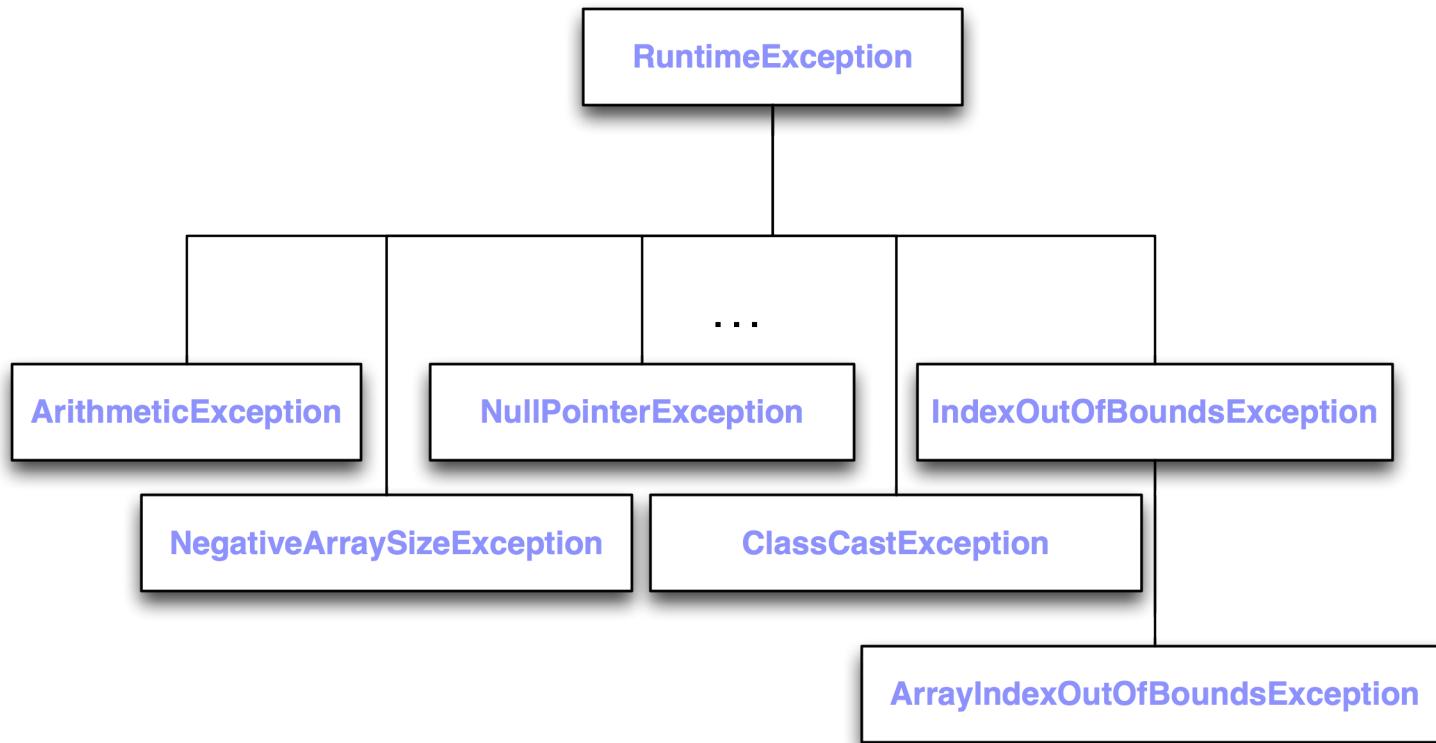
Suelen indicar errores de programación, no del usuario, por lo que tampoco se suelen capturar.

Todas las excepciones estándar disponen de al menos dos constructores: Uno por defecto y otro que acepta un String (que describe lo ocurrido).

Podemos generar (throw) cualquier tipo de excepción estándar, pero no se suelen generar las de tipo Error o RuntimeException.

Gestión de Errores

RuntimeException: Excepciones no verificadas



Excepciones no verificadas (unchecked exceptions):

Son todas aquellas que derivan (extends) de RuntimeException.

Si un método puede lanzarlas, no es obligatorio incluirlas en su especificación de excepciones.

Gestión de Errores

RuntimeException: **especificación de excepción**

- La ausencia de especificador indica que la función podría lanzar cualquier excepción no verificada:

```
public int f() {  
    if (algo_falla)  
        throw new RuntimeException("Algo ha fallado");  
}
```

- O ninguna excepción

```
public int g() {  
    return 3+2;  
}
```

Gestión de Errores

Excepciones de usuario

- Es habitual tipificar errores de una aplicación creando clases de objetos que representan diferentes circunstancias de error.
- La ventaja de hacerlo así es que
 - Podemos incluir información extra al lanzar la excepción.
 - Podemos agrupar las excepciones en jerarquías de clases.
- Creamos excepciones de usuario tomando como base la clase `Exception`:

```
class miExcepcion extends Exception {  
    private int x;  
  
    public miExcepcion(int a, String m) {  
        super(m); x=a; }  
    public String queHaPasado() { return getMessage(); }  
    public int getElCulpable() { return x; }  
}
```

Gestión de Errores. Particularidades

Orden de captura de excepciones

- **El orden de colocación de los bloques catch es importante.**

```
try {  
    if (error1) throw new miExcepcion("ERROR 1");  
    If (error2) throw new RuntimeException("ERROR 2");  
    If (error3) throw new Exception("ERROR 3");  
}  
catch (miExcepcion e1) { ... tratar error 1 ... }  
catch (RuntimeException e2) { ... tratar error 2 ... }  
catch (Exception e3) { ... tratar error 3 ... }
```

Si colocamos el 'catch (Exception e3)' el primero, todos los errores serían tratados como errores 3. El compilador de Java detecta esta situación incorrecta.

Gestión de Errores. Particularidades

Anidamiento de bloques **try**

- Los bloques try/catch se pueden anidar

```
try {
    if (error1) throw new miExcepcion("ERROR 1");
    try {
        if (error2) throw new RuntimeException("ERROR 2");
        try {
            if (error3) throw new Exception("ERROR 3");
        }
        catch (Exception e3) { ... tratar error 3 ... }
    }
    catch (RuntimeException e2) { ... tratar error 2 ... }
}
catch (miExcepcion e1) { ... tratar error 1 ... }
```

Gestión de errores. Particularidades

Excepciones en constructores

- Si se produce una excepción en un constructor (y no se captura dentro del mismo) el objeto no llega a ser construido.
- Normalmente esto sucede cuando los argumentos del constructor no permiten crear un objeto válido.

Opciones:

- Lanzar una excepción de usuario
- Lanzar `IllegalArgumentException`

```
class Construccion {  
    public Construccion(int x) throws IllegalArgumentException {  
        if (x<0)  
            throw new IllegalArgumentException("No admito negativas.");  
    }  
  
    public static void main(String args[]) {  
        try {  
            Construccion c = new Construccion(-3);  
        } catch (Exception ex) { ... 'c' no ha sido construído ... }  
    }  
}
```

Gestión de Errores

Regeneración de una excepción

- A veces es necesario tratar parte del problema que generó una excepción a un determinado nivel y regenerarla para terminar de tratarlo a un nivel superior.

- Estrategias
 1. Regenerar la misma excepción
 2. Generar una excepción distinta

Gestión de Errores

Regeneración de una excepción

- Estrategia 1: regenerar la misma excepción

```
public class Rethrowing {  
    public static void f() throws Exception {  
        throw new Exception("thrown from f()");  
    }  
    public static void g() throws Exception {  
        try {  
            f();  
        } catch(Exception e) {  
            // tratar parte del problema aquí...  
            throw e; // regenerar la excepción  
        }  
    }  
  
    public static void main(String[ ] args) {  
        try {  
            g();  
        } catch(Exception e) {  
            // terminar de tratar el problema generado en f()  
        }  
    }  
}
```

Gestión de Errores

Regeneración de una excepción

- Estrategia 2: generar una excepción distinta

```
class OneException extends Exception {}  
class TwoException extends Exception {}  
  
public class RethrowNew {  
    public static void f() throws OneException {  
        throw new OneException();  
    }  
    public static void g() throws TwoException {  
        try {  
            f();  
        } catch(OneException e) {  
            // tratar parte del problema aquí...  
            throw new TwoException(); // generar nueva excepción  
        }  
    }  
  
    public static void main(String[] args) {  
        try {  
            g();  
        } catch(TwoException e) {  
            // terminar de tratar el problema generado en f()  
        }  
    }  
}
```

Gestión de Errores

Excepciones: Resumen

- **Ventajas:**
 - Separar el manejo de errores del código normal
 - Agrupar los tipos de errores y la diferenciación entre ellos
 - Obliga al código cliente a tratar (o ignorar) expresamente las condiciones de error.
- **Inconvenientes**
 - Sobrecarga del sistema para gestionar los flujos de control de excepciones.

Gestión de errores

A tener en cuenta:

- Si se produce una excepción
 - ¿Se limpiará todo apropiadamente?
 - ¿Debemos tratarla en el lugar donde se produce?
 - ¿Mejor pasarla al siguiente nivel (al llamador)?
 - ¿Tratarla parcialmente y relanzarla (la misma u otra excepción diferente) al siguiente nivel?
 -
- Lo habitual es delegar en el llamador el tratamiento de la excepción:
 - “No captures una excepción si no sabes qué hacer con ella”

Gestión de Errores

Ejemplo

- EJERCICIO
 - **Definid una excepción de usuario llamada** ExcepcionDividirPorCero que sea lanzada por el siguiente método al intentar dividir por cero:

```
class Division {  
    static float div(float x, float y)  
    { return x/y; }  
}
```

- Escribe un programa que invoque a div() y trate correctamente la excepción.

Gestión de Errores

Ejemplo

- **SOLUCIÓN:**

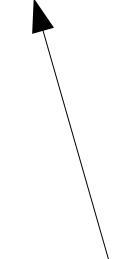
```
class ExcepcionDividirPorCero extends Exception {  
    public ExcepcionDividirPorCero(String msg)  
    { super(msg); }  
}
```

Gestión de Errores

Ejemplo

```
class Divisora {  
  
    public static void main(String args[])  
    {  
        double dividendo, divisor, resultado;  
        dividendo = 4.0;  
        divisor = 0.0;  
        try {  
            resultado = Division.div(dividendo, divisor);  
            System.out.println(dividendo+"/"+divisor+"="+resultado);  
        }  
        catch (ExcepcionDividirPorCero exce)  
        {  
            System.err.println(exce.getMessage());  
            exce.printStackTrace();  
        }  
    }  
}
```

¿Sería correcto
colocar esta instrucción
detrás del
bloque catch?



Gestión de Errores

Excepciones: Ejercicio propuesto

■ **Ejercicio**

- Define una clase PilaEnteros que gestione una pila de enteros. La pila se crea con una capacidad máxima. Queremos que la pila controle su desbordamiento al intentar apilar más elementos de los que admite, mediante una excepción de usuario ExcepcionDesbordamiento. Define la clase PilaEnteros y sus métodos Pila() y apilar().
- Implementa un programa principal de prueba que intente apilar más elementos de los permitidos y capture la excepción en cuanto se produzca (dejando de intentar apilar el resto de elementos).

Gestión de errores

Bibliografía

- Bruce Eckel. ***Piensa en Java 4^a edición***
 - Cap. 12
- Bruce Eckel. ***Thinking in Java, 3rd. edition***
 - Cap. 9
- C. Cachero et al. ***Introducción a la programación orientada a objetos***
 - Cap. 5 (ejemplos en C++)

U.D. 5

HERENCIA

Cristina Cachero, Pedro J. Ponce de León

versión 20141019



Tema 3. HERENCIA

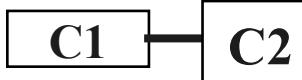
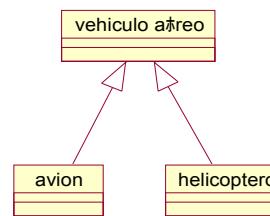
Objetivos



- Entender el mecanismo de abstracción de la herencia.
- Distinguir entre los diferentes tipos de herencia
- Saber implementar jerarquías de herencia en Java
- Saber discernir entre jerarquías de herencia seguras (bien definidas) e inseguras.
- Reutilización de código: Ser capaz de decidir cuándo usar herencia y cuándo optar por composición.

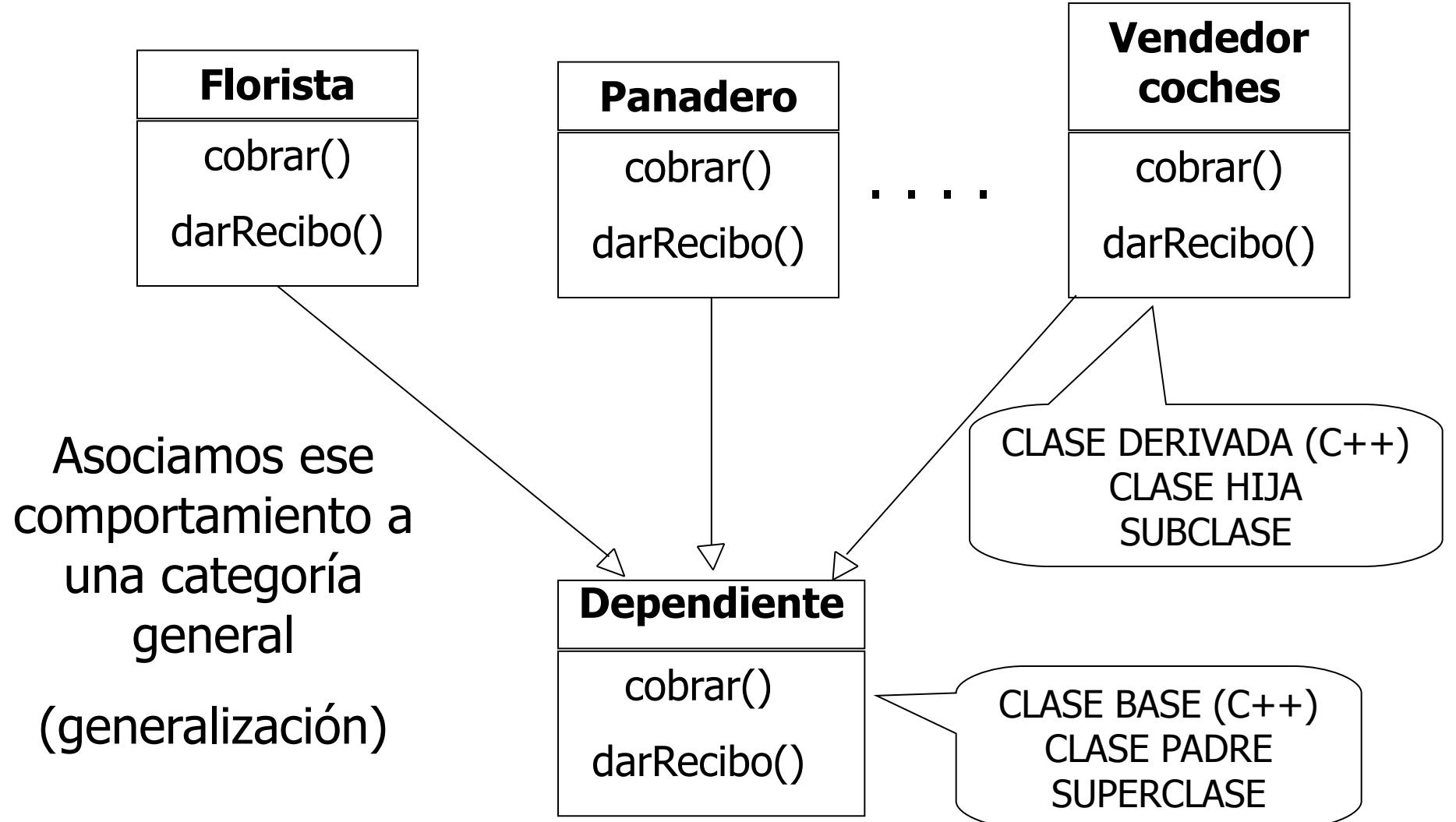
Herencia

Del tema anterior...

	Persistente	No persist.
Entre objetos	<ul style="list-style-type: none">▪ Asociación▪ Todo-Parte<ul style="list-style-type: none">▪ Agregación▪ Composición   	<ul style="list-style-type: none">▪ Uso (depend) 
Entre clases	<ul style="list-style-type: none">▪ Generalización 	

HERENCIA

Motivación



Clasificación y generalización



- La mente humana clasifica los conceptos de acuerdo a dos dimensiones:
 - Pertenencia (TIENE-UN) -> *Relaciones todo-parte*
 - Variedad (ES-UN) -> *Herencia*
- La herencia consigue **clasificar** los conceptos (abstracciones) por variedad, siguiendo el modo de razonar humano.
 - Este modo de razonar humano se denomina **GENERALIZACIÓN**, y da lugar a jerarquías de generalización/especialización.
 - La implementación de estas jerarquías en un lenguaje de programación da lugar a jerarquías de herencia.

Herencia como implementación de la Generalización



- La generalización es una relación semántica entre clases, que determina que la subclase debe incluir todas las propiedades de la superclase.
- Disminuye el número de relaciones (asociaciones y agregaciones) del modelo
- Aumenta la comprensibilidad, expresividad y abstracción de los sistemas modelados.
- Todo esto a costa de un mayor número de clases

- La herencia es el mecanismo de implementación mediante el cual elementos más específicos incorporan la estructura y comportamiento de elementos más generales (Rumbaugh 99)
- Gracias a la herencia es posible **especializar** o **extender** la funcionalidad de una clase, derivando de ella nuevas clases.
- La herencia es siempre **transitiva**: una clase puede heredar características de superclases que se encuentran muchos niveles más arriba en la jerarquía de herencia.
 - Ejemplo: si la clase *Perro* es una subclase de la clase *Mamífero*, y la clase *Mamífero* es una subclase de la clase *Animal*, entonces el *Perro* heredará atributos tanto de *Mamífero* como de *Animal*.

HERENCIA

Test “ES-UN”



- La clase A se debe relacionar mediante herencia con la clase B si **“A ES-UN B”**. Si la frase suena bien, entonces la situación de herencia es la más probable para ese caso
 - Un pájaro es un animal
 - Un gato es un mamífero
 - Un pastel de manzana es un pastel
 - Una matriz de enteros es un matriz
 - Un coche es un vehículo

- Sin embargo, si la frase suena rara por una razón u otra, es muy probable que la relación de herencia no sea lo más adecuado. Veamos unos ejemplos:
 - Un pájaro es un mamífero
 - Un pastel de manzana es una manzana
 - Una matriz de enteros es un entero
 - Un motor es un vehículo

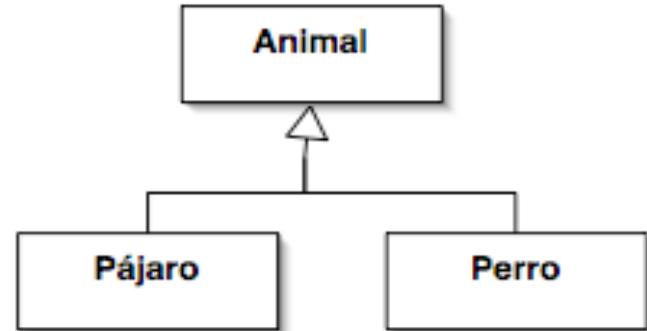
- *La herencia como reutilización de código:* Una clase derivada puede heredar comportamiento de una clase base, por tanto, el código no necesita volver a ser escrito para la derivada.
 - **Herencia de implementación**
- *La herencia como reutilización de conceptos:* Una clase derivada **sobscribe** el comportamiento definido por la clase base. Aunque no se comparte ese código entre ambas clases, ambas comparten el prototipo del método (comparten el concepto).
 - **Herencia de interfaz**

- Simple/Múltiple
- De implementación/de interfaz

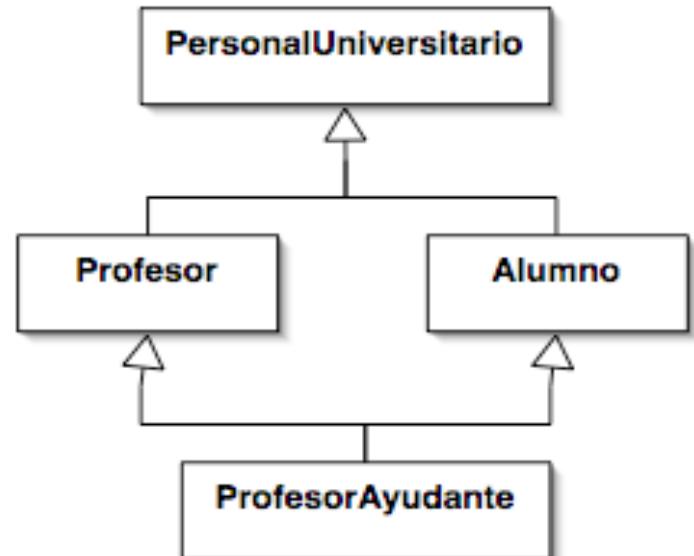
Tipos de Herencia

- Simple/Múltiple

- **Simple:** única clase base



- **Múltiple:** Más de una clase base



- De implementación/de interfaz
 - **De implementación:** La implementación de los métodos es heredada. Puede sobreescibirse en las clases derivadas.
 - **De interfaz:** Sólo se hereda la interfaz, no hay implementación a nivel de clase base (*interfaces* en Java, *clases abstractas* en C++)

- Atributos de la generalización

- **Solapada/Disjunta**

- Determina si un objeto puede ser *a la vez* instancia de dos o más subclases de ese nivel de herencia.
 - Java/C++ no soporta la herencia solapada (tipado fuerte)

- **Completa/Incompleta**

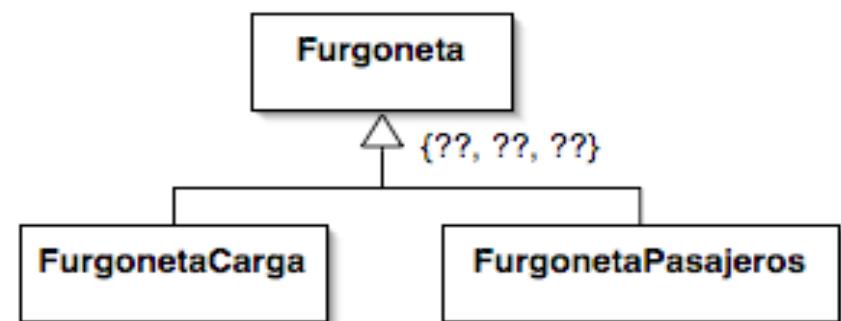
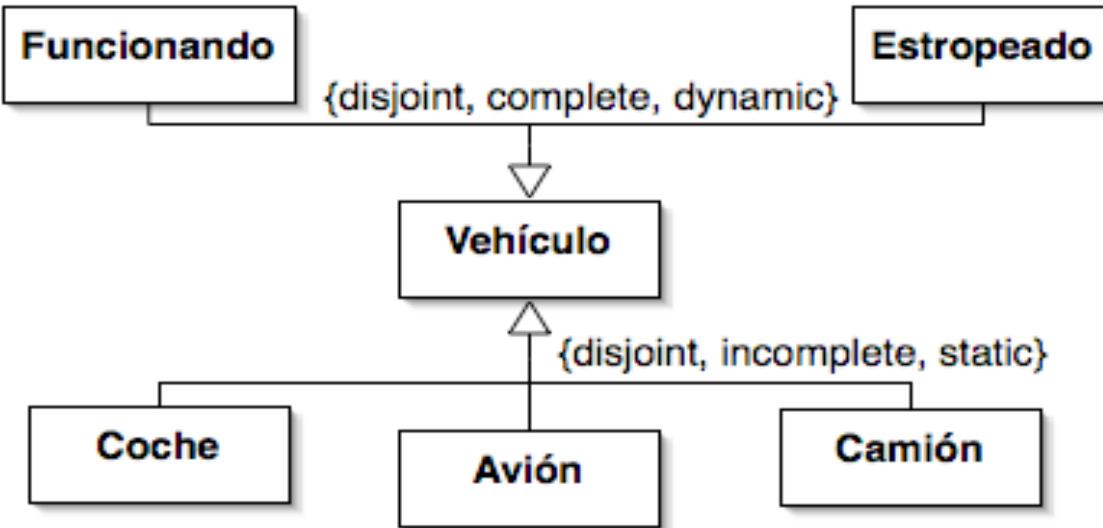
- Determina si todas las instancias de la clase padre son *a la vez* instancias de alguna de las clases hijas (completa) o, por el contrario, hay objetos de la clase padre que no pertenecen a ninguna subcategoría de las reflejadas por las clases hijas (incompleta).

- **Estática/Dinámica**

- Determina si un determinado objeto *puede pasar de ser instancia de una clase hija a otra* dentro de un mismo nivel de la jerarquía de herencia.
 - Java/C++ no soporta la herencia dinámica (tipado fuerte)

Herencia

Caracterización: ejemplos

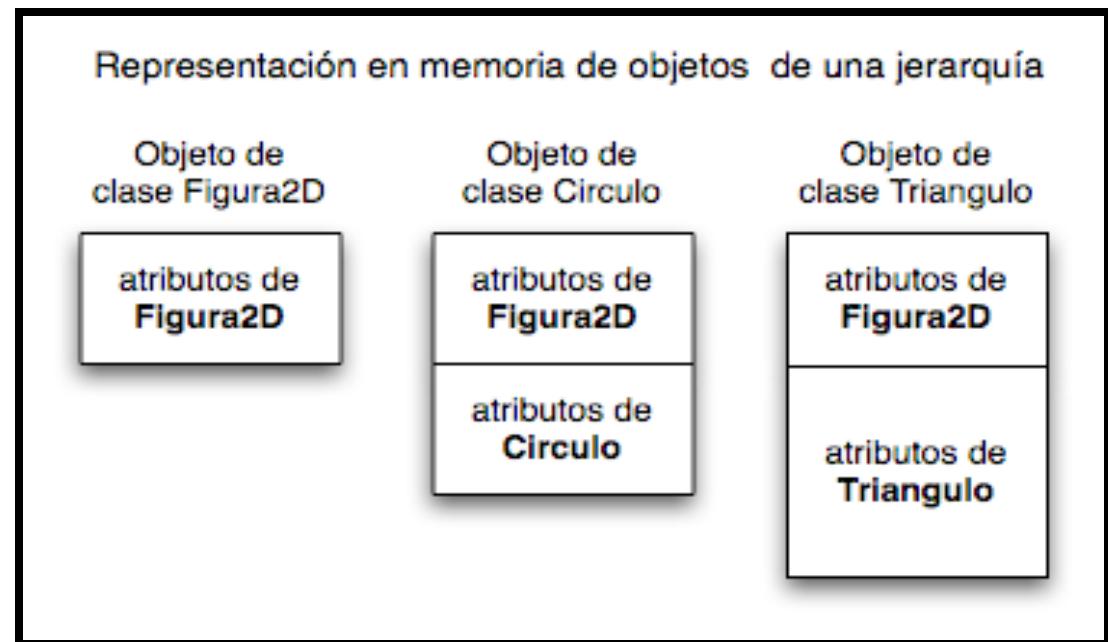
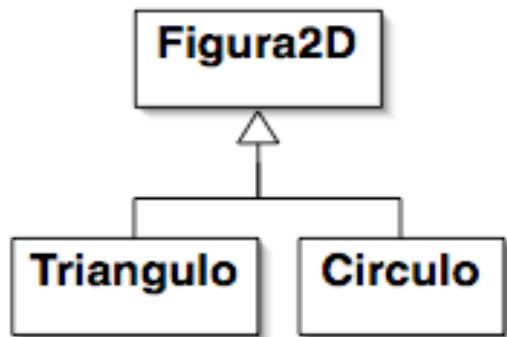


HERENCIA DE IMPLEMENTACIÓN

Herencia Simple

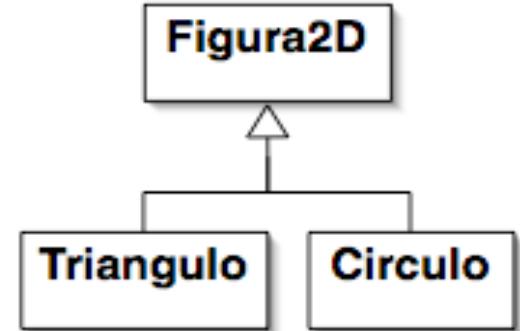
Herencia Simple

- Mediante la herencia, las propiedades definidas en una clase base son heredadas por la clase derivada.
- La clase derivada puede añadir propiedades específicas (atributos, métodos o roles)



Herencia Simple

```
class Figura2D {  
    public void setColor(Color c) {...}  
    public Color getColor() {...}  
    private Color colorRelleno;  
    ... }  
  
class Circulo extends Figura2D {  
...  
    public void vaciar() {  
        colorRelleno=Color.NINGUNO;  
        // ;ERROR! colorRelleno es privado  
        setColor(Color.NINGUNO); // OK  
    }  
}
```



La parte privada de una clase base no es directamente accesible desde la clase derivada.

```
// código cliente  
Circulo c = new Circulo();  
c.setColor(AZUL);  
c.getColor();  
c.vaciarCirculo();
```

Herencia simple

Visibilidad atributos/métodos



■ Ámbito de visibilidad **protected** (UML: '#')

- C++: Los atributos/métodos *protected* son directamente accesibles desde la propia clase y sus clases derivadas. Tienen visibilidad privada para el resto de ámbitos.
- Java: directamente accesibles desde la propia clase, sus derivadas y las clases que estén en el mismo paquete. Visibilidad privada para el resto de ámbitos.

```
class Figura2D {  
    protected Color colorRelleno;  
    ...  
}  
  
class Circulo extends Figura2D {  
    public void vaciarCirculo() {  
        colorRelleno=NINGUNO; //OK, protected  
    }  
    ...  
}
```

```
//código cliente en otro paquete  
Circulo c = new Circulo();  
c.colorRelleno=NINGUNO;  
// ¡ERROR! colorRelleno  
// es privado aquí
```

Herencia simple

Visibilidad atributos/métodos



- Ámbito de visibilidad **de paquete** (Java)

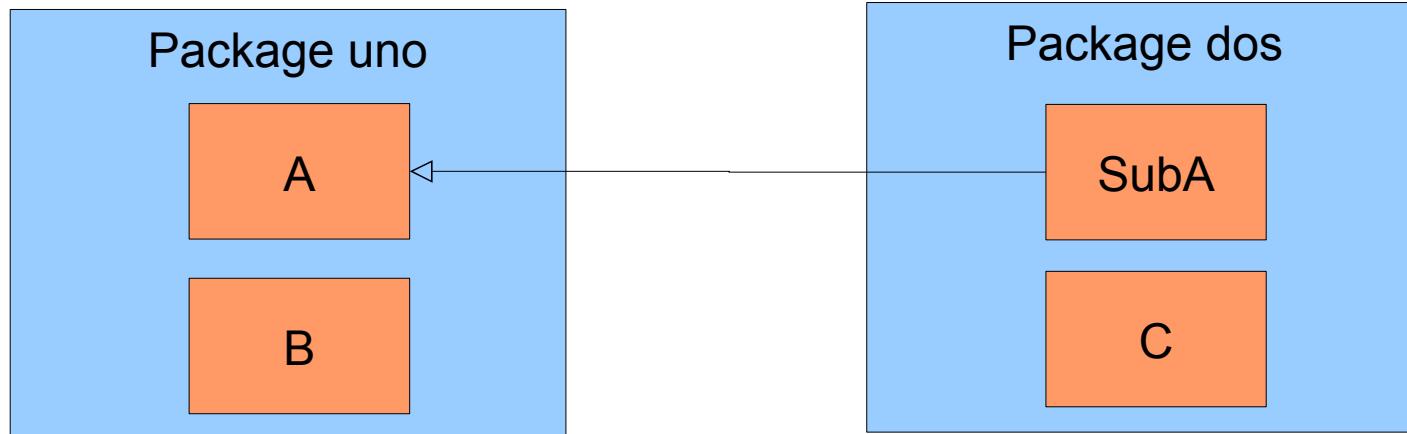
- Java: directamente accesibles desde la propia clase y las clases que estén en el mismo paquete. Visibilidad privada para el resto de ámbitos ¡(incluso clases derivadas en otros paquetes!)

```
package prog3;  
  
class Figura2D {  
    // visibilidad de paquete por defecto  
    Color colorRelleno;  
  
    ...  
}
```

```
package prog3;  
  
class Cliente { // en el mismo paquete  
    public void probarFigura(Figura2D f) {  
        f.colorRelleno=NINGUNO; //OK, protected  
    }  
  
    ...  
}
```

Herencia simple

Visibilidad atributos/métodos en Java



	A	B	SubA	C
public	ok	ok	ok	ok
protected	ok	ok	ok	-
package	ok	ok	-	-
private	ok	-	-	-

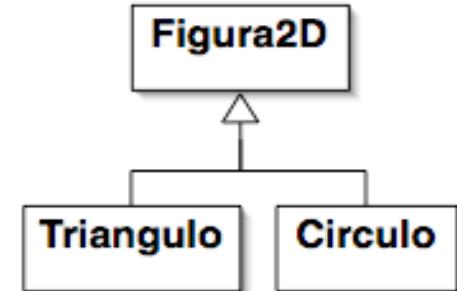
Tipos de Herencia Simple

- Herencia Pública
 - Se hereda interfaz e implementación

```
// JAVA sólo soporta herencia pública
class Circulo extends Figura2D
{
    ...
}
```

```
// C++
class Circulo : public Figura2D
{
    ...
};
```

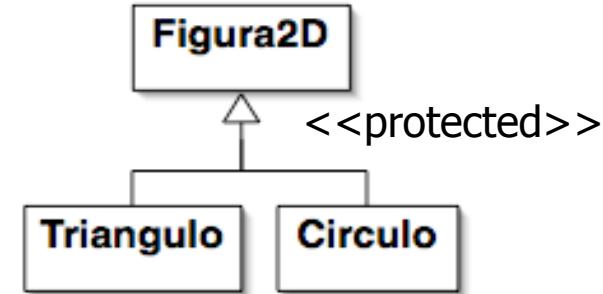
<<public>>



Tipos de Herencia Simple

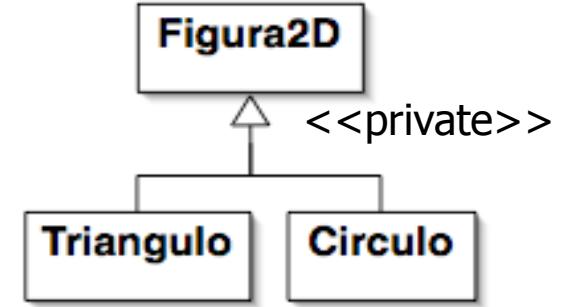
- Herencia Protegida (C++)

```
class Circulo : protected Figura2D {  
    ...  
};
```



- Herencia Privada (C++, por defecto)

```
class Circulo : private Figura2D {  
    ...  
};
```



Estos tipos de herencia permiten heredar sólo la implementación. La interfaz de la clase base queda innaccesible desde objetos de clase derivada.

Tipos de Herencia Simple

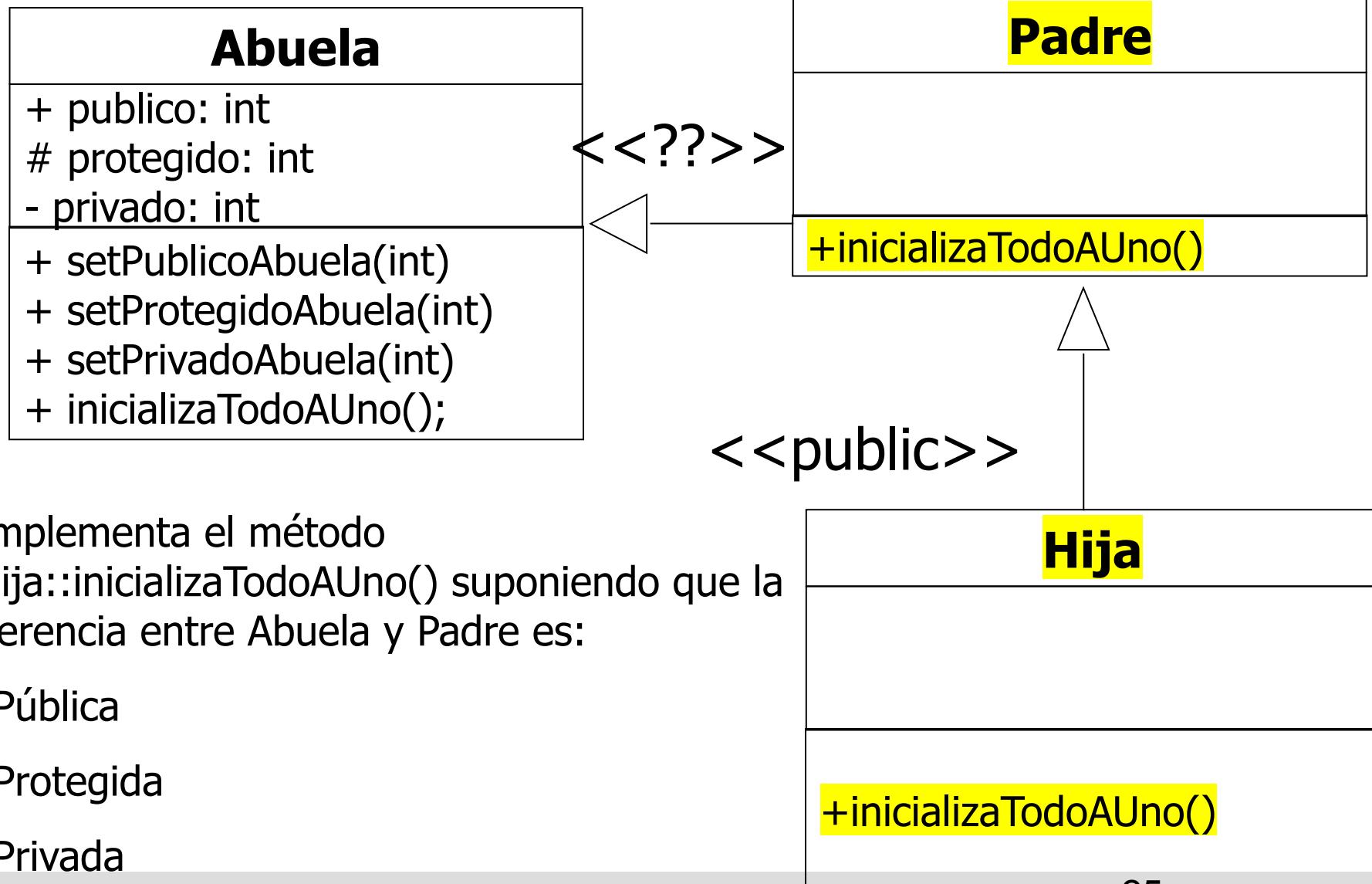


Visibilidad en clase base \ Ámbito Herencia	CD (*) H. Pública	CD H. Protegida	CD H. privada
Private	No direct. accesible	No direct. accesible	No direct. accesible
Protected	Protected	Protected	Private
Public	Public	Protected	Private

(*) CD: Clase derivada

Tipos Herencia Simple

Ejercicio



- En la clase derivada se puede:
 - Añadir nuevos métodos/atributos
 - Modificar los métodos heredados de la clase base
- **REFINAMIENTO:** se añade comportamiento nuevo antes y/o después del comportamiento heredado. (se puede simular en C++, Java)
 - *C++, Java:* Constructores y destructores se refinan
- **REEMPLAZO:** el método heredado se redefine completamente, de forma que sustituye al original de la clase base.

- Ejemplo de **reemplazo** en Java

```
class A {  
    public void doIt() {  
        System.out.println("HECHO en A");  
    }  
}  
  
class B extends A {  
    public void doIt() {  
        System.out.println("HECHO en B");  
    }  
}
```

■ Ejemplo de **refinamiento** en Java

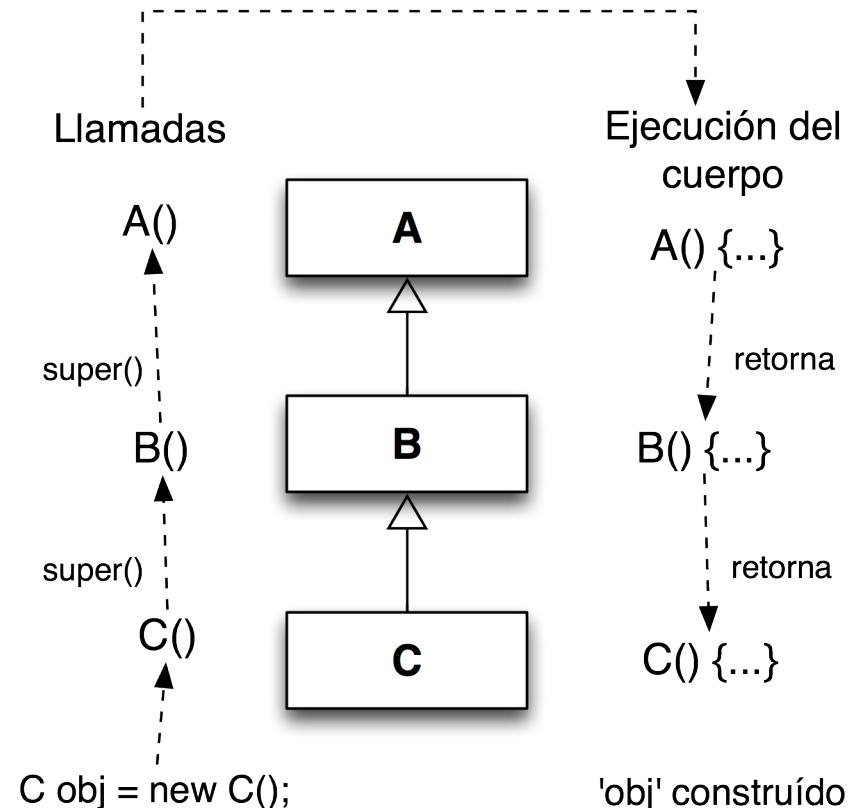
```
class A {  
    public void doIt() { System.out.println("HECHO en A."); }  
    public void doItAgain() {  
        System.out.println("HECHO otra vez en A.");  
    }  
}  
class B extends A {  
    public void doIt() {  
        System.out.println("HECHO en B.");  
        super.doIt(); // impl. base tras impl. derivada  
    }  
    public void doItAgain() {  
        super.doItAgain(); // impl. base antes de impl. derivada  
        System.out.println("HECHO otra vez en B.");  
    }  
}
```

this : referencia a objeto actual usando implementación de la clase actual.
super : referencia a objeto actual usando implementación de la clase base.

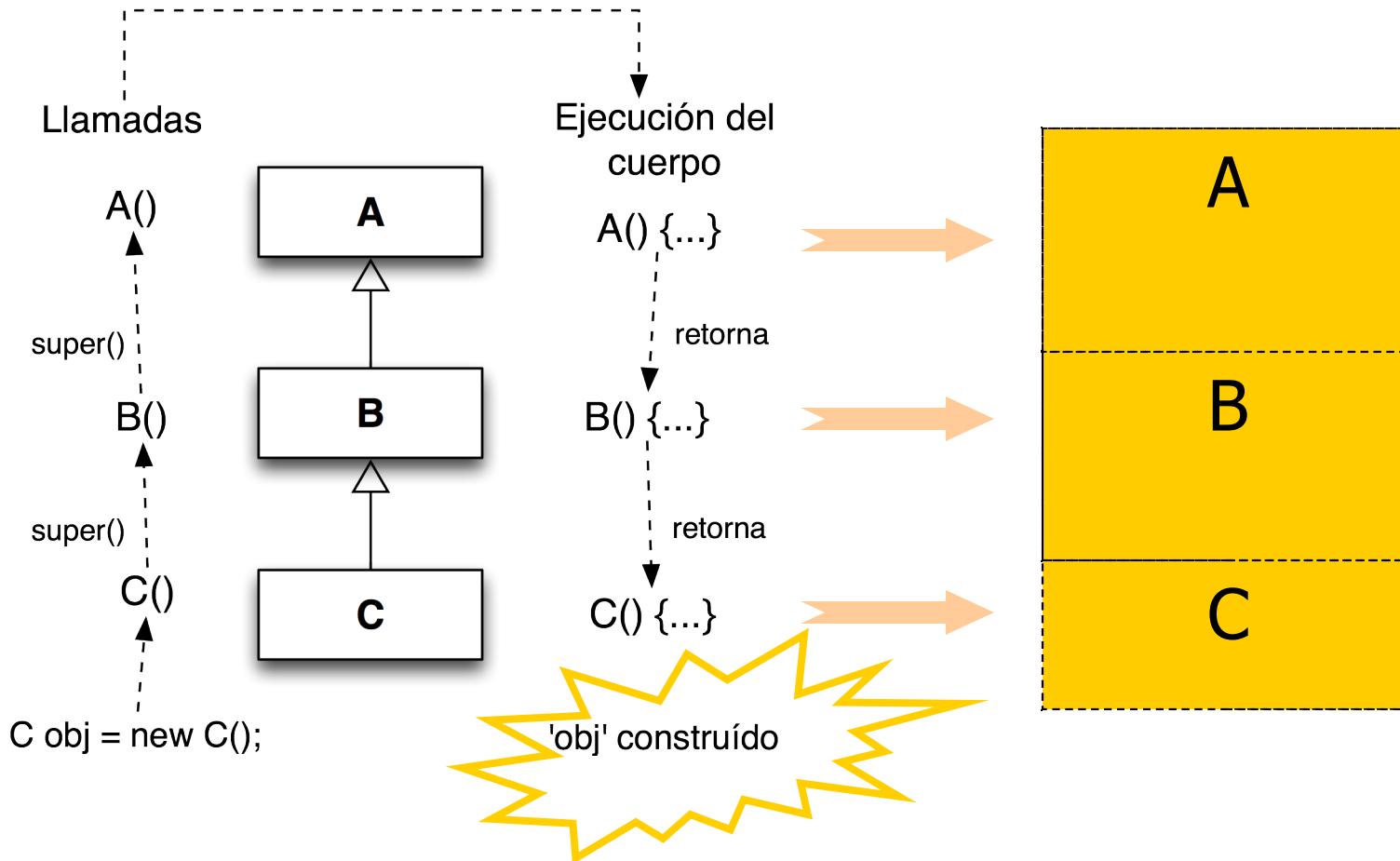
El constructor en herencia simple

- Los constructores no se heredan

- Siempre son definidos para las clases derivadas
- Creación de un objeto de clase derivada: Se invoca a todos los constructores de la jerarquía
- Orden de ejecución de constructores: Primero se ejecuta el constructor de la clase base y luego el de la derivada.



El constructor en herencia simple



El constructor en herencia simple



- Esto implica que la clase derivada aplica una política de **refinamiento**: añadir comportamiento al constructor de la clase base.
- Ejecución implícita del constructor por defecto de clase base al invocar a un constructor de clase derivada.
- Ejecución explícita de cualquier otro tipo de constructor en la zona de inicialización (refinamiento explícito). En particular, el constructor de copia.

(CONSEJO: Inicialización de atributos de la clase base: en la clase base, no en la derivada)

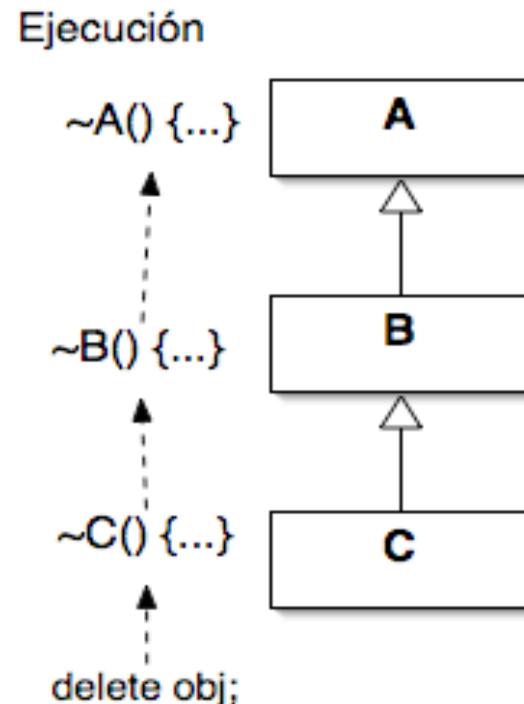
El constructor en herencia simple

■ Ejemplo

```
class Figura2D {  
    private Color colorRelleno;  
    public Figura2D() { colorRelleno= Color.NINGUNO; }  
    public Figura2D(Color c) { colorRelleno=c; }  
    public Figura2D(Figura2D f) { colorRelleno=f.colorRelleno; }  
    ...}  
  
class Circulo extends Figura2D {  
    private double radio;  
    public Circulo() { radio=1.0; } //llamada implícita a Figura2D()  
    public Circulo() { super(); radio=1.0; } //llamada explícita  
    public Circulo(Color col, double r) { super(col); radio=r; }  
    public Circulo(Circulo cir) { super(cir); radio=cir.radio; }  
    ...}
```

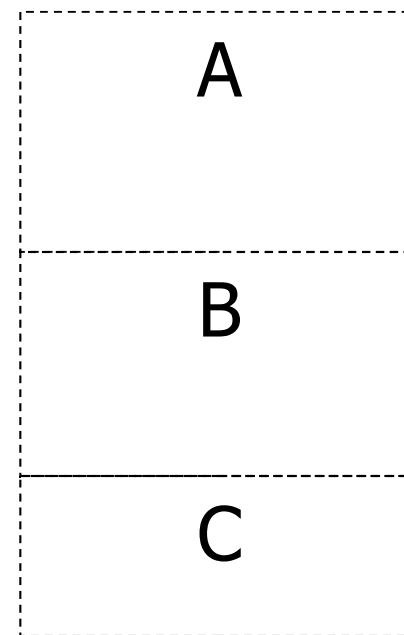
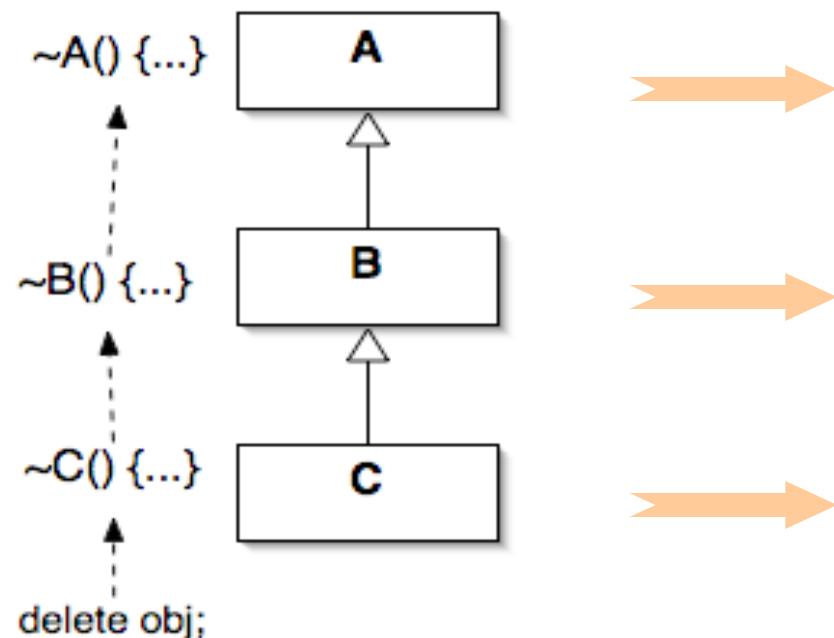
El destructor en herencia simple (C++)

- C++: el destructor no se hereda.
 - Siempre es definido para la clase derivada
 - Destrucción de un objeto de clase derivada: se invoca a todos los destructores de la jerarquía
 - Primero se ejecuta destructor de la clase derivada y luego el de la clase base.
 - Llamada implícita a los destructores de la clase base.



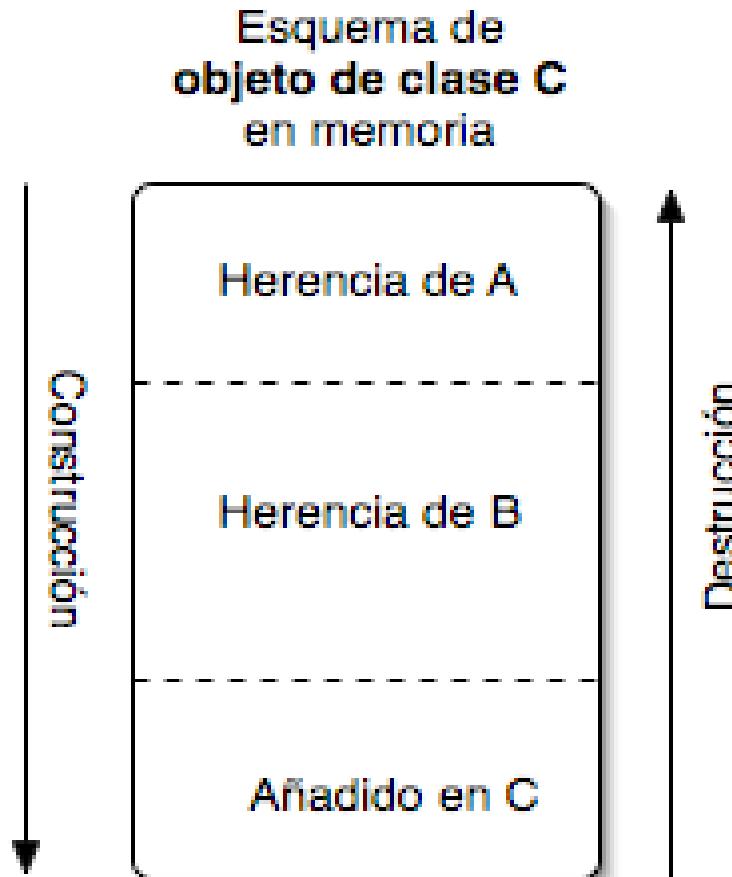
El destructor en herencia simple (C++)

Ejecución



Orden de construcción/destrucción en C++

- Los objetos se destruyen en orden inverso al de construcción.



Orden de construcción/destrucción en Java



- El orden de construcción es el mismo que en C++: de la clase base a la derivada.
- El orden de destrucción es responsabilidad del programador. Se debe implementar si existen recursos (distintos de la memoria reservada para objetos) que liberar.
 - Dos estrategias:
 - Usar métodos **finalize()**
 - Desventaja: No sabemos cuándo se ejecutarán.
 - Crear métodos propios para garantizar la correcta liberación de recursos (aparte de la memoria)
 - Desventaja: el código cliente debe invocar explícitamente dichos métodos.

Orden de construcción/destrucción en Java



Destrucción/limpieza usando métodos finalize()

```
class Animal  {
    Animal() {
        System.out.println("Animal()");
    }
    protected void finalize() throws Throwable{
        System.out.println("Animal finalize");
    }
}

class Amphibian extends Animal {
    Amphibian() {
        System.out.println("Amphibian()");
    }
    protected void finalize() throws Throwable{
        System.out.println("Amphibian finalize");
        try {
            super.finalize();
        } catch(Throwable t) {}
    }
}
```

```
public class Frog extends Amphibian {
    Frog() {
        System.out.println("Frog()");
    }
    protected void finalize() throws Throwable {
        System.out.println("Frog finalize");
        try {
            super.finalize();
        } catch(Throwable t) {}
    }
    public static void main(String[] args) {
        new Frog(); // Instantly becomes garbage
        System.out.println("bye!");
        // Must do this to guarantee that all
        // finalizers will be called:
        System.runFinalizersOnExit(true);
    }
} ///:~
```

(tomado de 'Piensa en Java, 4^a ed.', Bruce Eckl)

Orden de construcción/destrucción en Java



Destrucción/limpieza usando métodos propios

```
class Shape {  
    Shape(int i) { print("Shape ctor"); }  
    void dispose() { print("Shape dispose"); }  
}  
  
class Circle extends Shape {  
    Circle(int i) {  
        super(i);  
        print("Drawing Circle");  
    }  
    void dispose() {  
        print("Erasing Circle");  
        super.dispose();  
    }  
}  
  
class Triangle extends Shape {  
    Triangle(int i) {  
        super(i);  
        print("Drawing Triangle");  
    }  
    void dispose() {  
        print("Erasing Triangle");  
        super.dispose();  
    }  
}
```

```
public class CADSystem extends Shape {  
    private Circle c;  
    private Triangle t;  
  
    public CADSystem(int i) {  
        super(i + 1);  
        c = new Circle(1);  
        t = new Triangle(1);  
        print("Combined constructor");  
    }  
    public void dispose() {  
        print("CADSystem.dispose()");  
        // The order of cleanup is the reverse  
        // of the order of initialization:  
        t.dispose();  
        c.dispose();  
        super.dispose();  
    }  
    public static void main(String[] args) {  
        CADSystem x = new CADSystem(47);  
        try {  
            // Code and exception handling...  
        } finally {  
            x.dispose();  
        }  
    }  
}
```

Ejemplo Clase Base



Cuenta

```
# titular: string  
# saldo: double  
# interes: double  
# numCuentas: int
```

```
+ Cuenta()  
+ Cuenta(Cuenta)  
+ getTitular() : string  
+ getSaldo() : double  
+ getInteres() : double  
+ setSaldo(double) : void  
+ setInteres(double) : void  
+ abonarInteresMensual() : void  
+ toString() : String
```

Herencia Simple (base): TCuenta



```
class Cuenta{  
    public Cuenta(String t, double s, double i)  
    { titular=t; saldo=s; interes=i; numCuentas++; }  
    ...  
    protected string titular;  
    protected double saldo;  
    protected double interes;  
    protected static int numCuentas;  
//...}
```

TCuenta (II)

```
// ... (cont.)
```

```
public Cuenta(Cuenta tc)
{ titular=tc.titular; saldo=tc.saldo;
interes=tc.interes; numCuentas++; }

protected void finalize() throws Throwable
{ numCuentas--; }
```

Herencia Simple (base): TCuenta (III)

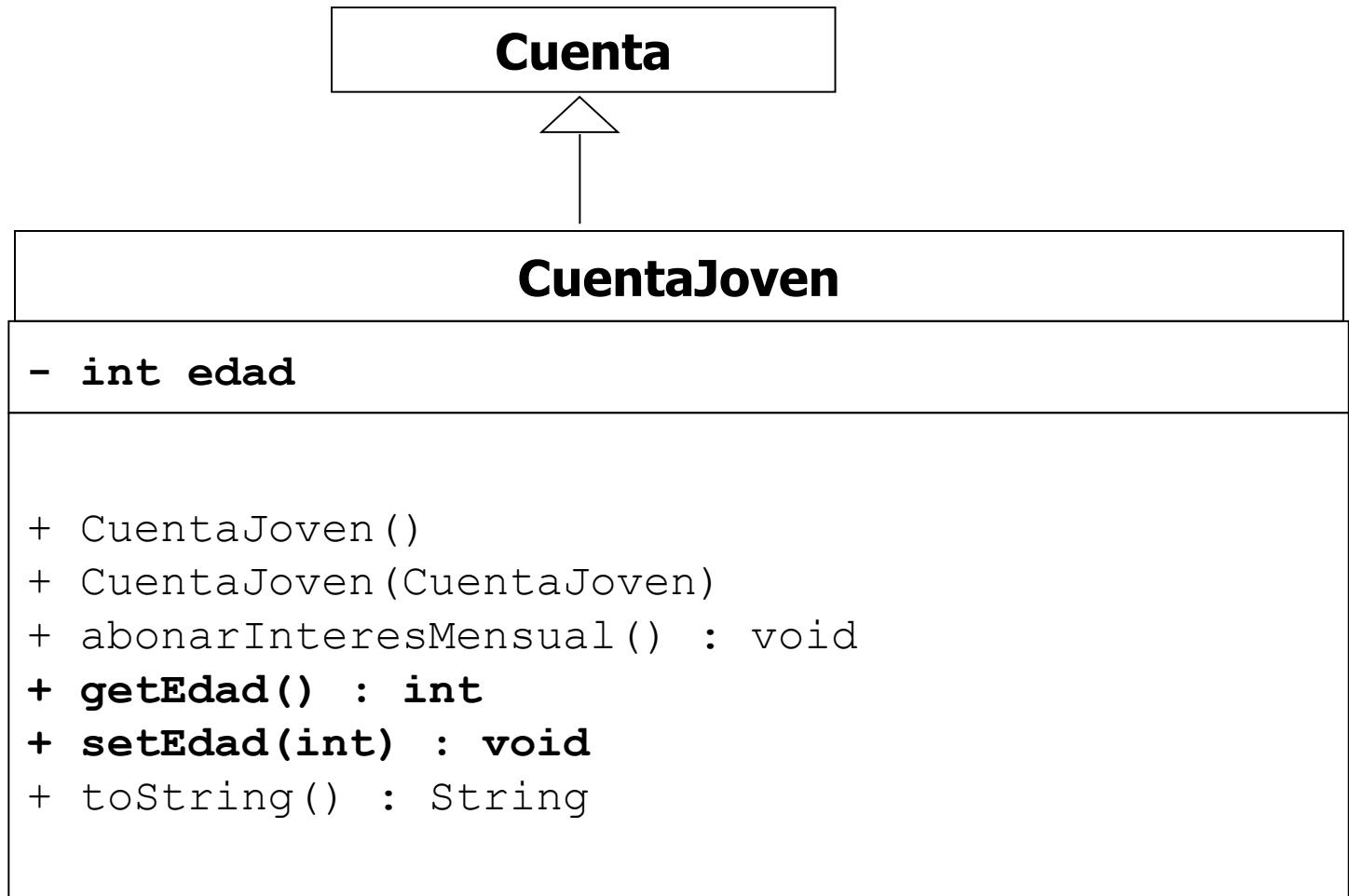


... (cont.)

```
void abonarInteresMensual()
{ setSaldo(getSaldo() * (1+getInteres()/100/12)) ; }
```

```
String toString ()
{
    return "NumCuentas=" + Cuenta.numCuentas + "\n"
        + "Titular=" + unaCuenta.titular + "\n"
        + "Saldo=" + unaCuenta.saldo + "\n"
        + "Interes=" + unaCuenta.interes + "\n";
}
```

Ejemplo clase derivada



(Los métodos cuya implementación se hereda de la clase base no se especifican en UML)

Herencia Simple (derivada): CuentaJoven (I)



```
class CuentaJoven extends Cuenta {  
    private int edad;  
  
    public CuentaJoven(String unNombre, int unaEdad,  
        double unSaldo, double unInteres)  
    {  
        super(unNombre,unSaldo,unInteres);  
        edad=unaEdad;  
    }  
  
    public CuentaJoven(CuentaJoven& tcj)  
    // llamada explícita a constructor de copia de Cuenta.  
    {  
        super(tcj);  
        edad=tcj.edad);  
    }  
  
    ...
```

¿Hay que incrementar numCuentas?

Refinamiento

Herencia Simple (derivada): TCuentaJoven (II)



```
...  
void abonarInteresMensual() {  
    //no interés si el saldo es inferior al límite  
    if (getSaldo()>=10000)  
        setSaldo(getSaldo() * (1+getInteres()/12/100));  
}  
  
int getEdad() {return edad;}  
void setEdad(int unaEdad) {edad=unaEdad;}  
  
void toString() {  
    String s = super.toString();  
    s = s + "Edad:" +edad;  
}  
}//fin clase CuentaJoven
```

Reemplazo

Métodos añadidos

Método Refinado

Herencia Simple Upcasting



Convertir un objeto de tipo derivado a tipo base

```
CuentaJoven tcj = new CuentaJoven();
```

```
Cuenta c;
```

```
c = (Cuenta)tcj; // explícito
```

```
c = tcj; // implícito
```

```
tcj.setEdad(18); //OK
```

```
c.setEdad(18); // ¡ERROR!
```

Un objeto de clase derivada al que se accede a través de una referencia a clase base sólo se puede manipular usando la interfaz de la clase base.

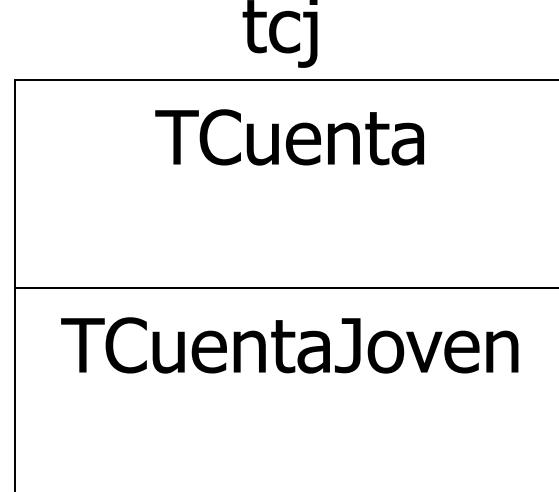
Herencia Simple (derivada): Upcasting



Cuando se convierten objetos en C++,
se hace **object slicing**

```
CuentaJoven tcj;
```

```
(TCuenta)tcj
```

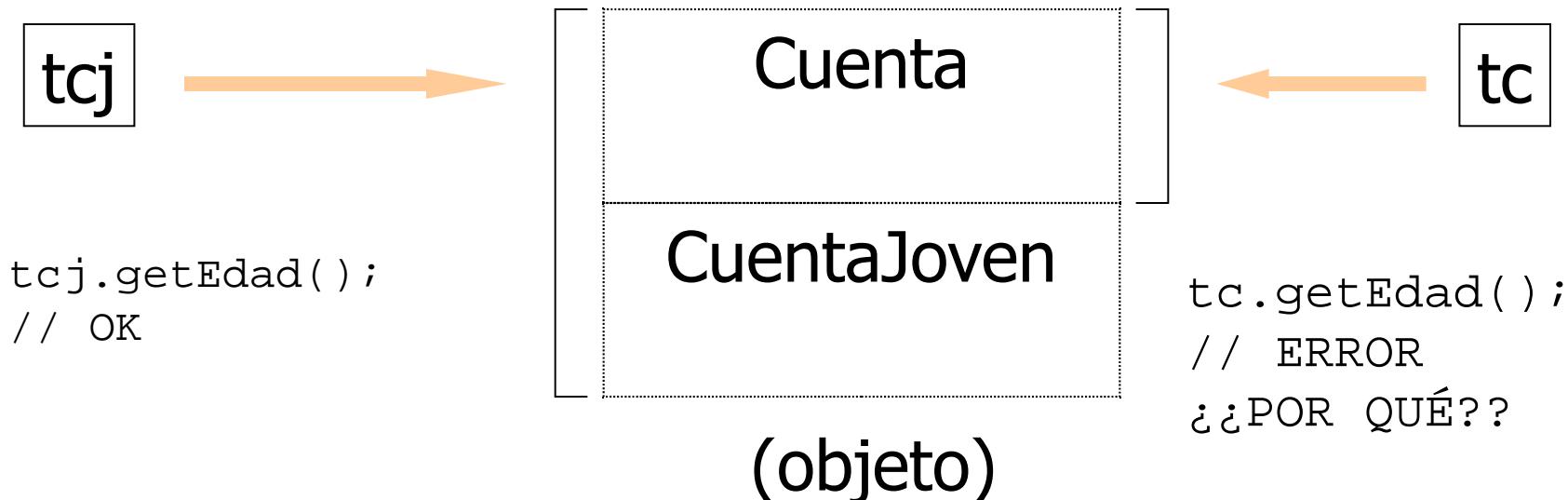


Herencia Simple (derivada): Upcasting



Con referencias (en Java o C++),
NO hay **object slicing**

```
CuentaJoven tcj = new CuentaJoven( );  
Cuenta tc = tcj; // upcasting
```



Particularidades Herencia



- En las jerarquías de herencia hay un refinamiento implícito de:
 - Constructor por defecto
- Los constructores sobrecargados se refinan explícitamente.
- Las propiedades de clase definidas en la clase base también son compartidas (heredadas) por las clases derivadas.

HERENCIA DE IMPLEMENTACIÓN

Herencia Múltiple

- Se da cuando existe más de una clase base.
- **C++** soporta la herencia múltiple de implementación.
 - Se heredan tanto las interfaces como las implementaciones de las clases base.
- **Java** sólo soporta la herencia múltiple de interfaz.
 - Sólo se hereda la interfaz de las clases base y no la implementación.

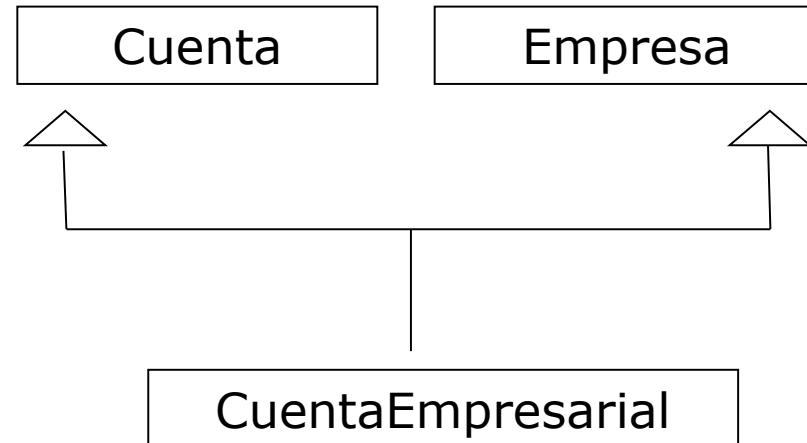
En esta sección tratamos la herencia múltiple de implementación.

Herencia múltiple de implementación



(Ejemplos en C++)

```
class Empresa {  
protected:  
    string nomEmpresa;  
  
public:  
    Empresa(string unaEmpresa)  
    { nomEmpresa=unaEmpresa; }  
    void setNombre(string nuevo)  
    { nomEmpresa = nuevo; }  
};
```



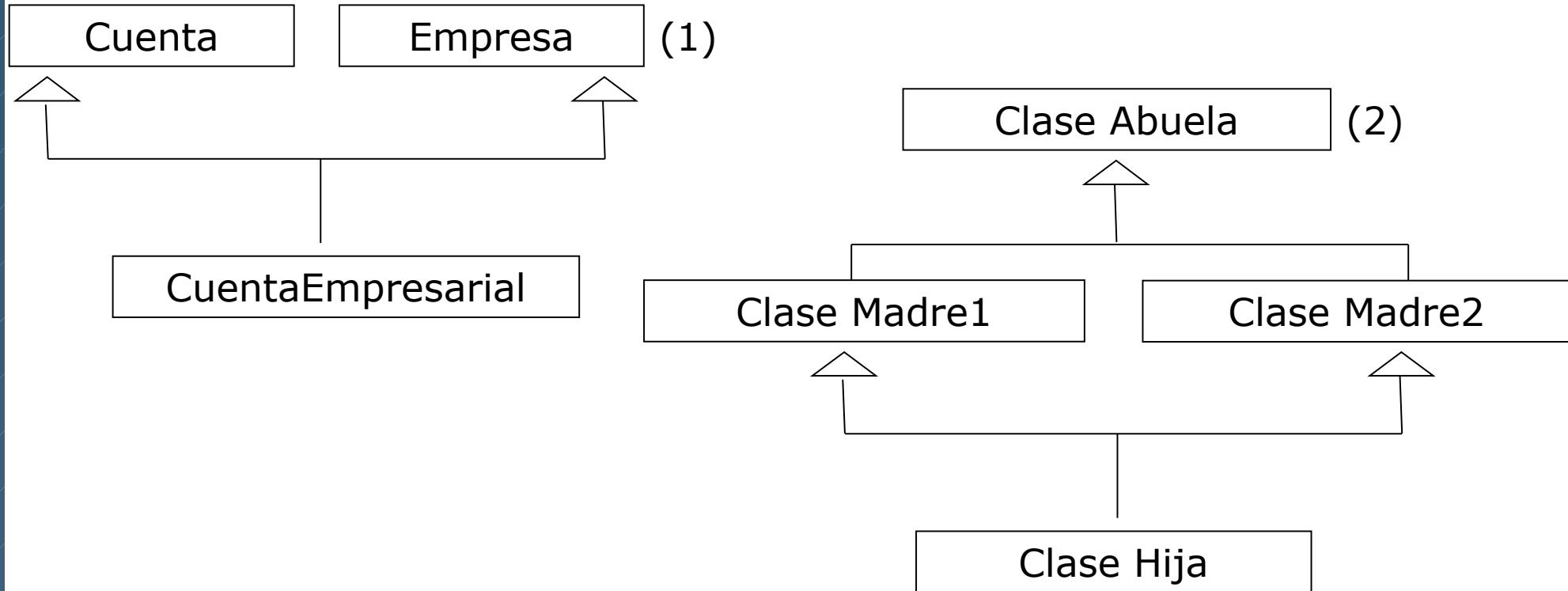
¿Cómo implementar CuentaEmpresarial?

Herencia Múltiple de implementación en C++



```
class CuentaEmpresarial
: public Cuenta, public Empresa {
public:
    CuentaEmpresarial(string unNombreCuenta,
                      string unNombreEmpresa,
                      double unSaldo=0, double unInteres=0)
        : Cuenta(unNombreCuenta,unSaldo,unInteres),
          Empresa(unNombreEmpresa)
    {};
};
```

Problemas en herencia múltiple de implementación



¿Qué problemas pueden darse en (1)? ¿Y en (2)?

Resolver los nombres mediante ámbitos:

```
class CuentaEmpresarial: public TCuenta,  
public Empresa {  
...  
{ ... string n;  
if ...  
    n= Cuenta::getNombre();  
else  
    n= Empresa::getNombre();  
}  
};
```

En C++ se resuelve usando **herencia virtual**:

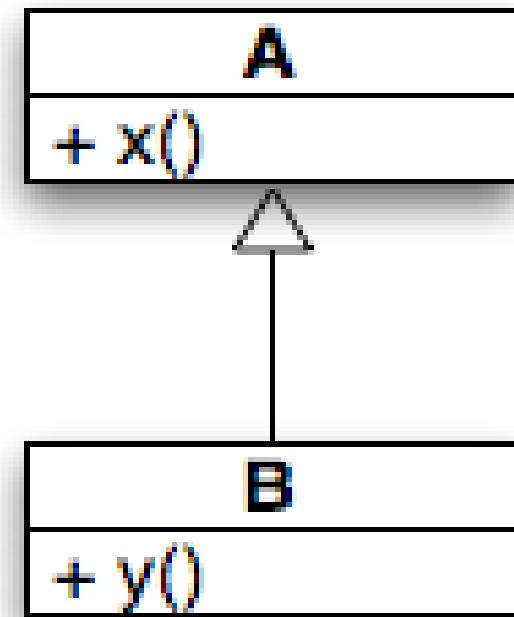
```
class Madre_1: virtual public Abuela{  
...  
}  
  
class Madre_2: virtual public Abuela{  
...  
}  
  
class Hija: public Madre_1, public Madre_2 {  
...  
    Hija( ) : Madre_1( ), Madre_2( ), Abuela( ){  
    };  
}
```

HERENCIA DE INTERFAZ

- La herencia de interfaz NO hereda código
- Sólo se hereda la interfaz (a veces con una implementación parcial o por defecto).
- Objetivos
 - Separar la interfaz de la implementación.
 - Garantizar la **sustitución**.

- **Sustitución**

```
A obj = new B();  
  
obj.x(); // OK  
obj.y(); // ERROR
```



El principio de sustitución

“Debe ser posible utilizar cualquier objeto instancia de una subclase en el lugar de cualquier objeto instancia de su superclase sin que la semántica del programa escrito en los términos de la superclase se vea afectado.”
(Liskov, 1987)

Subtipo: Una clase B, subclase de A, es un subtipo de A si podemos sustituir instancias de A por instancias de B en cualquier situación y sin ningún efecto observable.

El principio de sustitución

- Todos los LOO soportan subtipos.
 - Lenguajes fuertemente tipados (tipado estático)
 - Caracterizan los objetos por su clase
 - Lenguajes debilmente tipados (tipado dinámico)
 - Caracterizan los objetos por su comportamiento

Lenguaje fuertemente tipado:

```
funcion medir(objeto: Medible)
{...}
```

Lenguaje débilmente tipado:

```
funcion medir(objeto) {
    si (objeto <= 5)
        sino si (objeto == 0)
    ...}
```

El principio de sustitución

- **Java:** directamente
- **C++:** Subtipos sólo a través de punteros o referencias

```
class Dependiente {  
    public int cobrar() {...}  
    public void darRecibo()  
{...}  
...}  
  
class Panadero  
    extends Dependiente  
{...}  
  
Panadero p = new Panadero();  
Dependiente d1=p; // sustit.
```

```
class Dependiente {  
    public:  
        int cobrar();  
        void darRecibo();  
    ...};  
  
class Panadero  
    : public Dependiente  
{...}  
  
Panadero p;  
Dependiente& d1=p; // sustit.  
Dependiente* d2=&p; // sustit.  
Dependiente d3=p;  
// NO sustit.: object slicing
```

HERENCIA DE INTERFAZ

- **Objetivos:**
 - Reutilización de conceptos (interfaz)
 - Garantizar que se cumple el principio de sustitución
- Implementación mediante **interfaces** (Java/C#) o **clases abstractas** (C++) y **enlace dinámico**.

HERENCIA DE INTERFAZ

Tiempo de enlace

- Momento en el que se identifica el fragmento de código a ejecutar asociado a un mensaje (llamada a método) o el objeto concreto asociado a una variable.
 - **ENLACE ESTÁTICO (early or static binding)**: en tiempo de compilación
Ventaja: EFICIENCIA
 - **ENLACE DINÁMICO (late or dynamic binding)**: en tiempo de ejecución
Ventaja: FLEXIBILIDAD

HERENCIA DE INTERFAZ

Tiempo de enlace

- Tiempo de enlace de objetos

- **Enlace estático:** el tipo de objeto que contiene una variable se determina en tiempo de compilación.

```
// C++  
Circulo c;
```

- **Enlace dinámico:** el tipo de objeto al que hace referencia una variable no está predefinido, por lo que el sistema gestionará la variable en función de la naturaleza real del objeto que referencie durante la ejecución.

- Lenguajes como Smalltalk siempre utilizan enlace dinámico con variables.
- Java usa enlace dinámico con objetos y estático con los tipos escalares.

```
Figura2D f = new Circulo(); // ó new Triangulo...
```

- C++ sólo permite enlace dinámico con variables cuando éstos son punteros o referencias, y sólo dentro de jerarquías de herencia.

```
Figura2D *f = new Circulo(); // ó new Triangulo...
```

HERENCIA DE INTERFAZ

Tiempo de enlace

- Tiempo de enlace de métodos

- **Enlace estático:** la elección de qué método será el encargado de responder a un mensaje se realiza en tiempo de compilación, en función del tipo que tenía el objeto destino de la llamada en tiempo de compilación.

```
//C++ usa enlace estático por defecto
CuentaJoven tcj;
Cuenta tc;

tc=tcj; // object slicing
tc.abonarInteresMensual();
// Enlace estático: Cuenta::abonarInteresMensual()
```

- **Enlace dinámico** la elección de qué método será el encargado de responder a un mensaje se realiza en tiempo de ejecución, en función del tipo correspondiente al objeto que referencia la variable mediante la que se invoca al método en el instante de la ejecución del mensaje.

```
//Java usa enlace dinámico por defecto
Cuenta tc = new CuentaJoven(); // sustitución

tc.abonarInteresMensual();
// Enlace dinámico: CuentaJoven.abonarInteresMensual()
```

HERENCIA DE INTERFAZ

Enlace dinámico en Java

```
class Cuenta {  
  
    void abonarInteresMensual()  
    { setSaldo(getSaldo()*(1+getInteres()/100/12)); }  
    ... }  
  
class CuentaJoven extends Cuenta {  
  
    void abonarInteresMensual() { // enlace dinámico por defecto  
        if (getSaldo()>=10000) super.abonarInteresMensual();  
    }  
    ... }
```

- La clase derivada **sobreescribe** el comportamiento de la clase base
- Se pretende invocar a ciertos métodos sobreescritos desde referencias a objetos de la clase base (aprovechando el principio de sustitución).

```
Cuenta tc = new CuentaJoven(); // sustitución  
  
tc.abonarInteresMensual();  
// Enlace dinámico: CuentaJoven.abonarInteresMensual()
```

HERENCIA DE INTERFAZ

Enlace dinámico en C++

En C++ para que esto sea posible:

- El método debe ser declarado en la clase base como **método virtual** (mediante la palabra clave **virtual**). Esto indica que tiene enlace dinámico.
- La clase derivada debe proporcionar su propia implementación del método.

```
class Cuenta {  
    ...  
    virtual void abonarInteresMensual();  
    // En C++, cuando hay herencia, es aconsejable  
    // declarar siempre virtual el destructor de la clase  
    base.  
    virtual ~Cuenta();  
};
```

```
Cuenta* tc = new CuentaJoven();  
  
tc->abonarInteresMensual();  
// Enlace dinámico: CuentaJoven::abonarInteresMensual()  
delete tc; // CuentaJoven::~CuentaJoven();
```

HERENCIA DE INTERFAZ

Clases abstractas

- Alguno de sus métodos no está definido: son métodos abstractos
- Los métodos abstractos, por definición, tienen enlace dinámico
- No se pueden crear objetos de estas clases.
- Las referencias a clase abstracta apuntarán a objetos de clases derivadas.

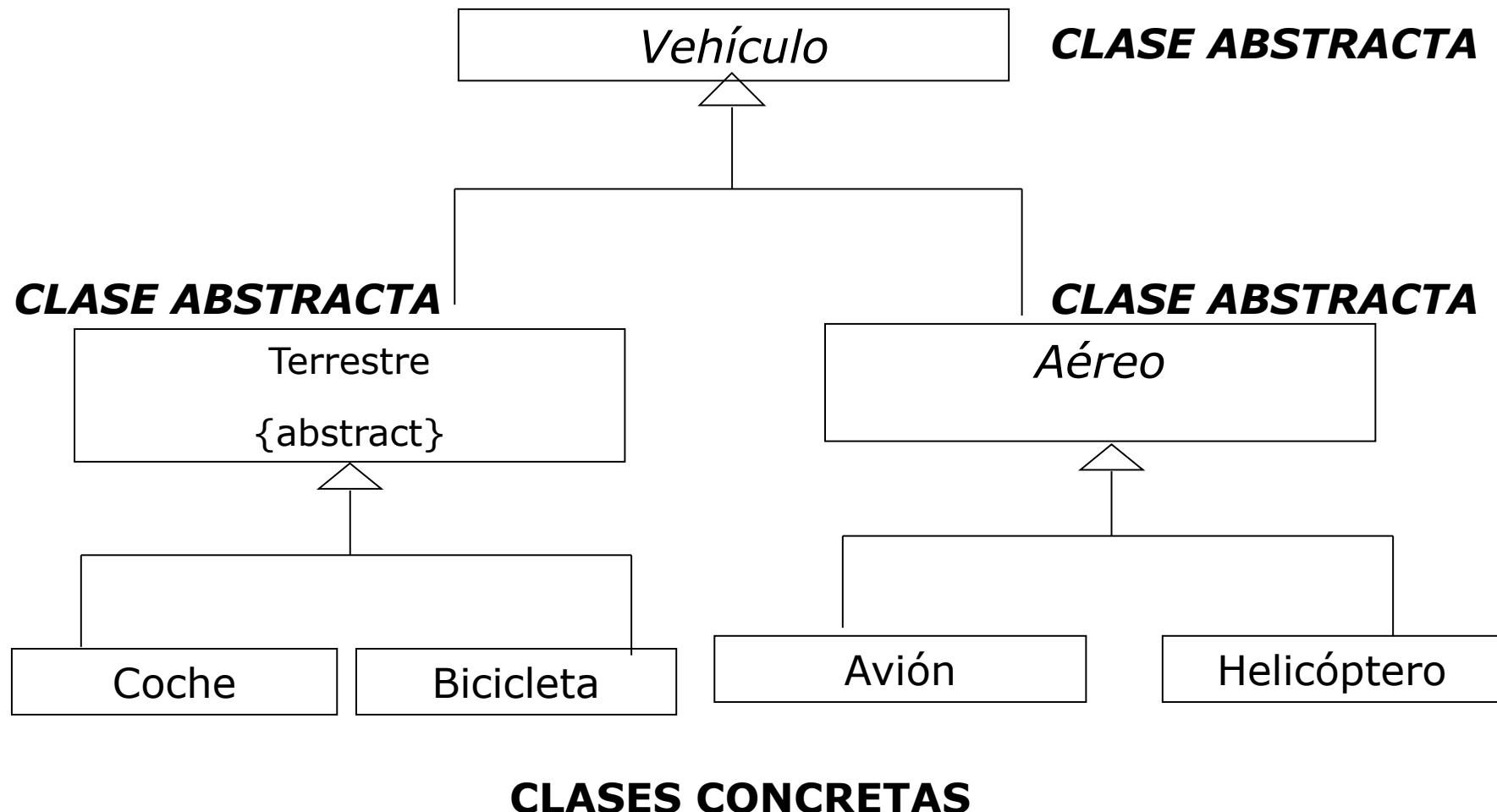
HERENCIA DE INTERFAZ

- **Clases abstractas**

- Las clases que deriven de clases abstractas (o interfaces) están obligadas a implementar todos los métodos abstractos (o serán a su vez abstractas).
- La clase derivada implementa el interfaz de la clase abstracta.
 - Se garantiza el principio de sustitución.

HERENCIA DE INTERFAZ

Notación UML para clases abstractas



HERENCIA DE INTERFAZ

- Clases abstractas en Java

```
abstract class {  
    ...  
    abstract <tipo devuelto> metodo(<lista args>);  
}
```

Clase abstracta

```
abstract class Forma  
{  
    private int posx, posy;  
    public abstract void dibujar();  
    public int getPosicionX()  
        { return posx; }  
    ...  
}
```

Clase derivada

```
class Circulo extends Forma {  
    private int radio;  
    public void dibujar()  
        {...};  
    ...  
}
```

HERENCIA DE INTERFAZ

- Clases abstractas en C++
 - Clases que contiene al menos un **metodo virtual puro** (método abstracto):

```
virtual <tipo devuelto> metodo(<lista args>) = 0;
```

Clase abstracta

```
class Forma
{
    int posx, posy;
public:
    virtual void dibujar()= 0;
    int getPosicionX()
        { return posx; }
    ...
}
```

Clase derivada

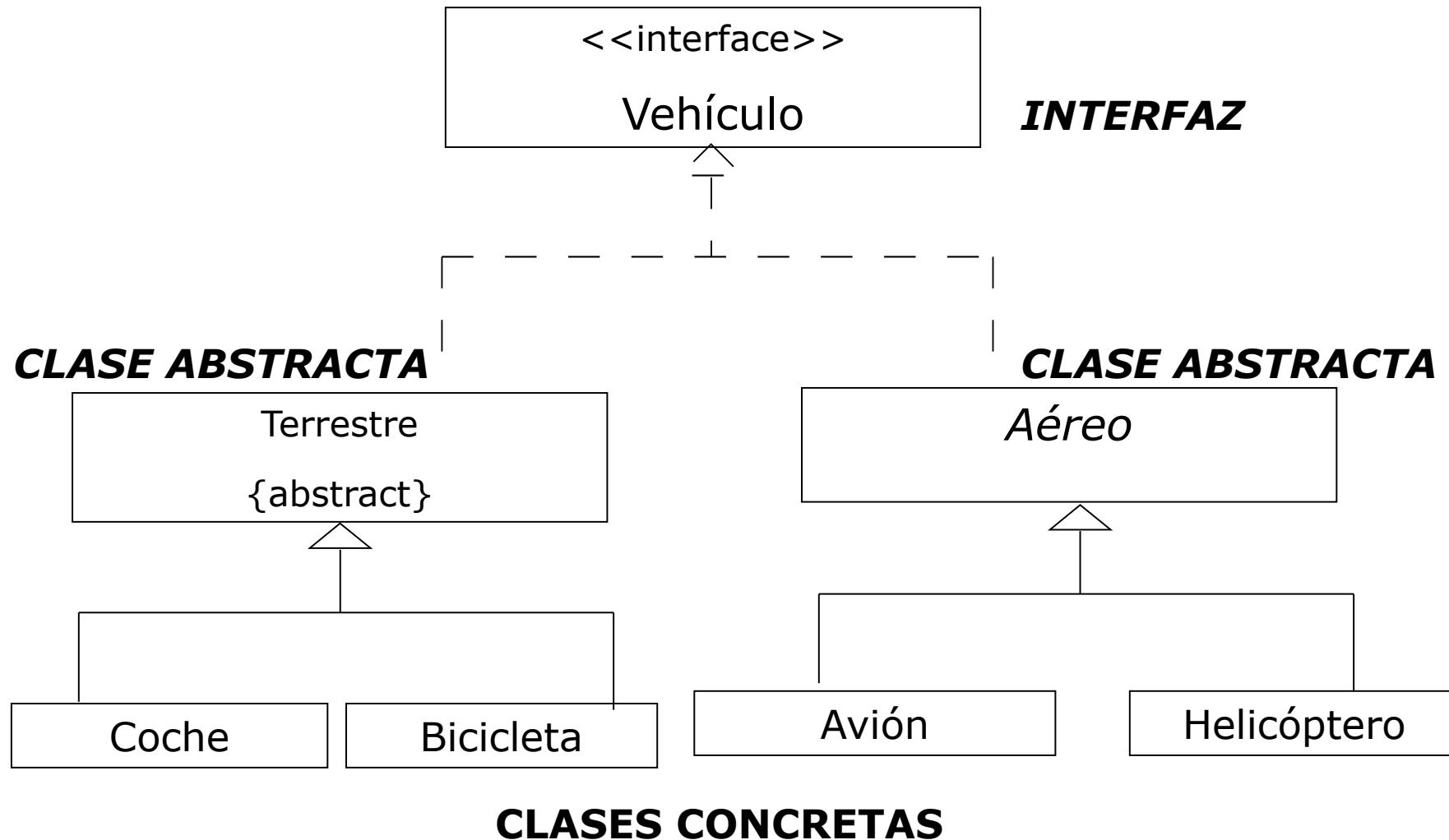
```
class Circulo : public Forma
{
    int radio;
public:
    void dibujar() {...};
    ...
}
```

HERENCIA DE INTERFAZ

- **Interfaces**
 - Declaración de un conjunto de métodos abstractos.
 - Separación total de interfaz e implementación
 - Java/C#: declaración explícita de interfaces
 - Las clases pueden implementar más de un interfaz (herencia múltiple de interfaz)

HERENCIA DE INTERFAZ

Notación UML para interfaces



HERENCIA DE INTERFAZ

- **Interfaces en Java**

```
interface Forma
{
    // - Todos los métodos son abstractos por definición
    // - Visibilidad pública
    // - Sin atributos de instancia, sólo constantes estáticas
    void dibujar();
    int getPosicionX();
    ...
}
```

```
class Circulo implements Forma
{
    private int posx, posy;
    private int radio;
    public void dibujar()
    {...}
    public int getPosicionX()
    {...}
}
```

```
class Cuadrado implements Forma
{
    private int posx, posy;
    private int lado;
    public void dibujar()
    {...}
    public int getPosicionX()
    {...}
}
```

HERENCIA DE INTERFAZ

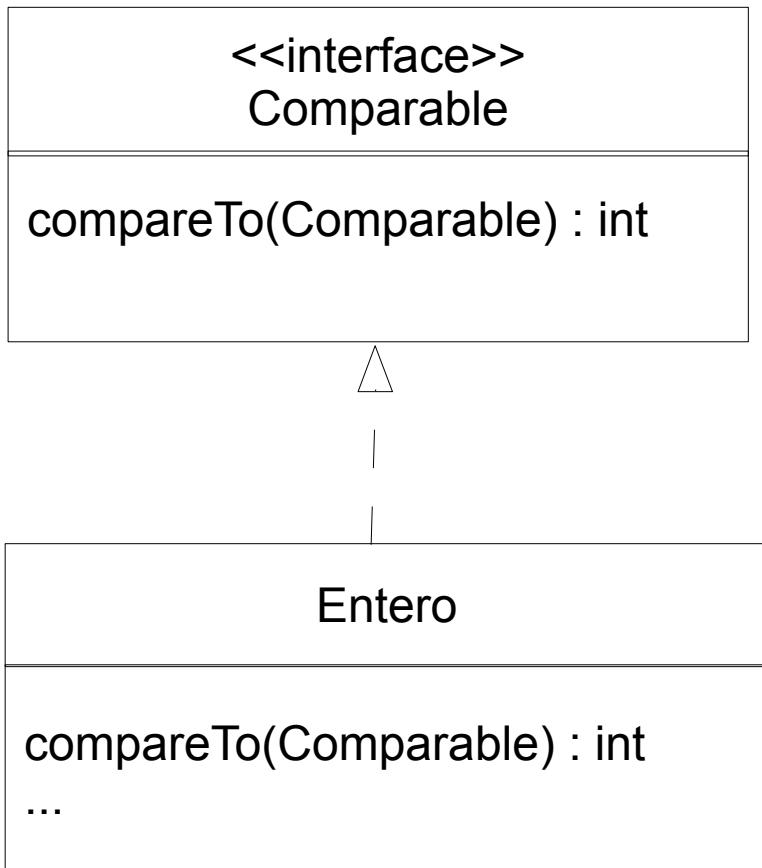
- **Interfaces en C++:** herencia pública de clases abstractas

```
class Forma
{
    // - Sin atributos de instancia
    // - Sólo constantes estáticas
    // - Todos los métodos se declaran abstractos
public:
    virtual void dibujar()=0;
    virtual int getPosicionX()=0;
    // resto de métodos virtuales puros...
}
```

```
class Circulo : public Forma // Herencia pública
{
private:
    int posx, posy;
    int radio;
public:
    void dibujar() {...}
    int getPosicionX() {...};
}
```

HERENCIA DE INTERFAZ

- Ejemplo de interfaz (Java)



```
interface Comparable {  
    int compareTo(Comparable o);  
}
```

```
class Entero implements Comparable {  
    private int n;  
  
    public Entero(int i) { n=i; }  
  
    public int compareTo(Comparable e) {  
        Entero e2=(Entero)e;  
        if (e2.n > n) return -1;  
        else if (e2.n == n) return 0;  
        return 1;  
    }  
}
```

HERENCIA DE INTERFAZ

- Ejemplo de interfaz (Java)

```
class ParOrdenado {  
    private Comparable[] par = new Comparable[2];  
  
    public ParOrdenado(Comparable p1, Comparable p2) {  
        int comp = p1.compareTo(p2);  
        if (comp <= 0) { par[0] = p1; par[1] = p2; }  
        else           { par[0] = p2; par[1] = p1; }  
    }  
    public Comparable getMenor() { return par[0]; }  
    public Comparable getMayor() { return par[1]; }  
}
```

```
// Código cliente  
  
ParOrdenado po = new ParOrdenado(new Entero(7), new Entero(3));  
  
po.getMenor(); // 3  
po.getMayor(); // 7
```

HERENCIA DE INTERFAZ

- Ejemplo de herencia múltiple de interfaz (Java)

```
interface CanFight {  
    void fight();  
}
```

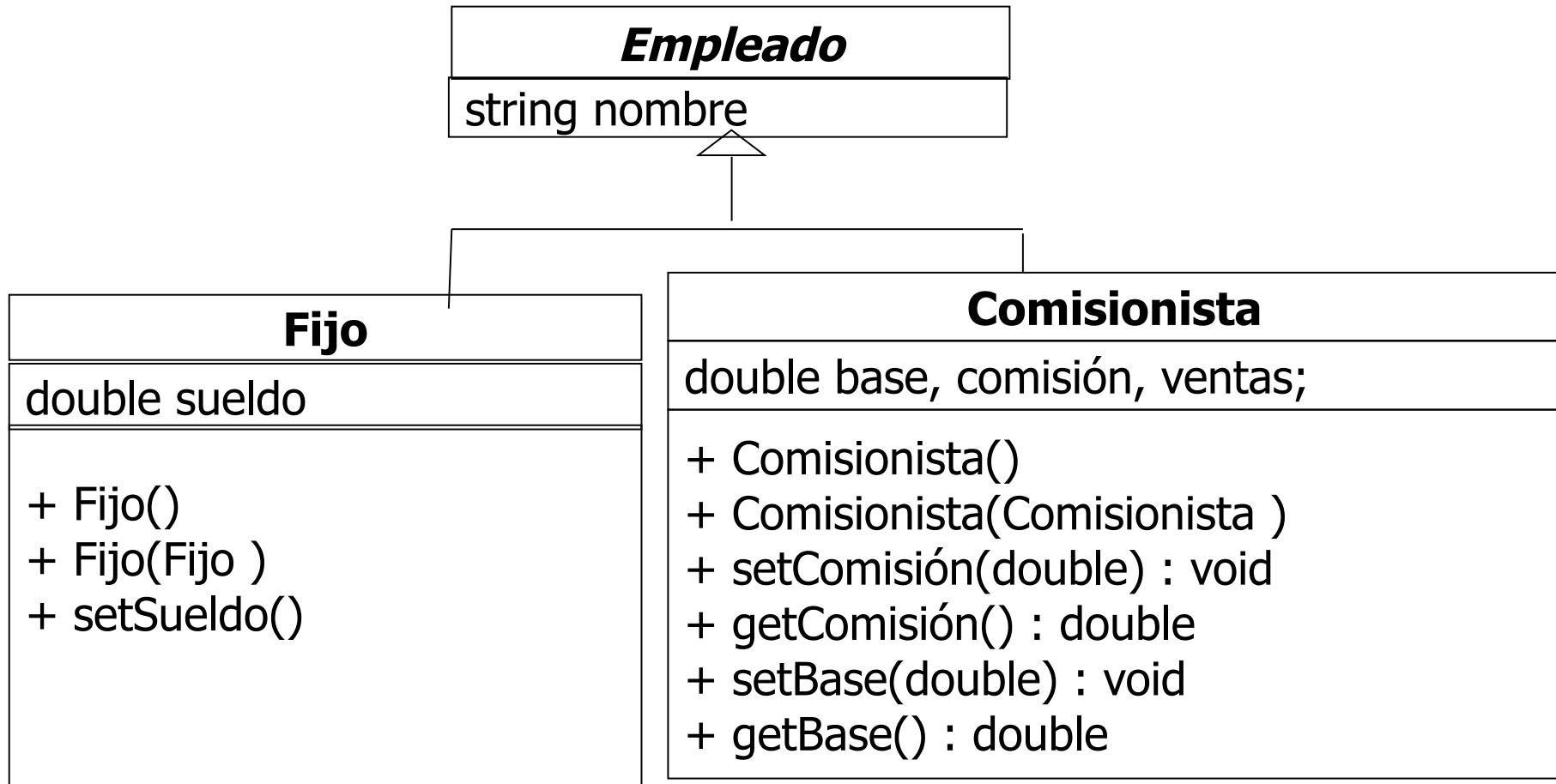
```
interface CanSwim {  
    void swim();  
}
```

```
interface CanFly {  
    void fly();  
}
```

```
class ActionCharacter {  
    public void fight() {}  
}  
  
class Hero extends ActionCharacter  
    implements CanFight, CanSwim, CanFly {  
    public void swim() {}  
    public void fly() {}  
}  
  
public class Adventure {  
    public static void t(CanFight x)  
        { x.fight(); }  
    public static void u(CanSwim x)  
        { x.swim(); }  
    public static void main(String[] args) {  
        Hero h = new Hero();  
        t(h); // Treat it as a CanFight  
        u(h); // Treat it as a CanSwim  
    }  
}
```

(tomado de 'Piensa en Java, 4^a ed.', Bruce Eckl)

Ejercicio: Pago de Nóminas



Ejercicio: Pago de Nóminas

Implementa las clases anteriores añadiendo un método `getSalario()`, que en el caso del empleado fijo devuelve el sueldo y en el caso del comisionista devuelve la base más la comisión, de manera que el siguiente código permita obtener el salario de un empleado independientemente de su tipo.

```
// código cliente
    int tipo =...; //1:fijo, 2 comisionista
    Empleado emp;

    switch (tipo) {
        case 1:
            emp=new Fijo();
            break;
        case 2:
            emp=new Comisionista();
            break;
    }
    System.out.println(emp.getSalario());
}
```

HERENCIA DE IMPLEMENTACIÓN

Uso seguro

Herencia de implementación



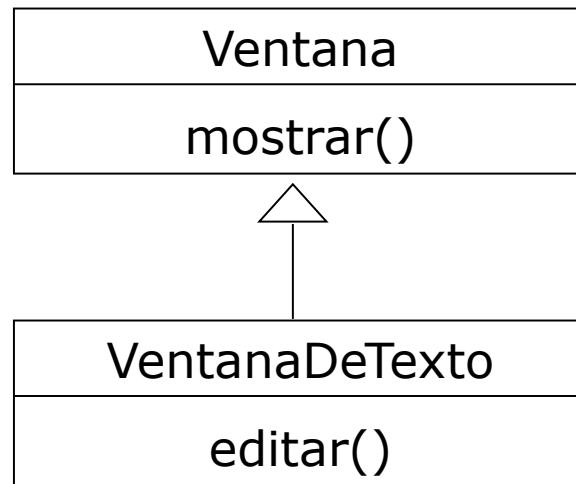
- Habilidad para que una clase herede parte o toda su implementación de otra clase.
- Debe ser utilizada con cuidado.

- En la herencia existe una tensión entre expansión (adición de métodos más específicos) y contracción (especialización o restricción de la clase padre)

- En general, la redefinición de métodos sólo debería usarse para hacer las propiedades más específicas
 - Constreñir restricciones
 - Extender funcionalidad

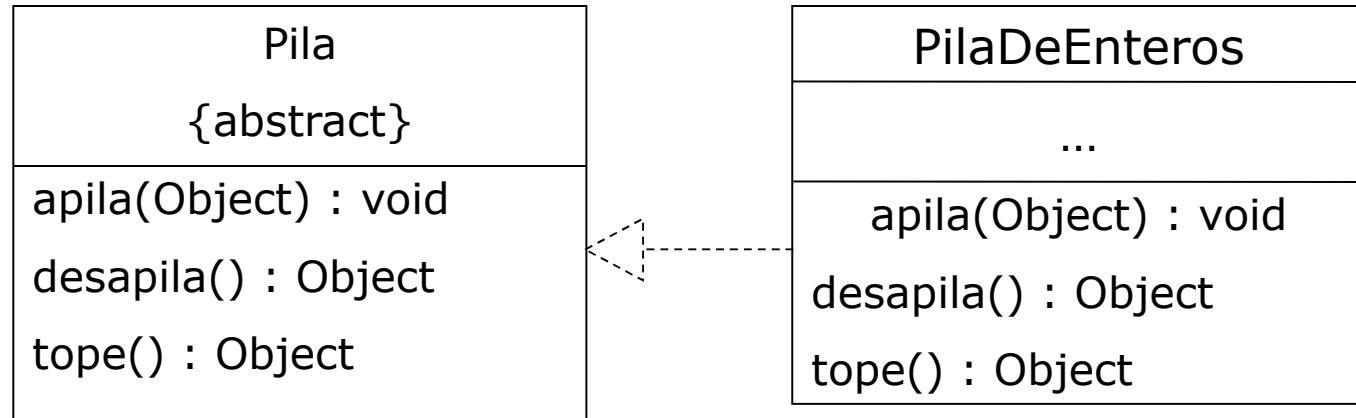
■ Especialización

- La clase derivada es una **especialización** de la clase base: añade comportamiento pero no modifica nada
 - Satisface las especificaciones de la clase base
 - Se cumple el principio de sustitución (subtipo)

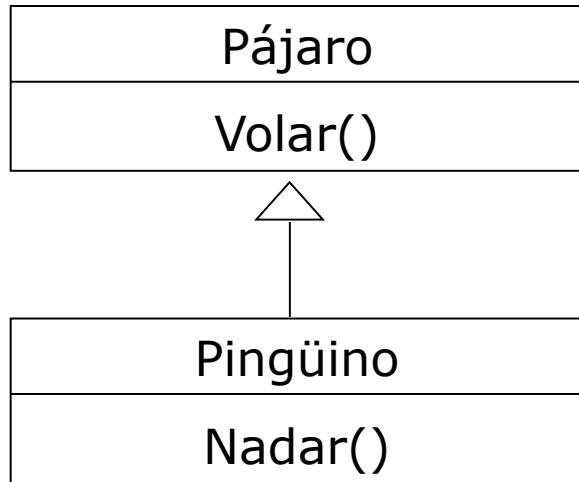


■ Especificación

- La clase derivada es una **especificación** de una clase base abstracta o interfaz.
 - Implementa métodos no definidos en la clase base (métodos abstractos o diferidos).
 - No añade ni elimina nada.
 - La clase derivada es una realización (o implementación) de la clase base.



- **Restricción (limitación)**

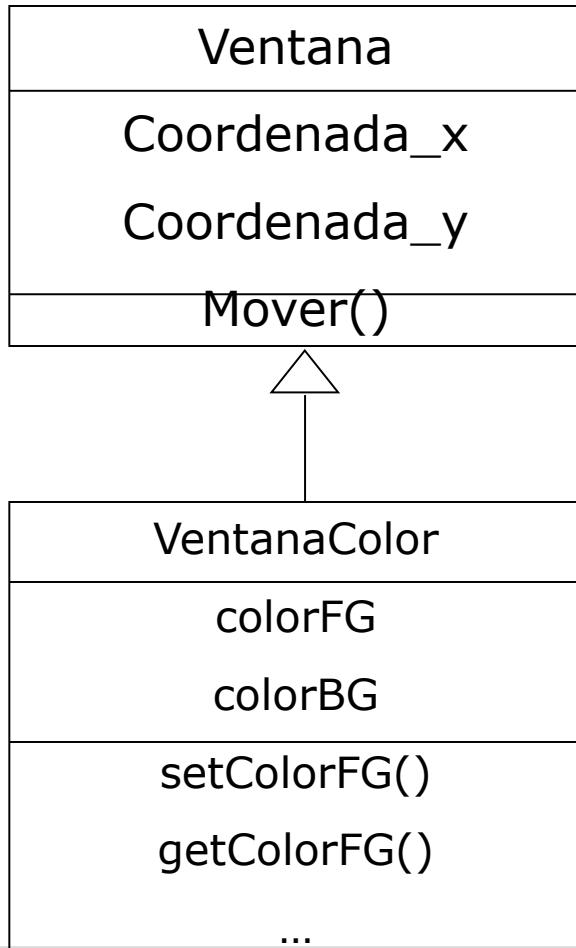


No todo lo de la clase base sirve a la derivada.

Hay que redefinir ciertos métodos para eliminar comportamiento presente en la clase base

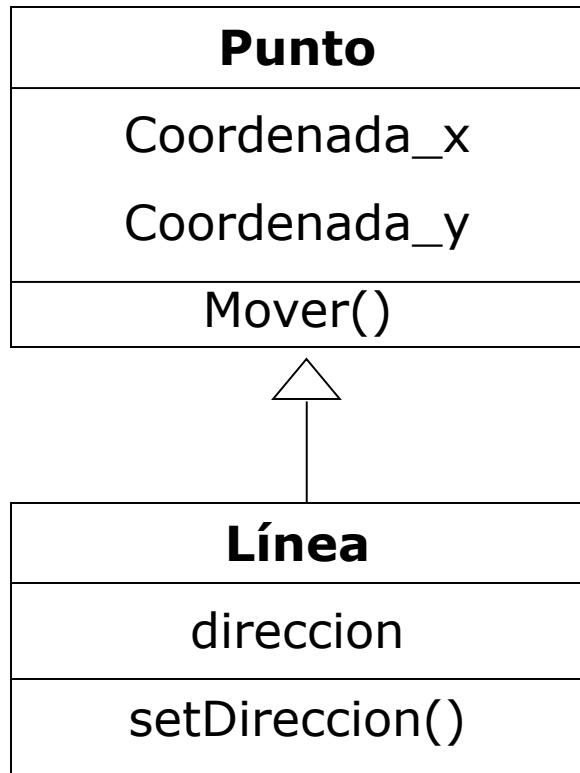
No se cumple el principio de sustitución
(un pingüino no puede volar)

■ Generalización



- Se extiende el comportamiento de la clase base para obtener un tipo de objeto más general.
- Usual cuando no se puede modificar la clase base. Mejor invertir la jerarquía.

- **Varianza (herencia de conveniencia)**



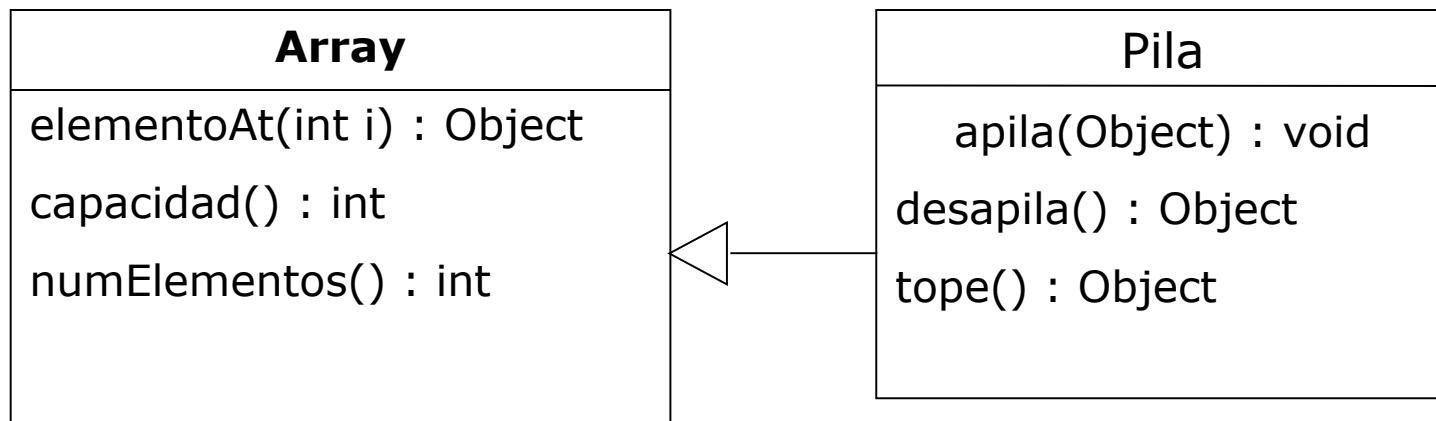
La implementación se parece pero semánticamente los conceptos no están relacionados jerárquicamente (test “es-un”).

INCORRECTA!!!!

Solución: si es posible, factorizar código común.
(p.ej. Ratón y Tableta_Grafica)

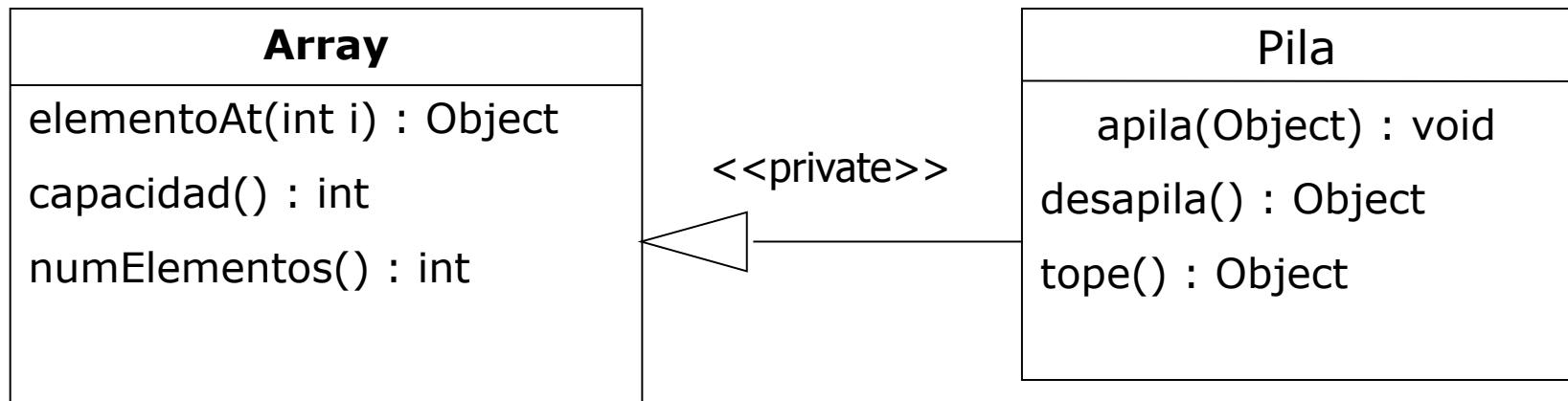
Herencia de Construcción

- También llamada ***Herencia de Implementación Pura***
- Una clase hereda parte de su funcionalidad de otra, modificando el interfaz heredado
- La clase derivada no es una especialización de la clase base (puede que incluso no haya relación “es-un”)
 - No se cumple el principio de sustitución (ni se pretende)
 - P. ej., una Pila puede construirse a partir de un Array



Herencia de Construcción en C++

- La herencia privada en C++ implementa un tipo de herencia de construcción que **sí** preserva el principio de sustitución:
 - El hecho de que Pila herede de Array no es visible para el código que usa la pila (Pila no publica el interfaz de Array).
 - Mejor usar composición si es posible.



HERENCIA

Beneficios y costes de la herencia

- Reusabilidad software
- Compartición de código
- Consistencia de interface
- Construcción de componentes
- Prototipado rápido
- Polimorfismo
- Ocultación de información

- Velocidad de ejecución
- Tamaño del programa
- Sobrecarga de paso de mensajes
- Complejidad del programa

HERENCIA

Elección de técnica de reuso

Elección de técnica de reuso

Introducción



- Herencia (IS-A) y Composición (HAS-A) son los dos mecanismos más comunes de reuso de software

- COMPOSICIÓN (Layering): Relación tener-un: ENTRE OBJETOS.
 - Composición significa contener un objeto.
 - Ejemplo: Un coche tiene un tipo de motor.

```
class Coche
{
    ...
    private Motor m;
}
```

- HERENCIA: Relación ser-un: ENTRE CLASES
 - Herencia significa contener una clase.
 - Ejemplo: Un coche es un vehículo

```
class Coche extends Vehiculo{
    ...
}
```

- **Regla del cambio:** no se debe usar herencia para describir una relación IS-A si se prevé que los componentes puedan cambiar en tiempo de ejecución (si preveo que pueda cambiar mi vocación 😊).
 - Las relaciones de composición se establecen entre **objetos**, y por tanto permiten un cambio más sencillo del programa.
- **Regla del polimorfismo:** la herencia es apropiada para describir una relación IS-A cuando las entidades o los componentes de las estructuras de datos del tipo más general pueden necesitar relacionarse con objetos del tipo más especializado (e.g. por reuso).

Elección de técnica de reuso

Introducción



- Ejemplo: construcción del tipo de dato 'Conjunto' a partir de una clase preexistente 'Lista'.
- Queremos que la nueva clase Conjunto nos permita añadir un valor al conjunto, determinar el número de elementos del conjunto y determinar si un valor específico se encuentra en el conjunto.

Lista
...
+ Lista()
+ add (int element) : void
+ firstElement() : int
+ size() : int
+ includes (int element) : boolean
+ remove (int position) : int

Conjunto
...
+ Conjunto()
+ add (int element) : void
+ size() : int
+ includes (int element) : boolean
+ remove (int element) : int

Elección de técnica de reuso

Uso de **Composición** (Layering)



- Si utilizamos la composición estamos diciendo que parte del estado de los nuevos objetos de tipo Conjunto es una instancia de una clase ya existente.

```
class Conjunto {  
  
    public Conjunto() { losDatos = new Lista(); }  
    public int size(){ return losDatos.size()(); }  
    public int includes (int el){return losDatos.includes(el);}  
    //un conjunto no puede contener valor más de una vez  
    public void add (int el){  
        if (!includes(el)) losDatos.add(el);  
    }  
  
    private Lista losDatos;  
}
```

Elección de técnica de reuso

Uso de **Composición** (Layering)



- La composición no realiza ninguna asunción respecto a la sustituibilidad. Cuando se forma de esta manera, un Conjunto y una Lista son clases de objetos totalmente distintos, y se supone que ninguno de ellos puede sustituir al otro en ninguna situación.

Elección de técnica de reuso

Uso de Herencia



- Con herencia, todos los atributos y métodos asociados con la clase base Lista se asocian automáticamente con la nueva clase Conjunto.

```
class Conjunto extends Lista {  
    public Conjunto() { super(); }  
    //un conjunto no puede contener valores repetidos  
    void add (int el){ //refinamiento  
        if (!includes(el)) super.add(el);  
    }  
}
```

- Implementamos en términos de clase base
 - No existe una lista como dato privado
- Las operaciones que actúan igual en la clase base y en la derivada no deben ser redefinidas (con composición sí).

Elección de técnica de reuso

Uso de Herencia



- El uso de la herencia asume que las subclases son además subtipos.
 - En nuestro ejemplo, un Conjunto NO ES una Lista.
 - En este caso, la composición es más adecuada.

Elección de técnica de reuso

Composición vs. Herencia



- La **composición** es una técnica generalmente **más sencilla** que la herencia.
 - Define más claramente la interfaz que soporta el nuevo tipo, independientemente de la interfaz del objeto parte.
- La **composición** es más flexible (y más resistente a los cambios)
 - La composición sólo presupone que el tipo de datos X se utiliza para IMPLEMENTAR la clase C. Es fácil por tanto:
 - Dejar sin implementar los métodos que, siendo relevantes para X, no lo son para la nueva clase C
 - Reimplementar C utilizando un tipo de datos X distinto sin impacto para los usuarios de la clase C.

- La **herencia** (pública) presupone el concepto de subtipo (principio de sustitución)
 - La herencia permite una definición más escueta de la clase
 - Requiere menos código.
 - Oferta más funcionalidad: cualquier nuevo método asociado a la clase base estará inmediatamente disponible para todas sus clases derivadas.
- Desventajas
 - Los usuarios pueden manipular la nueva estructura mediante métodos de la clase base, incluso si éstos no son apropiados.
 - Cambiar la base de una clase puede causar muchos problemas a los usuarios de dicha clase.

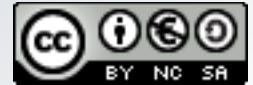
- Bruce Eckel. ***Piensa en Java 4^a edición***
 - Cap. 7 y 9
- Timothy Budd. ***An Introduction to object-oriented programming, 3rd ed.***
 - Cap 8 al 13.
- C. Cachero et al. ***Introducción a la programación orientada a objetos***
 - Cap. 3 (ejemplos en C++)

UD 6

POLIMORFISMO

Cristina Cachero, Pedro J. Ponce de León

Versión 20111024



Tema 4. Polimorfismo

Objetivos básicos



- Comprender el concepto de polimorfismo
- Conocer y saber utilizar los diferentes tipos de polimorfismo.
- Comprender el concepto de enlazado estático y dinámico en los lenguajes OO.
- Comprender la relación entre polimorfismo y herencia en los lenguajes fuertemente tipados.
- Apreciar la manera en que el polimorfismo hace que los sistemas sean extensibles y mantenibles.

Indice



1. Motivación y conceptos previos
 - Signatura
 - Ámbito
 - Sistema de tipos
1. Polimorfismo y reutilización
 - Definición
 - Tipos de polimorfismo
1. Sobrecarga
 - Sobrecarga basada en ámbito
 - Sobrecarga basada en firma de tipo
 - Alternativas a la sobrecarga
1. Polimorfismo en jerarquías de herencia
 - Redefinición
 - Shadowing
 - Sobrescritura
1. Variables polimórficas
 - La variable receptora
 - Downcasting
 - Polimorfismo puro
1. Genericidad
 - Métodos genéricos
 - Plantillas de clase
 - Herencia en clases genéricas

1. Motivación



- Objetivo de la POO
 - Aproximarse al modo de resolver problemas en el mundo real.
- El polimorfismo es el modo en que los lenguajes OO implementan el concepto de **polisemía** del mundo real:
 - Un único nombre para muchos significados, según el contexto.

1. Conceptos previos

Signatura



■ Signatura de tipo de un método:

- Descripción de los tipos de sus argumentos, su orden y el tipo devuelto por el método.
 - Notación: **<argumentos> → <tipo devuelto>**
 - Omitimos el nombre del método y de la clase a la que pertenece

■ Ejemplos

double power (double base, int exp)

■ **double * int → double**

double distanciaA(Posicion p)

■ **Posicion → double**

1. Conceptos previos:

Ámbito



■ Ámbito de un nombre:

- Porción del programa en la cual un nombre puede ser utilizado de una determinada manera.

- Ejemplo:

```
double power (double base, int exp)
```

- La variable *base* sólo puede ser utilizada dentro del método *power*

- **Ámbitos activos:** puede haber varios simultáneamente

- Las clases, los cuerpos de métodos, cualquier bloque de código define un ámbito:

```
class A {  
    private int x,y;  
    public void f() {  
        // Ámbitos activos:  
        // GLOBAL  
        // CLASE (atrib. de clase y de instancia)  
        // METODO (argumentos, var. locales)  
        if (...) {  
            String s;  
            // ámbito LOCAL (var. locales)  
        }  
    }  
}
```

1. Conceptos previos:

Ámbito: Espacio de nombres



- Un **espacio de nombres** es un ámbito con nombre

- Agrupa declaraciones (clases, métodos, objetos...) que forman una unidad lógica.
 - Java: paquetes (package)

Circulo.java

```
package Graficos;  
class Circulo {...}
```

Rectangulo.java

```
package Graficos;  
class Rectangulo {...}
```

1. Conceptos previos:

Ámbito: Espacio de nombres



- Un **espacio de nombres** es un ámbito con nombre
 - Agrupa declaraciones (clases, métodos, objetos...) que forman una unidad lógica.
 - C++: namespace

Graficos.h

(declaraciones agrupadas)

```
namespace Graficos {  
    class Circulo {...};  
    class Rectangulo {...};  
    class Lienzo {...};  
  
    ...  
}
```

Circulo.h

(cada clase en su .h)

```
namespace Graficos {  
    class Circulo {...};  
}
```

Rectangulo.h

```
namespace Graficos {  
    class Rectangulo {...};  
}
```

1. Conceptos previos:

Ámbito: Espacio de nombres



- Java : instrucción **import**

```
class Main {  
    public static void main(String args[ ]) {  
        Graficos.Circulo c;  
        c.pintar(System.out);  
    }  
}
```

```
import Graficos.*;  
class Main {  
    public static void main(String args[ ]) {  
        Circulo c;  
        c.pintar(System.out);  
    }  
}
```

1. Conceptos previos: Ámbito: Espacio de nombres



- C++: cláusula **using**

```
#include "Graficos.h"
int main() {
    Graficos::Circulo c;
    Graficos::Rectangulo r;
    c.setRadio(4);
    double a = r.getArea();
}
```

```
#include "Graficos.h"
using Graficos::Circulo;
int main() {
    Circulo c;
    Graficos::Rectangulo r;
    c.setRadio(4);
    double a = r.getArea();
}
```

1. Conceptos previos: Ámbito: Espacio de nombres



- C++ : cláusula **using namespace**

```
#include "Graficos.h"
int main() {
    Graficos::Circulo c;
    Graficos::Rectangulo r;
    c.setRadio(4);
    double a = r.getArea();
}
```

```
#include "Graficos.h"
using namespace Graficos;
int main() {
    Circulo c;
    Rectangulo r;
    c.setRadio(4);
    double a = r.getArea();
}
```

1. Conceptos previos: **Sistema de tipos**



- Un sistema de tipos de un lenguaje asocia un tipo a cada expresión, con el objetivo de evitar errores en el código. Para ello proporciona
 - Un mecanismo para definir tipos y asociarlos a las expresiones.

```
class A {} // definición de un tipo en Java/C++
A objeto; // 'objeto' es de tipo A
```

- Un conjunto de reglas para determinar la equivalencia o compatibilidad entre tipos.

```
String s = "una cadena";
int a = 10;
long b = 100;
a = s; // ERROR en Java/C++, los tipos 'String' e 'int' no son
       compatibles
b = a; // OK en Java/C++
```

1. Conceptos previos:



Sistema de tipos

- Según sea el mecanismo que asocia (enlaza) tipos y expresiones, tendremos:
 - Sistema de tipos **estático**
 - El enlace se realiza en tiempo de compilación. Las variables tienen siempre asociado un tipo.

```
String s; // (Java/C++) 's' se define como una cadena.
```
 - Sistema de tipos **dinámico**
 - El enlace se realiza en tiempo de ejecución. El tipo se asocia a los valores, no a las variables.

```
my $a; // (Perl) 'a' es una variable
$a = 1; // 'a' hace referencia a un entero...
$a = "POO"; // ... y ahora a una cadena
```

1. Conceptos previos:



Sistema de tipos

- Según las reglas de compatibilidad entre tipos, tendremos:

- Sistema de tipos **fuerte**

- Las reglas de conversión implícita entre tipos del lenguaje son muy estrictas:

```
int a=1;  
bool b=true;  
a=b; // ERROR
```

- Sistema de tipos **débil**

- El lenguaje permite la conversión implícita entre tipos

```
int a=1;  
bool b=true;  
a=b; // OK
```

Nota: 'fuerte' y 'débil' son términos relativos: un lenguaje puede tener un sistema de tipos más fuerte/débil que otro.

1. Conceptos previos:



Sistema de tipos

- El **sistema de tipos** de un lenguaje determina su soporte al enlace dinámico:
 - **Lenguajes Procedimentales**: habitualmente tiene sistemas de tipos estáticos y fuertes y en general no soportan enlace dinámico: el *tipo* de toda expresión (identificador o fragmento de código) se conoce en tiempo de compilación.
 - C, Fortran, BASIC
 - **Lenguajes orientados a objetos**:
 - Con sistema de tipos estático (C++, Java, C#, Objective-C, Pascal,...)
 - Sólo soportan enlace dinámico dentro de la jerarquía de tipos a la que pertenece una expresión (identificador o fragmento de código).
 - Con sistema de tipos dinámico (Javascript, PHP, Python, Ruby,...)
 - soportan enlace dinámico (obviamente)

Indice



1. Motivación y conceptos previos
 - Signatura
 - Ámbito
 - Sistema de tipos
1. Polimorfismo y reutilización
 - Definición
 - Tipos de polimorfismo
1. Sobrecarga
 - Sobrecarga basada en ámbito
 - Sobrecarga basada en firma de tipo
 - Alternativas a la sobrecarga
1. Polimorfismo en jerarquías de herencia
 - Redefinición
 - Shadowing
 - Sobrescritura
1. Variables polimórficas
 - La variable receptora
 - Downcasting
 - Polimorfismo puro
1. Genericidad
 - Métodos genéricos
 - Plantillas de clase
 - Herencia en clases genéricas

2. Polimorfismo

Definición



- **Capacidad de una entidad de referenciar distintos elementos en distintos instantes de tiempo.**
- Estudiaremos cuatro formas de polimorfismo, cada una de las cuales permite una forma distinta de **reutilización de software**:
 - Sobrecarga
 - Sobreescritura
 - Variables polimórficas
 - Genericidad

Tipos de polimorfismo



- **Sobrecarga** (*Overloading*, Polimorfismo ad-hoc)
 - Un sólo nombre de método y muchas implementaciones distintas.
 - Las funciones sobrecargadas normalmente se distinguen en tiempo de compilación por tener distintos parámetros de entrada y/o salida.

Factura.**imprimir()**

Factura.**imprimir(int numCopias)**

ListaCompra.**imprimir()**

- **Sobreescritura** (*Overriding*, Polimorfismo de inclusión)
 - Tipo especial de sobrecarga que ocurre dentro de relaciones de herencia en métodos con enlace dinámico.
 - Dichos métodos, definidos en clases base, son refinados o reemplazados en las clases derivadas.

Tipos de polimorfismo



- **Variables polimórficas** (Polimorfismo de asignación)
 - Variable que se declara con un tipo pero que referencia en realidad un valor de un tipo distinto (normalmente relacionado mediante herencia).

```
Figura2D fig = new Circulo();
```

- **Genericidad** (plantillas o *templates*)
 - Clases o métodos parametrizados (algunos elementos se dejan sin definir).
 - Forma de crear herramientas de propósito general (clases, métodos) y especializarlas para situaciones específicas.

```
Lista<Cliente> clientes;  
Lista<Articulo> articulos;  
Lista<Alumno> alumnos;
```

Indice



1. Motivación y conceptos previos
 - Signatura
 - Ámbito
 - Sistema de tipos
1. Polimorfismo y reutilización
 - Definición
 - Tipos de polimorfismo
1. Sobrecarga
 - Sobrecarga basada en ámbito
 - Sobrecarga basada en firma de tipo
 - Alternativas a la sobrecarga
1. Polimorfismo en jerarquías de herencia
 - Redefinición
 - Shadowing
 - Sobrescritura
1. Variables polimórficas
 - La variable receptora
 - Downcasting
 - Polimorfismo puro
1. Genericidad
 - Métodos genéricos
 - Plantillas de clase
 - Herencia en clases genéricas

3. Sobrecarga (*Overloading*, polimorfismo *ad-hoc*)

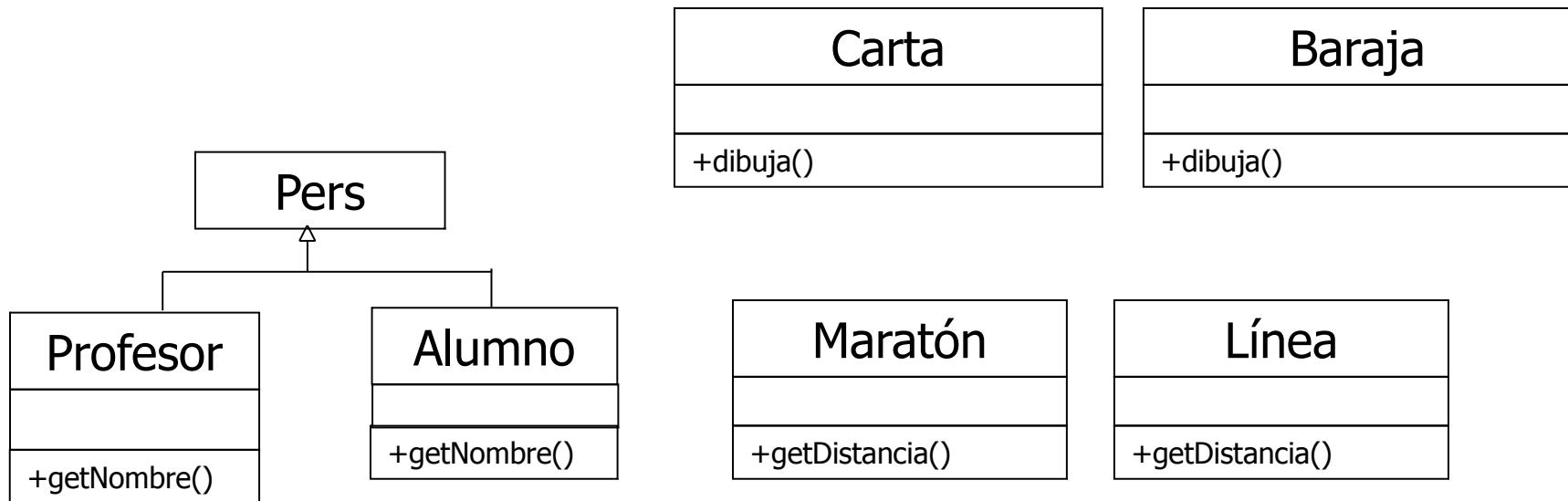


- Un mismo nombre de mensaje está asociado a varias implementaciones
- La sobrecarga se realiza en **tiempo de compilación** (enlace estático) en función de la signatura completa del mensaje.
- Dos tipos de sobrecarga:
 - **Basada en ámbito:** Métodos con **diferentes ámbitos de definición**, independientemente de sus signaturas de tipo.
 - P. ej. método `toString()` en Java.
 - **Basada en signatura:** Métodos con **diferentes signaturas de tipo en el mismo ámbito de definición**.

Sobrecarga basada en ámbito



- Distintos ámbitos implican que el mismo nombre de método puede aparecer en ellos sin ambigüedad.
- La firma de tipo puede ser la misma.



- ¿Son Profesor y Alumno ámbitos distintos?
- ¿Y Pers y Profesor?

Sobrecarga basada en **signaturas de tipo**



- Métodos en el mismo ámbito pueden compartir el mismo nombre siempre que difieran en número, orden y tipo de los argumentos que requieren (el tipo devuelto no se tiene en cuenta).
 - C++ y Java permiten esta sobrecarga de manera implícita siempre que la selección del método requerido por el usuario pueda establecerse de manera no ambigua en tiempo de compilación.
 - Esto implica que la firma no puede distinguirse sólo por el tipo de retorno

```
int f() {}  
string f() {}  
System.out.println( f() ); // ???
```

Suma
+add(int a) : int
+add(int a, int b) : int
+add(int a, double c) : double

Sobrecarga basada en signaturas de tipo



- Ejercicio: Si usamos sobrecarga basada en firma de tipos, y los métodos tienen enlace estático ¿qué ocurre cuando los tipos son diferentes pero relacionados por herencia?

```
class Base{...}

class Derivada extends Base { ... }

class Cliente {

    public static void Test (Base b)
        {System.out.println("Base");}
    public static void Test (Derivada d) // sobrecarga
        {System.out.println("Derivada");}
    public static void main(String args[]) {
        Base obj;
        if (...) obj = new Base();
        else obj = new Derivada(); //ppio de sustitución
        Test (obj); //¿a quién invoco?
    }
}
```



- **No todos los LOO permiten la sobrecarga:**
 - Permiten sobrecarga de métodos y operadores: C++
 - Permiten sobrecarga de métodos pero no de operadores: Java, Python, Perl
 - Permiten sobrecarga de operadores pero no de métodos: Eiffel

Sobrecarga de operadores en C++



- Dentro de la sobrecarga basada en signaturas de tipo, tiene especial relevancia la **sobrecarga de operadores**
- **Uso:** Utilizar operadores tradicionales con tipos definidos por el usuario.
- Forma de sobrecargar un operador @ en C++:
`<tipo devuelto> operator@(<args>)`
- Para utilizar un operador con un objeto de tipo definido por usuario, éste debe ser sobrecargado.
 - Definidos por defecto: operador de asignación (=) y el operador de dirección (&)

Sobrecarga de operadores en C++



- En la sobrecarga de operadores no se puede cambiar
 - Precedencia (qué operador se evalúa antes)
 - Asociatividad $a=b=c \rightarrow a= (b=c)$
 - Aridad (operadores binarios para que actúen como unarios o viceversa)
- No se pueden crear nuevos operadores
- No se pueden sobrecargar operadores para tipos predefinidos.
- Algunos operadores no se pueden sobrecargar: ".", ".*", "::", **sizeof**, "? :"

Sobrecarga de operadores en C++



- La sobrecarga de operadores se puede realizar mediante funciones miembro o no miembro de la clase.
 - Como función miembro: el operando de la izquierda (en un operador binario) debe ser un objeto (o referencia a un objeto) de la clase.
 - Ejemplo: sobrecarga de + para la clase Complejo:

```
Complejo Complejo::operator+(const Complejo&)  
...  
Complejo c1(1,2), c2(2,-1);  
c1+c2; // c1.operator+(c2);  
c1+c2+c3; // c1.operator+(c2).operator+(c3)
```

Sobrecarga de operadores en C++



- Como función no miembro:
 - Útil cuando el operando de la izquierda no es miembro de la clase

Ejemplo: sobrecarga de operadores << y >> para la clase Complejo:

```
ostream& operator<<(ostream&, const Complejo&);  
istream& operator>>(istream&, Complejo&);  
...  
Complejo c;  
cout << c; // operator<<(cout, c)  
cin >> c; // operator>>(cin, c)
```



■ **Funciones poliádicas**

- Funciones con número variable de argumentos
- Se encuentran en distintos lenguajes
 - P. ej. printf de C y C++

- Si el número máximo de argumentos es conocido, en C++ podemos acudir a la definición de valores por defecto:

```
int sum (int e1, int e2, int e3=0, int e4=0);
```



■ **Métodos poliádicos en Java**

```
void f(Object... args)
{   for (Object obj : args) {...} }
```

```
f("A", new A(), new Float(10.0));
f();
```

```
void g(int a, int... resto) {...}
```

```
g(3,"A","B"); g(3);
```

- Los métodos poliádicos complican la sobrecarga. Deben usarse con precaución.

Alternativas a sobrecarga: Coerción y Conversión



■ COERCIÓN

- Un valor de un tipo se convierte DE MANERA IMPLÍCITA en un valor de otro tipo distinto

- P. ej. Coerción implícita entre reales y enteros en C++/Java.

```
double f(double x) {...}  
f(3); // coerción de entero a real
```

- El principio de sustitución en los LOO introduce además una forma de coerción que no se encuentra en los lenguajes convencionales

- // ppio. sustitución (coerción entre punteros)

```
class B extends A {...}  
B pb = new B();  
A pa = pb;
```

Alternativas a sobrecarga: Coerción y Conversión



■ CONVERSIÓN

- Cambio en tipo de manera explícita
- Operador de conversión: se denomina CAST
 - Ejemplo:

```
double x; int i;  
x= i + x; // COERCION  
x= (double)i + x; // CONVERSION
```

- Java permite
 - Conversión entre tipos escalares (excepto boolean)
 - Entre tipos relacionados por herencia (upcasting, downcasting)
 - Otro tipo de conversiones: mediante métodos específicos:
 - `Integer.valueOf("15.4");`

Alternativas a sobrecarga: Coerción y Conversión



■ CONVERSIÓN en C++

- Definición de operación de conversión (*cast*) en C++:
 - De un tipo externo al tipo definido por la clase:
 - Constructor con un solo parámetro del tipo desde el cual queremos convertir.
 - Del tipo definido por la clase a otro tipo distinto:
 - Implementación de un operador de conversión.

```
class Fraccion{  
    private: int num, den;  
    public : operator double() {  
        return (numerador() / (double)denominador());  
    }  
};  
Fraccion f; double d = f * 3.14;
```

Indice



1. Motivación y conceptos previos
 - Signatura
 - Ámbito
 - Sistema de tipos
1. Polimorfismo y reutilización
 - Definición
 - Tipos de polimorfismo
1. Sobrecarga
 - Sobrecarga basada en ámbito
 - Sobrecarga basada en firma de tipo
 - Alternativas a la sobrecarga
1. Sobrecarga en jerarquías de herencia
 - Redefinición
 - Shadowing
 - Sobrescritura
1. Variables polimórficas
 - La variable receptora
 - Downcasting
 - Polimorfismo puro
1. Genericidad
 - Métodos genéricos
 - Plantillas de clase
 - Herencia en clases genéricas

Sobrecarga en jerarquías de herencia



- TIEMPO DE ENLACE POR DEFECTO
 - JAVA
 - Enlace dinámico para métodos de instancia públicos y protegidos.
 - Enlace estático para métodos privados, de clase (estáticos) y atributos.
 - C++
 - Enlace estático para todas las propiedades (métodos de instancia, de clase y atributos).

Sobrecarga en jerarquías de herencia



- Modificación del tiempo de enlace por defecto
 - JAVA
 - Métodos de instancia con enlace estático: **No hay**
 - En realidad, un método declarado como final en la raíz de una jerarquía de herencia, se comporta como si tuviera enlace estático.
 - `public final void doIt() {...}`
 - C++
 - Métodos de instancia con enlace dinámico:
 - `virtual void doIt();`

Sobrecarga en jerarquías de herencia



- **Shadowing:** Métodos con el mismo nombre, la misma
signatura de tipo y enlace estático:
 - Refinamiento/reemplazo en clase derivada: las signaturas de tipo
son la misma en clases base y derivadas. El método a invocar se
decide en tiempo de compilación.
- **Redefinición:** Métodos con el mismo nombre y distinta
signatura de tipo y enlace estático:
 - La clase derivada define un método con el mismo nombre que en la
base pero con **distinta signatura de tipos en los argumentos.**

Sobrecarga en jerarquías de herencia



- Dos formas de resolver la **redefinición** en LOO:
 - Modelo **merge** (Java):
 - Los diferentes significados que se encuentran en todos los ámbitos actualmente activos se unen para formar una sola colección de métodos.
 - Modelo **jerárquico** (C++):
 - Una redefinición en clase derivada oculta el acceso directo a otras definiciones en la clase base:

```
class Padre{  
    public void ejemplo(int a){System.out.println("Padre");}  
}  
class Hija extends Padre{  
    public void ejemplo (int a, int b){System.out.println("Hija");}  
}  
  
Hija h;  
h.ejemplo(3); // OK en Java  
                    //pero ERROR DE COMPILACIÓN EN C++  
h.Padre::ejemplo(3); // OK (C++)
```

Sobreescritura



- Decimos que un método en una clase derivada sobrescribe un método en la clase base si los dos métodos tienen el mismo nombre, la misma signatura de tipos y enlace dinámico.
 - El método en la clase base tiene enlace dinámico.
 - Los métodos sobrescritos en clase derivada pueden suponer un reemplazo del comportamiento o un refinamiento del método base.
 - La resolución del método a invocar se produce en **tiempo de ejecución** (enlace dinámico) en función del tipo dinámico del receptor del mensaje.

Sobrecarga en jerarquías de herencia

Sobreescritura



- En Java, al tener los métodos de instancia enlace dinámico por defecto, su reimplementación en clases derivadas implica sobreescritura.
 - No obstante, se utiliza la anotación `@Override` en la clase derivada para indicar expresamente la decisión de sobreescribir un método de la clase base.
- En Java, podemos indicar que un método no puede ser sobreescrito, mediante la palabra clave '`final`'
-

```
class Base { public void f() {} }

class Derivada {

    @Override
    // Error de compilación si 'void f()' es privada, final, o
    // escribimos mal su nombre o la lista de argumentos
    public void f() {}

}
```

Sobrecarga en jerarquías de herencia

Sobreescritura



- Java:

```
class Padre {  
    public int ejemplo(int a)  
        {System.out.println("padre");}  
    public final void f() {}  
}  
  
class Hija extends Padre {  
    @Override // anotación opcional recomendada  
    public int ejemplo (int a)  
        {System.out.println("hija");}  
    //public void f() { ... }  
    // ERROR, f() no se puede sobrescribir  
}  
  
// código cliente  
Padre p = new Hija(); //ppio. de sustitución  
p.ejemplo(10); // ejecuta Hija.ejemplo(10)
```

Sobreescritura: Covarianza



- Tipos de retorno covariantes: Al sobreescibir un método en clase derivada, podemos cambiar el tipo de retorno del método a un subtipo del especificado en la clase base:

```
class A {...}
class B extends A {...}
class Base {
    A objA = new A();
    public A getA() { return objA; } }
class Derivada {
    B objB = new B();
    @Override
    public B getA() { return objB; } }

Base b = new Derivada();
A objetoA = b.getA(); // Upcasting.
// objetoA apuntará a b.objB
```

Sobreescritura



En otros lenguajes:

C++: la clase base debe indicar que el método tiene enlace dinámico (y puede por tanto sobrescribirse).

Smalltalk: como en Java.

Object Pascal: la clase derivada debe indicar que sobreescribe un método: `procedure setAncho(Ancho: single); override;`

C#, Delphi Pascal: exigen que tanto la clase base como la derivada lo indiquen. Ej. C#:

En la base: `public virtual double Area() {...}`

En la derivada: `public override double Area() {...}`



- Es importante distinguir entre **Sobreescritura**, **Shadowing** y **Redefinición**
 - **Sobreescritura**: la signatura de tipo para el mensaje es la misma en clase base y derivada, pero el método se enlaza con la llamada en función del tipo real del objeto receptor en tiempo de ejecución.
 - **Shadowing**: la signatura de tipo para el mensaje es la misma en clase base y derivada, pero el método se enlaza en tiempo de compilación (en función del tipo declarado de la variable receptora).
 - **Redefinición**: La clase derivada define un método con el mismo nombre que en la clase base y con **distinta signatura de tipos**.

Indice



1. Motivación y conceptos previos
 - Signatura
 - Ámbito
 - Sistema de tipos
1. Polimorfismo y reutilización
 - Definición
 - Tipos de polimorfismo
1. Sobrecarga
 - Sobrecarga basada en ámbito
 - Sobrecarga basada en firma de tipo
 - Alternativas a la sobrecarga
1. Polimorfismo en jerarquías de herencia
 - Redefinición
 - Shadowing
 - Sobrescritura
1. Variables polimórficas
 - La variable receptora
 - Downcasting y RTTI
 - Polimorfismo puro
1. Genericidad
 - Métodos genéricos
 - Plantillas de clase
 - Herencia en clases genéricas



- Una variable polimórfica es aquélla que puede referenciar más de un tipo de objeto
 - Puede mantener valores de distintos tipos en distintos momentos de ejecución del programa.
- En un lenguaje con sistema de tipos dinámico todas las variables son potencialmente polimórficas
- En un lenguaje con sistema de tipos estático la variable polimórfica es la materialización del principio de sustitución.
 - En Java: las referencias a objetos.
 - En C++: punteros o referencias a clases polimórficas



Clase polimórfica

- En Java, por defecto todas las clases son polimórficas.
- En C++, clase con al menos un método virtual.
 - Podemos indicar que no se pueden crear clases derivadas con final:
 - **final class ClaseNoDerivable { ... }**

El efecto es que las referencias de tipo *ClaseNoDerivable* ya no son polimórficas: sólo pueden referenciar objetos de tipo *ClaseNoDerivable*.



- **Variables polimórficas simples**

- Figura2D img; // Puntero a clase base polimórfica que en realidad apuntará a // objetos de clases derivadas (Circulos, Cuadrados,...)

- **Variables receptoras: `this` y `super`**

- En un método, hacen referencia al receptor del mensaje.
 - En cada clase representan un objeto de un tipo distinto.

(en otros lenguajes recibe otros nombres, como 'self')



■ **Downcasting** (polimorfismo inverso):

- Conversión de una referencia a clase base a referencia a clase derivada.
- Implica 'deshacer' el ppio. de sustitución.
- Tipos
 - **Estático** (en tiempo de compilación)
 - **Dinámico** (en tiempo de ejecución)

C++ soporta ambos tipos

En Java es siempre dinámico.



■ Downcasting dinámico

- Se comprueba en tiempo de ejecución que la conversión es posible
- En Java, sólo permitido dentro de jerarquías de herencia
- Si no es posible se lanza *ClassCastException*

```
class Base {  
    public void f() {}  
}  
  
class Derivada extends Base {  
    public void f() {}  
    public void g() {}  
}  
  
// Downcasting no seguro  
Base[] x = { new Base(), new Derivada() };  
Derivada y = (Derivada)x[1]; // Downcasting OK  
y = (Derivada)x[0]; // ClassCastException thrown  
y.g();
```



- **Downcasting seguro y RTTI**
 - **RTTI: Run Time Type Information**
 - Mecanismo que proporciona información sobre tipos en tiempo de ejecución
 - Permite averiguar y utilizar información acerca de los tipos de los objetos mientras el programa se está ejecutando.
 - En particular, podemos identificar subtipos a partir de referencias al tipo base: downcasting seguro



■ RTTI: La clase Class

- Es una metaclasa cuyas instancias representan clases
- Cada clase tiene asociado un objeto Class

```
class MiClase {}  
  
MiClase c = new MiClase();  
Class clase = MiClase.class; // literal de clase  
clase = c.getClass(); // idem
```

Literal de clase: es el objeto Class que representa a MiClase



■ RTTI: La clase Class

```
class Animal {}
class Perro extends Animal {
    public void ladrar() {}
}
class PastorBelga extends Perro {}
```

```
// Downcasting seguro
Animal a = new PastorBelga();
if (a.getClass() == PastorBelga.class) // cierto
{ PastorBelga pb = (PastorBelga)a; }
if (a.getClass() == Perro.class) // falso
{ Perro p = (Perro)a; }
```



- **RTTI: instanceof**

- **Instrucción que devuelve cierto si el objeto referenciado es del tipo indicado**

```
class Animal {}  
class Perro extends Animal {  
    public void ladrar() {}  
}  
class PastorBelga extends Perro {}  
  
// Downcasting seguro  
Animal a = new PastorBelga();  
if (a instanceof PastorBelga) // cierto  
{ PastorBelga pb = (PastorBelga)a; }  
if (a instanceof Perro) // cierto  
{ Perro p = (Perro)a; }
```



- **RTTI:**

- **Class.isInstance(): instanceof dinámico**

- `instanceof` necesita conocer el nombre de la clase objetivo en tiempo de compilación
 - ¿y si no lo conozco?

```
// Downcasting seguro
Animal a = new Perro();
Animal b = new PastorBelga();

Class clasePerro = a.getClass();
if (clasePerro.isInstance(b)) { //cierto
    Perro p = (Perro)b; // seguro
    p.ladrar();
}
```



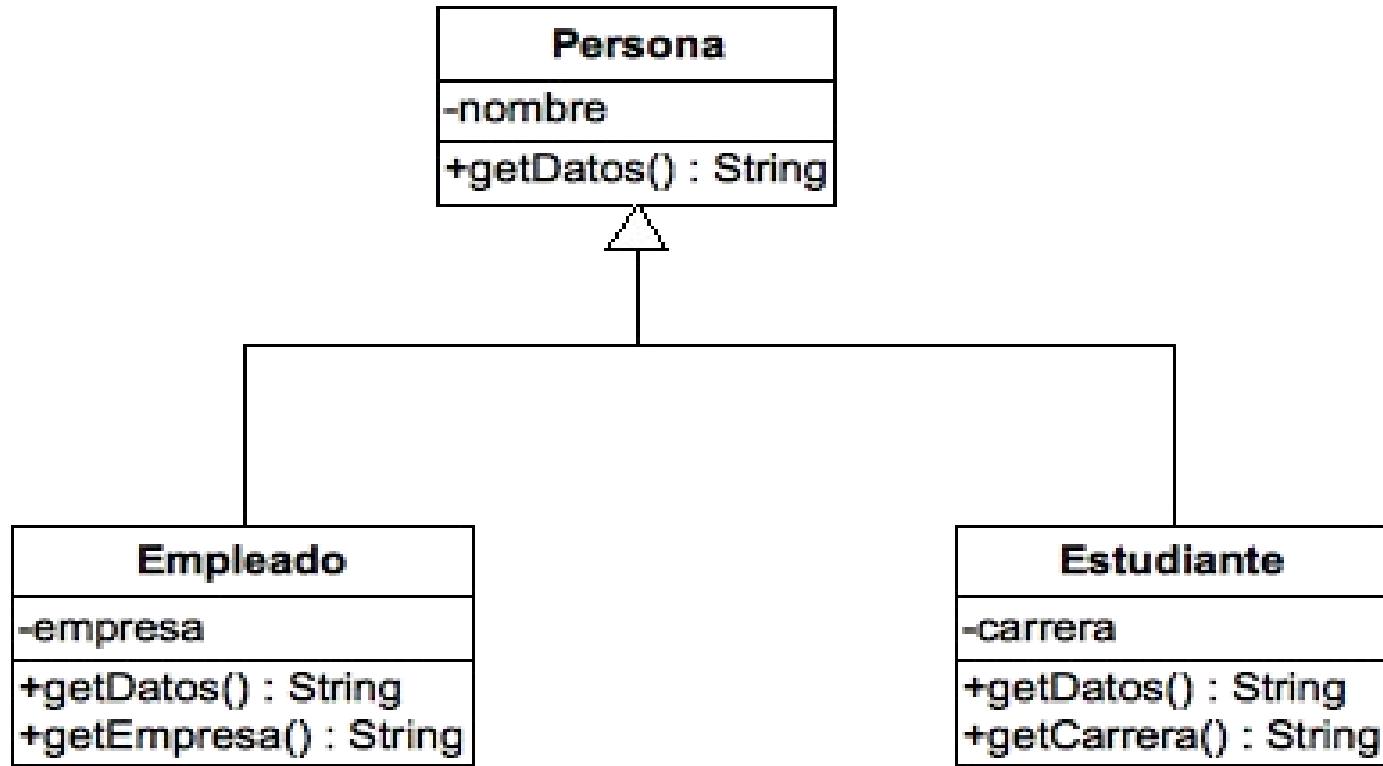
- Método con **polimorfismo puro** o **método polimórfico**
 - Alguno de sus argumentos es una variable polimórfica:
 - Un solo método puede ser utilizado con un número potencialmente ilimitado de tipos distintos de argumento.
- Ejemplo de polimorfismo puro

```
class Base { ... }
class Derivada1 extends Base { ... }
class Derivada2 extends Base { ... }

void f(Base obj) { // Método polimórfico
    // Aquí puedo usar sólo la interfaz de Base para manipular obj
    // Pero obj puede ser de tipo Base, Derivada1, Derivada2, ...
}

public static void main(String args[]) {
    Derivada1 objeto = new Derivada1();
    f(objeto); // OK
}
```

Ejemplo: Uso de polimorfismo y jerarquía de tipos



Ejemplo: Uso de polimorfismo y jerarquía de tipos



```
class Persona {  
    public Persona(String n) { nombre=n; }  
    public String getDatos() { return nombre; }  
    ...  
    private String nombre;  
}
```

Ejemplo: Uso de polimorfismo y jerarquía de tipos



```
class Empleado extends Persona {  
    public Empleado(String n, String e)  
    { super(n); empresa=e }  
    public String getDatos()  
    { return super.getDatos() + "trabaja en " + empresa; }  
    ...  
    private String empresa;  
}  
  
class Estudiante extends Persona {  
    public Estudiante(String n, String c)  
    { super(n); carrera=c }  
    public String getDatos()  
    { return super.getDatos() + " estudia " + carrera; }  
    ...  
    private String carrera;  
}
```

Refinamiento

Refinamiento

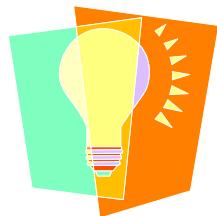
Ejemplo: Uso de polimorfismo y jerarquía de tipos



```
// código cliente
```

```
Empleado empleado =  
    new Empleado("Carlos", "Lavandería");  
Persona pers =  
    new Persona("Juan");
```

```
Empleado = pers;  
System.out.println( empleado.getDatos() );
```

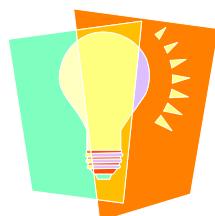


¿Qué ocurre?

Ejemplo: Uso de polimorfismo y jerarquía de tipos



```
//código cliente  
  
Empleado emp = new Empleado("Carlos", "lavanderia");  
Estudiante est = new Estudiante("Juan", "Derecho");  
Persona pers;  
pers = emp;  
System.out.println( emp.getDatos() );  
System.out.println( est.getDatos() );  
System.out.println( pers.getDatos() );
```

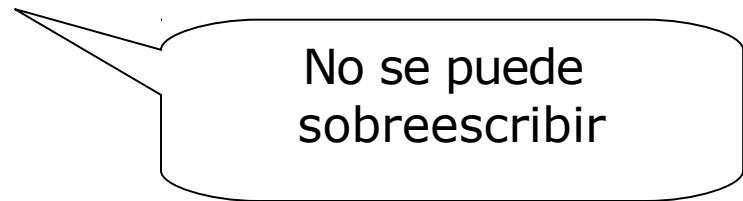


**¿Qué salida dará este programa?
¿Se produce un enlazado estático o dinámico?**

Ejemplo: Uso de polimorfismo y jerarquía de tipos



```
class Persona {  
    public Persona(String n) { nombre=n; }  
    public final String getDatos() {  
        return (nombre);  
    }  
    ...  
    private String nombre;  
}
```



Ejemplo: Uso de polimorfismo y jerarquía de tipos



```
//código cliente  
  
Empleado emp = new Empleado("Carlos", "lavanderia");  
Estudiante est = new Estudiante("Juan", "Derecho");  
Persona pers;  
pers = emp;  
System.out.println( emp.getDatos() );  
System.out.println( est.getDatos() );  
System.out.println( pers.getDatos() );
```



**¿Qué salida dará este programa?
¿Se produce un enlazado estático o dinámico?**

Ejemplo: Uso de polimorfismo y jerarquía de tipos



```
// código cliente 1  
  
Empleado uno= new Empleado("Carlos", "lavanderia");  
Persona desc = uno;  
System.out.println( desc.getEmpresa() ) ;  
  
// código cliente 2  
Persona desc = new Persona("Carlos");  
Empleado emp = (Empleado)desc;  
System.out.println( emp.getEmpresa() ) ;
```



¿Qué sucede en ambos casos?

Ejemplo: Uso de polimorfismo y jerarquía de tipos



```
// código cliente 2
Persona desc = new Persona("Carlos");
if (desc instanceof Empleado) {
    Empleado emp = (Empleado) desc;
    System.out.println( emp.getEmpresa() );
}
```



**¿Qué sucede en ambos casos?
¿Se produce un enlazado estático o dinámico?**

Polimorfismo

Implementación interna en Java



- Los métodos con enlace dinámico son algo menos eficientes que las funciones normales.
 - Cada clase no abstracta en Java dispone de un vector de punteros a métodos llamado *Tabla de métodos*. Cada puntero corresponde a un método de instancia con enlace dinámico, y apunta a su implementación más conveniente (la de la propia clase o, en caso de no existir, la del ancestro más cercano que la tenga definida)
 - Cada objeto de la clase tiene un puntero oculto a esa *tabla de métodos*.

Polimorfismo

Ventajas



- El polimorfismo hace posible que un usuario pueda añadir nuevas clases a una jerarquía sin modificar o recomilar el código escrito en términos de la clase base.
- Permite programar a nivel de clase base utilizando objetos de clases derivadas (posiblemente no definidas aún): Técnica base de las librerías/frameworks (UD 8)

Indice



1. Motivación y conceptos previos
 - Signatura
 - Ámbito
 - Sistema de tipos
1. Polimorfismo y reutilización
 - Definición
 - Tipos de polimorfismo
1. Sobrecarga
 - Sobrecarga basada en ámbito
 - Sobrecarga basada en firma de tipo
 - Alternativas a la sobrecarga
1. Polimorfismo en jerarquías de herencia
 - Redefinición
 - Shadowing
 - Sobrescritura
1. Variables polimórficas
 - La variable receptora
 - Downcasting
 - Polimorfismo puro
1. Genericidad
 - Métodos genéricos
 - Plantillas de clase
 - Herencia en clases genéricas

Genericidad

Motivación



- La genericidad es otro tipo de polimorfismo
- Para ilustrar la idea de la genericidad se propone un ejemplo:
 - Suponed que queremos implementar una función *máximo*, donde los parámetros pueden ser de distinto tipo

Genericidad

Motivación



- Solución: usar interfaces

```
interface Comparable { boolean mayorQue(Object); }
class A implements Comparable { ... }
class B implements Comparable { ... }
```

```
Comparable maximo(Comparable a, Comparable b) {
    if (a.mayorQue(b))
        return a;
    else
        return b;
}
```

```
A a1 = new A(), a2 = new A();
B b1 = new B(), b2 = new B();
A mayorA = maximo (a1,a2);
B mayorB = maximo (b1,b2);
```

Genericidad

Motivación



- Maximo() está restringido a clases que implementen el interface Comparable.
- ¿Y si queremos algo todavía más general, que funcione con cualquier tipo de objeto? Por ejemplo, una clase Lista que pueda contener cualquier tipo de dato, ya sean tipos primitivos u objetos.
 - Necesitaríamos una función genérica

Genericidad

DEFINICION



- *Propiedad que permite definir una clase o una función sin tener que especificar el tipo de todos o algunos de sus miembros o argumentos.*
 - Su utilidad principal es la de agrupar variables cuyo tipo base no está predeterminado (p. ej., listas, colas, pilas etc. de objetos genéricos: Java Collection Framework).
 - Es el usuario el que indica el tipo de la variable cuando crea un objeto de esa clase.
 - En C++ esta característica apareció a finales de los 80. En Java, existe desde la versión 1.5.

Genericidad en Java:

Genéricos



- Dos tipos de genéricos:
 - **Métodos genéricos**: son útiles para implementar funciones que aceptan argumentos de tipo arbitrario.
 - **Clases genéricas**: su utilidad principal consiste en agrupar variables cuyo tipo no está predeterminado (*clases contenedoras*)

Genericidad

Métodos genéricos



Un argumento genérico

```
public <T> void imprimeDos(T a, T b)
{
    System.out.println(
        "Primero: " + a.toString() +
        " y Segundo:" + b.toString());
}
```

```
Cuenta a = new Cuenta(),
Cuenta b = new Cuenta();
imprimeDos(a,b);
```



Inferencia de tipo de los argumentos: la realiza el compilador

Genericidad

Métodos genéricos



Más de un argumento genérico

```
public <T,U> void imprimeDos(T a, U b)
{
    System.out.println(
        "Primero: " + a.toString() +
        " y Segundo:" + b.toString() );
}
```

```
Cuenta c = new Cuenta(),
Perro p = new Perro();
imprimeDos(c,p);
```





- A continuación se plantea un ejemplo de una clase genérica *Cajon*, que va a contener un elemento de tipo genérico, no se conoce a priori.

Genericidad

Ejemplo Clase Genérica en C++



```
class Cajon<T> { // un argumento genérico: T  
  
    private T cajon;  
  
    public Cajon(T obj)  
    { cajon = obj; }  
  
    T get()  
    { return cajon; }  
}
```

Genericidad

Ejemplo de uso de una Clase Genérica



- Creación de objeto:

```
Cajon<int> ci = new Cajon<int>(10);  
Cajon<Animal> ca =  
    new Cajon<Animal>(new Perro());
```

Hay que indicar el tipo de objeto al instanciar la clase.

En el caso de 'ca', podemos almacenar ahí cualquier Animal o subtipo de Animal.

Genericidad

Herencia en clases genéricas



- Se pueden **derivar clases genéricas** de otras clases genéricas:

Clase derivada genérica:

```
class DoblePila<T> extends Pila<T>
{
    public void apilar2(T a, T b) {...}
}
```

- La clase doblePila es a su vez genérica:

```
DoblePila<float> dp = new DoblePila(10);
```



- Se pueden **derivar clases no genéricas** de una genérica:

Clase derivada no genérica:

```
class monton extends Pila<int>
{
    public void amontonar(int a) {...}
}
```

- 'monton' es una clase normal. No tiene ningún parámetro genérico.

Genericidad

Herencia en clases genéricas



- En Java, no existe relación alguna entre dos clases generadas desde la misma clase genérica, aunque los tipos estén relacionados por herencia:

```
class Uno {}  
class Dos extends Uno {}
```

```
ArrayList<Uno> u = new ArrayList<Uno>();  
ArrayList<Dos> d = new ArrayList<Dos>();  
  
u = d; // Error: incompatible types
```

Genericidad

Borrado de tipos



- Sin embargo,

```
ArrayList<Integer> v = new ArrayList<Integer>();
ArrayList<String> w = new ArrayList<String>();
System.out.println(
    v.getClass() == w.getClass() );
// imprime 'true'
//v = w; // Error: incompatible types
```

Borrado de tipos: Java no guarda información RTTI sobre tipos genéricos. En tiempo de ejecución, sólo podemos asumir que los parámetros genéricos son de tipo Object.

En tiempo de compilación, obviamente, sí existe dicha información.

Genericidad

Interfaces genéricas



- Las interfaces también pueden ser genéricas

```
interface Factoria<T>
{ T newObject(); }

class FactoriaDeAnimales implements Factoria<Animal>
{

    Animal newObject() {
        if (...) return new Perro();
        else if (...) return new Gato();
        else return new ProgramadorDeJava();
    }
}
```

Genericidad

Genericidad restringida



- *El problema con la genericidad es que sólo podemos utilizar aquellos métodos que estén definidos en Object.*
- *La genericidad restringida permite indicar que los tipos genéricos pertenezcan a una determinada jerarquía de herencia*
 - *Esto permite utilizar la interfaz de la clase usada como raíz de la jerarquía en los métodos/clases genéricos.*

```
class Perrera<T extends Perro> {  
    public void acoger(T p) { jaula.add(p); }  
    public void alimentar() {  
        for (T p : jaula)  
            if (p.ladrar()) p.alimentar();  
    }  
    private ArrayList<T> jaula = new ArrayList<T>();  
}
```

Genericidad

Comodines



```
public class NonCovariantGenerics {  
    // Compile Error: incompatible types:  
    List<Fruit> flist = new ArrayList<Apple>();  
} ///:~
```

- Una lista de manzanas NO ES una lista de frutas. Las listas de manzanas no pueden contener cualquier tipo de fruta.
- **Comodines de subtipo**

```
List<? extends Fruit> flist =  
    new ArrayList<Apple>();  
flist.add(new Apple()); // ERROR  
flist.get(0); // retorna un fruit
```



- ***Comodines de tipo base***

```
List<? super Apple> flist =  
    New ArrayList<Apple>();  
flist.add(new Apple()); // OK  
flist.add(new Fruit()); // ERROR  
flist.get(0); // retorna un Apple
```

Tema 4. Polimorfismo

Bibliografía



- Cachero et. al.
 - ***Introducción a la programación orientada a Objetos***
 - Capítulo 4
- T. Budd.
 - ***An Introduction to Object-oriented Programming, 3rd ed.***
 - Cap. 11,12,14-18; cap. 9: caso de estudio en C#
- Bruce Eckl
 - ***Piensa en Java***, 4^a edición.
 - Cap. 8, 14 y 15

UD 7

REFLEXION

Pedro J. Ponce de León

Versión 20131120





1. Motivación
 - Qué es la reflexión
2. El API de reflexión de Java
3. Objetos Class
4. La clase Array
5. Interfaz Member
6. La clase Method
7. Objetos Field
8. Mitos sobre la reflexión
9. Lo que no podemos hacer con reflexión
10. Ejemplos de uso de reflexión
 - Navegando la jerarquía de herencia
 - Escribiendo factorías de objetos con reflexión.
11. Bibliografía

Motivación. Qué es la reflexión



- Cuando nos miramos en un espejo:
 - Vemos nuestro reflejo
 - Podemos actuar según lo que vemos (ajustarnos la corbata, maquillarnos,...)
- En programación de ordenadores:
 - La reflexión es una infraestructura del lenguaje que permite a un programa conocerse y manipularse a sí mismo en tiempo de ejecución. Consiste en metadatos más operaciones que los manipulan.

(Un *metadato* es un dato que proporciona información sobre otro dato. Por ejemplo, una clase que contiene información sobre otra clase: cual es el nombre de esa clase, qué atributos o métodos tiene, etc.)

Motivación. Qué es la reflexión



- La reflexión puede usarse para
 - Construir nuevos objetos y arrays
 - Acceder y modificar atributos de instancia o de clase
 - Invocar métodos de instancia y estáticos
 - Acceder y modificar elementos de arrays
 - etc...
- La reflexión permite hacer esto sin necesidad de conocer el nombre de las clases en tiempo de compilación. Esta información puede ser proporcionada en forma de datos en tiempo de ejecución (por ejemplo, en un String).

Motivación. Qué es la reflexión



■ Ejemplo

- De un fichero de texto, leemos la cadena “Barco”, que indica que hay que crear un objeto de esa clase. Mediante reflexión, el programa actuará de la siguiente forma

- Se preguntará ¿existe una clase 'Barco' a la que tenga acceso?
 - Si es así, la cargará en memoria.

```
Class c = Class.forName("Barco")
```

- Invocará a un constructor de la clase para crear un objeto Barco.

```
Object obj = c.newInstance();
```

Fíjate que en ninguna de las dos líneas de código aparece 'Barco' como tipo.



- Java proporciona dos formas de obtener información sobre los tipos en tiempo de ejecución
 - **RTTI tradicional** (upcasting, downcasting)
 - Cuando el tipo de un objeto está disponible en tiempo de compilación y ejecución
 - **Reflexión**
 - Cuando el tipo de un objeto puede no estar disponible en tiempo de compilación y/o ejecución.

El API de reflexión de Java



- El API de reflexión de Java
 - **java.lang.Class**
 - **java.lang.reflect.***
- Representa, o refleja, las clases, interfaces y objetos de la JVM en ejecución.
- Introducida en JDK 1.1 para dar soporte a la especificación JavaBeans.



■ Clases del API de reflexión

java.lang.reflect

- El paquete de reflexión
- Introducido en JDK 1.1

java.lang.reflect.Array

- Proporciona métodos estáticos para crear y acceder dinámicamente a los arrays.

java.lang.reflect.Member

- Interfaz que refleja información sobre un miembro de una clase (atributo, método o constructor)



■ Clases del API de reflexión (cont.)

java.lang.reflect.Constructor

- Proporciona información y acceso sobre un método constructor

java.lang.reflect.Field

- Proporcionan información y acceso dinámico a un atributo de una clase.
- El atributo puede ser de clase (static) o de instancia.

java.lang.reflect.Method

- Proporciona información y acceso a un método.



■ Clases del API de reflexión (cont.)

java.lang.Class

- Representa clases e interfaces

java.lang.Package

- Proporciona información sobre un paquete.

java.lang.ClassLoader

- Clase abstracta. Proporciona servicios para cargar clases dinámicamente.

1. El objeto **Class**



- Toda clase que se carga en la JVM tiene asociado un objeto de tipo **Class**
 - Corresponde a un fichero .class
 - El *cargador de clases* (`java.lang.ClassLoader`) es el responsable de encontrar y cargar una clase en la JVM.
- **Carga dinámica de clases:** al instanciar un objeto...
 - JVM comprueba si la clase ya ha sido cargada
 - Localiza y carga la clase en caso necesario.
 - Una vez cargada, es usada para instanciar el objeto.

1. El objeto Class



Un objeto Class contiene información sobre una clase:

- Sus métodos
- Sus campos
- Su superclase
- Los interfaces que implementa
- Si es o no un array
- ...



- **Class.forName (String)**
 - Obtiene el objeto Class correspondiente a una clase cuyo nombre se pasa como argumento.

```
try
{
    // busca y carga la clase B,
    // si no estaba ya cargada
    Class c = Class.forName ("B");
    // c apuntará a un objeto Class
    // que representa a la clase B
}
catch (ClassNotFoundException e)
{ // B no existe, es decir,
  // no está en el CLASSPATH
  e.printStackTrace ();
}
```



- En el tema anterior, hemos visto otras formas de obtener el objeto Class y/o acceder a la información que contiene:

- **Literales de clase**

- Circulo.**class**

- Integer.**class**

- int.**class**

- **instanceof**

- ```
if (x instanceof Circulo)
 ((Circulo) x).setRadio(10);
```



## ■ Métodos en Class

`public String getName( );`

- Devuelve el nombre de la clase reflejada por este objeto Class.

`public boolean isInterface( );`

- Devuelve cierto si la clase es un interfaz.

`public boolean isArray( );`

- Devuelve cierto si la clase es un array.

`public Class getSuperclass( );`

- Devuelve el objeto Class que representa a la clase base de la clase actual.



## ■ Métodos en Class

```
public Class[] getInterfaces();
```

- Devuelve un array de objetos Class que representan los interfaces implementados por esta clase.

```
public Object newInstance();
```

- Crea una instancia de esta clase (invocando al constructor por defecto).



## ■ Métodos en Class

```
public Constructor[] getConstructors();
```

- Devuelve un array con todos los constructores públicos de la clase actual.

```
public Method[] getDeclaredMethods();
```

- Devuelve un array de todos los métodos públicos y privados declarados en la clase actual.

```
public Method[] getMethods();
```

- Devuelve un array de todos los métodos públicos en la clase actual, así como los declarados en todas las clases base o interfaces implementados por esta clase.



## ■ Métodos en Class

```
public Method getMethod(String methodName,
 Class[] paramTypes);
```

- Devuelve un objeto Method que representa el método público identificado por su nombre y tipo de parámetros, declarado en esta clase o heredado de una clase base.

```
public Method getDeclaredMethod(String
methodName, Class[] paramTypes);
```

- Devuelve un objeto Method que representa el método público identificado por su nombre y tipo de parámetros, declarado en esta clase. El método puede ser privado.



## ■ La clase Array

```
public class Array {

 // todo métodos static y public:
 int getLength(Object arr);
 Object newInstance(Class elements, int length);
 Object get(Object arr, int index);
 void set(Object arr, int index, Object val);

 // Varias versiones especializadas, como...
 int getInt(Object arr, int index);
 void setInt(Object arr, int index, int val);
}
```

# La clase Array



## ■ Ejemplos usando Array

```
Perro[] perrera = new Perro[10];
.
int n = Array.getLength(perrera);

// asigna uno de los elementos del array
Array.set(perrera, (n-1), new Perro("Spaniel"));

// Obtiene un objeto del array, determina su clase y
muestra su valor:

Object obj = Array.get(perrera, (n-1));
Class c1 = obj.getClass();
System.out.println(c1.getName() + "-->"
+ obj.toString());
```

# La clase Array



## ■ Dos formas de declarar un array

```
// 1:
```

```
Perro[] perrera = new Perro[10];
```

```
// 2:
```

```
Class c1 = Class.forName("Perro");
```

```
Perro[] perrera = (Perro[]) Array.newInstance(c1, 10);
```



- **Ejemplo: expandir un array**

**Enunciado del problema:**

**Escribir una función que reciba un array arbitrario, reserve memoria para dos veces el tamaño del array, copie los datos al nuevo array y lo devuelva como resultado.**



## ■ Ejemplo: expandir un array

```
// Este código no funcionará

public static Object[] doubleArrayBad(Object[] arr
{
 int newSize = arr.length * 2 + 1;

 Object[] newArray = new Object[newSize];

 for(int i = 0; i < arr.length; i++)
 newArray[i] = arr[i];

 return newArray;
}
```



## ■ Ejemplo: expandir un array

```
// Este código no funcionará

public static Object[] doubleArrayBad(Object[] arr
{
 int newSize = arr.length * 2 + 1;

 Object[] newArray = new Object[newSize];

 for(int i = 0; i < arr.length; i++)
 newArray[i] = arr[i];

 return newArray;
}
```

Este método siempre devuelve un array de Object, en lugar de un array del mismo tipo que el array que estamos copiando



## ■ Ejemplo: expandir un array

Usemos reflexión para obtener el tipo del array:

```
public Object[] doubleArray(Object[] arr)
{
 Class c1 = arr.getClass();
 if(!c1.isArray()) return null;

 int oldSize = Array.getLength(arr);
 int newSize = oldSize * 2 + 1;

 Object[] newArray = (Object[]) Array.newInstance(
 c1.getComponentType(), newSize);

 for(int i = 0; i < arr.length; i++)
 newArray[i] = arr[i];

 return newArray;
}
```

# Interface Member



- Representa a un miembro de una clase
- Implementado por Constructor, Method, y Field

`Class getDeclaringClass( )`

- Devuelve el objeto Class que representa a la clase donde se declara el miembro.

`int getModifiers( )`

- Devuelve un entero que representa los modificadores aplicados a este miembro.

`String getName( )`

- Devuelve el nombre simple del miembro.



Con un objeto Method, podemos...

- Obtener su nombre y lista de parámetros e invocarlo.
- Obtener un método a partir de una signatura, u obtener una lista de todos los métodos con esa signatura.
- Para especificar una signatura, hay que crear un array de objetos Class que representen los tipos de los parámetros del método (será de longitud cero si el método no lleva parámetros)

# Clase Method



```
public class Method implements Member
{
 public Class getReturnType();
 public Class[] getParameterTypes();
 public String getName();
 public int getModifiers();
 public Class[] getExceptionTypes();
 public Object invoke(Object obj, Object[] args);
}
```

- Los modificadores son almacenados como un patrón de bits; la clase Modifier tiene métodos para interpretar los bits.

# Clase Method. Ejemplos



- Obtener el nombre de un método:

```
(Method meth;)
```

```
String name = meth.getName();
```

- Obtener un array de tipos de los parámetros:

```
Class parms[] = meth.getParameterTypes();
```

- Obtener el tipo de retorno de un método:

```
Class retType = meth.getReturnType();
```



# Clase Method.

- **Method.invoke()**

```
public Object invoke(Object obj, Object[] args)
```

- Si los parámetros o el tipo de retorno son tipos primitivos, se usan los objetos Class de las clases 'wrapper' equivalente.
  - ejemplo: Integer.class
- El primer argumento es el objeto receptor (o null para métodos estáticos)
- El segundo argumento es la lista de parámetros.
- Inconvenientes de usar invoke( ):
  - Se ejecuta más lento que una llamada normal
  - Hay que ocuparse de todas las posibles excepciones verificadas.
  - Se pierden muchas de las comprobaciones que se realizan en tiempo de compilación.

# Clase Method. Ejemplos



- **Method.invoke()** y excepciones
- Si el método invocado lanza una excepción al ejecutarlo, invoke() lanzará InvocationTargetException
  - La excepción del método se puede obtener llamando a al método getException de InvocationTargetException
- Otras excepciones a tener en cuenta al usar invoke();
  - ¿Se cargó la clase? **ClassNotFoundException**
  - ¿Se encontró el método? **NoSuchMethodException**
  - ¿Es el método accesible? **IllegalAccessException**



- **Pasos para invocar un método mediante reflexión**
- Obtener un objeto Class, `c`.
- Obtener de `c` un objeto Method, `m`:
  - Crear un array de tipos de los parámetros del método a invocar.
  - Llamar a `getDeclaredMethod( )`, pasándole el nombre del método y el array de tipos. Devuelve `m`.
- Crear un array de Object que contenga los argumentos que queremos pasarle al método (segundo argumento de `m.invoke()`).
  - `new String[ ] { "Agua", "Fuego" }`
- Pasar el objeto receptor (o null si el método es estático) como primer argumento de `m.invoke()`.
- Llamar a `m.invoke( )`, y capturar la excepción `InvocationTargetException`



## Ejemplo: Invocar a main( )

Llamada: main( String[] args )

simplificado, sin control de errores:

```
Class cl = Class.forName(className);
Class[] paramTypes = new Class[] { String[].class };
Method m = cl.getDeclaredMethod("main", paramTypes);
Object[] args = new Object[]
 { new String[] { "Agua", "Fuego" } }
m.invoke(null, args);
```



## Invocar a un constructor

Circulo(Coordenada centro, float radio)

Llamar a `getConstructor()`, despues a `newInstance()`, capturar InstantiationException

```
Class c1 = Class.forName("Circulo");
Class[] paramTypes = new Class[] {Coordenada.class,
 float.class};
Constructor m = c1.getConstructor(paramTypes);

Object[] arguments = new Object[] {
 new Coordenada(10,20), 30 };

Figura2D c = (Figura2D) m.newInstance(arguments);
```



## Obtener objetos Field a partir de Class

```
public Field getField(String name)
 throws NoSuchFieldException, SecurityException
```

Devuelve un objeto Field público.

```
public Field[] getFields()
 throws SecurityException
```

Devuelve un array de atributos publicos en la clase actual o sus superclases.

```
public Field[] getDeclaredFields()
 throws SecurityException
```

Devuelve un array de atributos declarados en la clase actual.



## Cosas que se pueden hacer con objetos Field

- Cambiar el tipo de acceso al atributo.
- Obtener el nombre del atributo: `String getName()`
- Obtener su tipo – `Class getType()`
- Obtener su valor o asignarlo – `Object get()`
- Asignarle un valor - `void set( Object receptor, Object valor )`
- Comprobar si es igual a otro – `boolean equals(Object obj)`
- Obtener la clase donde se declara – `Class getDeclaringClass()`
- Obtener sus modificadores - `getModifiers()`



## Otros métodos en Field

- Métodos "get" específicos:
  - `boolean getBoolean( Object receptor )` obtiene el valor de un atributo de tipo booleano.
  - Tambien: `getByte`, `getChar`, `getDouble`, `getFloat`, `getInt`, `getLong`, y `getShort`
- Métodos "set" específicos:
  - `void setBoolean( Object receptor, boolean z )` asigna el valor 'z' al atributo en el objeto especificado.
  - También: `setByte`, `setChar`, `setDouble`, `setFloat`, `setInt`, `setLong`, and `setShort`



## Modificar y acceder a un atributo por reflexión Métodos `get()` y `set()` en Field

- Por ejemplo:

```
Object d = new Heroe();
Field f = d.getClass().getField("fuerza");
System.out.println(f.get(d));
```

- Posibles excepciones:

- `NoSuchFieldException`, `IllegalAccessException`

# Mitos sobre la reflexión



- “La reflexión sólo es útil para trabajar con componentes JavaBeans™”
- “La reflexión es demasiado compleja para usarla en aplicaciones de propósito general”
- “La reflexión reduce el rendimiento de las aplicaciones”
- “La reflexión no puede usarse en aplicaciones certificadas con el estándar *100% Pure Java*”

# Mitos sobre la reflexión



“La reflexión sólo es útil para trabajar con componentes JavaBeans™”

- Falso
- Es una técnica común en otros lenguajes orientados a objetos puros, como Smalltalk y Eiffel
- Ventajas
  - Ayuda a mantener software robusto
  - Permite a las aplicaciones ser más
    - Flexibles
    - Extensibles
    - Uso de plug-ins

# Mitos sobre la reflexión



“La reflexión es demasiado compleja para usarla en aplicaciones de propósito general”

- Falso
- Para la mayoría de propósitos, el uso de reflexión requiere únicamente saber llamar a unos pocos métodos.
- Es relativamente fácil adquirir dicha destreza
- La reflexión puede mejorar la reusabilidad del código.

# Mitos sobre la reflexión



“La reflexión reduce el rendimiento de las aplicaciones”

- Falso
- En realidad, la reflexión puede aumentar el rendimiento del código.
- Ventajas
  - Puede reducir y eliminar mucho código condicional.
  - Puede simplificar el código fuente y el diseño.
  - Puede aumentar enormemente las capacidades de una aplicación

# Mitos sobre la reflexión



**“La reflexión no puede usarse en aplicaciones certificadas con el estándar *100% Pure Java*”**

- Falso
- Sólo hay algunas restricciones en la certificación:
  - “El programa debe limitar las llamadas a métodos a clases que sean parte del programa o del JRE”

# Lo que no podemos hacer con reflexión



- ¿Cuales son las clases derivadas de una clase dada?
  - No es posible debido a la carga dinámica de clases en la JVM
- ¿Qué método se está ejecutando en este momento?
  - No es el propósito de la reflexión
  - Algunas librerías Java permiten saberlo.



# Ejemplos de uso de la reflexión

- Navegar por la jerarquía de herencia
- Eliminar código condicional

# Navegando por la jerarquía de herencia



- Encontrar un método heredado

Se trata de buscar un método hacia arriba en la jerarquía de herencia.

Funciona tanto para métodos públicos como no-públicos.

Sólo podemos ir 'hacia arriba', hacia la clase base

# Navegando por la jerarquía de herencia



```
Method buscaMetodo(Class cls, String nombreMetodo,
Class[] tiposParam)
{
 Method met = null;
 while (cls != null) {
 try {
 met = cls.getDeclaredMethod(nombreMetodo,
 tiposParam);
 break;
 } catch (NoSuchMethodException ex) {
 cls = cls.getSuperclass();
 }
 }
 return met;
}
```

Ejemplo: `buscaMetodo(Figura2D.class, "equals", new Class[] {} )`



# Eliminar código condicional

Considera este método 'factoría': Dado el nombre de una clase, crea un objeto de esa clase

```
public static Figura2D creaFigura2D (String s)
{
 Figura2D temp = null;
 if (s.equals ("Circulo"))
 temp = new Circulo ()
 else
 if (s.equals ("Cuadrado"))
 temp = new Cuadrado ();
 else
 if (s.equals ("Triangulo"))
 temp = new Triangulo ();
 else
 // ...
 return temp;
}
```



# Eliminar código condicional

El mismo método, con reflexión:

```
public static Figura2D creaFigura2D (String s)
{
 Figura2D temp = null;
 try
 {
 temp = (Figura2D) Class.forName (s).newInstance ();
 }
 catch (Exception e)
 {
 }
 return temp;
}
```



# Bibliografía

- **Introduction to Object Oriented Programming, 3rd Ed,**  
Timothy A. Budd  
*Cap. 25 : Reflection and Introspection*
- **Java reflection in action**
  - Ira R. Forman and Nate Forman, 2004  
*http://www.manning.com/forman/*
- **Piensa en Java, 4<sup>a</sup> ed.**
  - Bruce Eckl  
*Cap. 14 : Información de tipos*

UD 8

## FRAMEWORKS

*David Rizo Valero*

*Pedro J. Ponce de León*

Versión 20131125



# Indice



1. Introducción
2. Java Collection Framework
3. Ejemplo de un framework propio
4. Librerías y toolkits JDK
5. JDBC
6. Tratamiento de XML
7. Logging
8. Hibernate
9. Apache Commons
10. Otros frameworks



- **Librerías y frameworks:**
  - Implementan funcionalidades útiles que el propio lenguaje de programación no incorpora
- **Framework:**
  - 'Esqueletos' para fines específicos que debemos completar y personalizar mediante la implementación de interfaces, herencia de clases abstractas o ficheros de configuración.
  - El framework llama a nuestros métodos.
- **Librería:**
  - Conjunto de clases que realizan funciones más o menos concretas.
  - Nosotros llamamos a los métodos de las librerías.

# Frameworks



- Utilizan los tres mecanismos principales de la POO:
  - *Encapsulación* → ocultar implementación
  - *Polimorfismo* → mismo código para diferentes tipos de objetos
  - *Herencia* → reuso de funcionalidad.
    - Dividimos el universo en categorías, conjuntos de objetos, o sea clases, y subclases. Éstas últimas reutilizan la implementación y/o el interfaz de sus clases base.
- Esto permite

**Programar para un interfaz,  
no para una implementación**



## ■ **Inversión de control**

- El framework llama a nuestros métodos.
- El control de cuándo se llaman reside en el framework.
- El usuario del framework especifica métodos a invocar, normalmente proporcionando implementación para métodos declarados en interfaces o clases abstractas.
- Implementación del ***Principio de Hollywood***: "*no nos llames; nosotros te llamaremos*".

# Frameworks. Ejemplo de inversión de control (\*)



- Pedir que el usuario introduzca un dato:
  - Por línea de comando

```
String usuario=null;
String pregunta = null;
BufferedReader br =
 new BufferedReader(new InputStreamReader(System.in));

System.out.print("Diga su nombre: ");
usuario = br.readLine();
procesarUsuario(usuario);

System.out.print("Diga su pregunta: ");
pregunta = br.readLine();
procesarPregunta(pregunta);
```

(\*) inspirado en <http://martinfowler.com/bliki/InversionOfControl.html>

# Frameworks. Ejemplo de inversión de control (\*)



- Pedir que el usuario introduzca un dato:
  - Mediante un diálogo, usando un hipotético framework de gestión de ventanas:

```
CuadroDialogo d = new CuadroDialogo("Diga su nombre");
// registra el método a invocar cuando el usuario
// pulse tecla INTRO
d.registra(procesarUsuario);
// muestra el diálogo e interactúa con el usuario.
// El contenido del cuadro de texto se pasa al método
// procesarUsuario()
d.mostrar();

d = new CuadroDialogo("Diga su pregunta");
d.registra(procesarPregunta);
d.mostrar();
```

(\*) inspirado en <http://martinfowler.com/bliki/InversionOfControl.html>

# Frameworks. Inversión de control



- En el ejemplo de línea de comando, tenemos control absoluto sobre el flujo del programa.
- En el cuadro de diálogo, no. Parte del flujo está controlado por el framework de gestión de ventanas.
- Se ha ***invertido*** el control. El framework llama a nuestros métodos, en lugar de ser nosotros quienes llamamos al código del framework para pedirle la cadena de entrada.

# Frameworks. Inversión de control



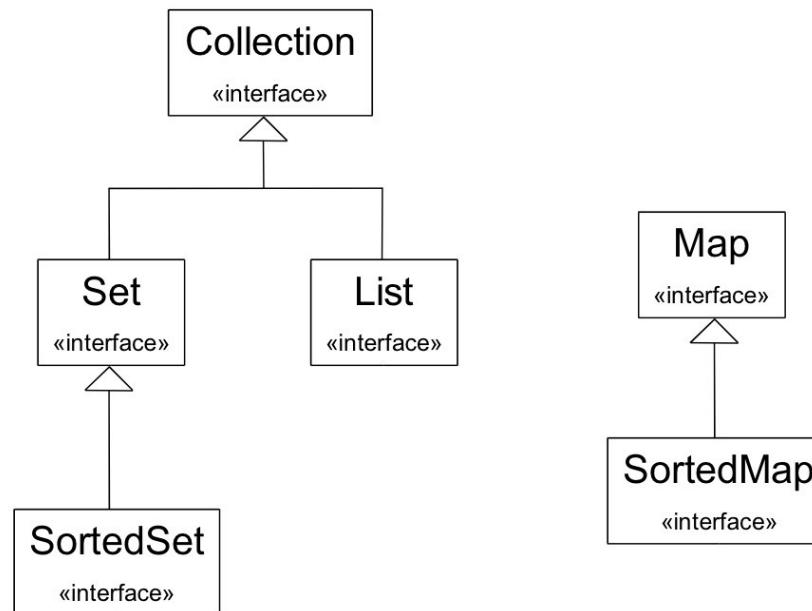
- Ejemplo: **JUnit** es un framework.
  - Funciona mediante inversión de control. Nosotros especificamos una serie de métodos que se tienen que ejecutar (métodos `setUp()`, `testAlgo()`,...). El framework se encarga de
    - Arrancar el proceso de prueba
    - Cargar las clases que contienen las pruebas
    - Invocar a los métodos `@BeforeClass`
    - Para cada test
      - Invocar a los métodos `@Before` antes de cada test
      - Invocar al método de test
      - Invocar a los métodos `@After` tras cada test
    - etc...
- JUnit controla el flujo de control, nosotros sólo indicamos qué métodos ejecutar en el momento de hacer las pruebas.

## Java Collection Framework (JCF)

# Java Collection Framework (JCF)



- Conjunto de clases incluido en el JDK representando tipos abstractos de datos básicos: pilas, colas, vectores, mapas, etc...
- Tiene formato de framework porque está diseñado para que cualquiera pueda usarlo como base para realizar su propia implementación de cada tipo de dato (extendiendo los interfaces)





```
public interface Collection {
 int size();
 boolean isEmpty();
 boolean contains(Object element);
 boolean add(Object element);
 boolean remove(Object element);
 Iterator iterator();
 boolean containsAll(Collection c);
 boolean addAll(Collection c);
 boolean removeAll(Collection c);
 boolean retainAll(Collection c);
 void clear();
 Object[] toArray();
 Object[] toArray(Object a[]);
}
```

```
public interface Map {
 Object put(Object key, Object value);
 Object get(Object key);
 Object remove(Object key);
 boolean containsKey(Object key);
 boolean containsValue(Object value);
 void putAll(Map t);
 public Set keySet();
 public Collection values();
 public Set entrySet();
}

public interface Entry {
 Object getKey();
 Object getValue();
 Object setValue(Object value);
}
```

```
public interface SortedSet extends Set {
 SortedSet subSet(Object fromElement, Object
toElement);
 SortedSet headSet(Object toElement);
 SortedSet tailSet(Object fromElement);
 Object first();
 Object last();
 Comparator comparator();
}
```

```
public interface Set
 extends Collection {
 // intentionally empty.
}
```



```
public interface List extends Collection {
 Object get(int index);
 Object set(int index, Object element);
 void add(int index, Object element);
 Object remove(int index);
 boolean addAll(int index, Collection c);
 int indexOf(Object o);
 int lastIndexOf(Object o);
 ListIterator listIterator();
 ListIterator listIterator(int index);
 List subList(int from, int to);
}
```

```
public interface SortedMap extends Map {
 SortedMap subMap(Object fromKey, Object toKey);
 SortedMap headMap(Object toKey);
 SortedMap tailMap(Object fromKey);
 Object first();
 Object last();
 Comparator comparator();
}
```



- Iteradores

```
public interface Iterator {
 boolean hasNext();
 Object next();
 void remove();
}
```

```
public interface ListIterator extends Iterator {
 void add(Object o);
 int nextIndex();
 boolean hasPrevious();
 Object previous();
 int previousIndex();
 void set(Object o);
}
```



- JCF proporciona implementaciones de referencia para cada interface
  - Si sólo usamos esas implementaciones, usaremos JCF como **librería y no como framework**

| Interfaces | Implementations |           |          |            |
|------------|-----------------|-----------|----------|------------|
|            | Hash            | Resizable | Balanced | Linked     |
| Table      | ArrayList       | Tree      | List     |            |
| Set        | HashSet         |           | TreeSet  |            |
| List       |                 | ArrayList |          | LinkedList |
| Map        | HashMap         |           | TreeMap  |            |

# JCF: algoritmos



```
public class Collections {
 public static int binarySearch(List list, Object key) {/*código*/}
 public static void copy(List dest, List src) {/*código*/}
 public static void fill(List list, Object o) {/*código*/}
 public static Object max(Collection coll) {/*código*/}
 public static Object min(Collection coll) {/*código*/}
 public static void reverse(List list) {/*código*/}
 public static void shuffle(List list) {/*código*/}
 public static void shuffle(List list, Random rnd) {/*código*/}
 public static void sort(List list) {/*código*/}
 public static void sort(List list, Comparator c) {/*código*/}
 // etc...
}
```

- JCF no tiene una implementación de referencia de Comparator

```
public interface Comparator {
 int compare(Object o1, Object o2);
 void equals(Object obj);
}
```

# JCF: algoritmos



- Uso del framework en modo librería  
(imprime: [2,7,9])

```
SortedSet<Integer> cjtoOrdenado = new TreeSet<Integer>();
cjtoOrdenado.add(7);
cjtoOrdenado.add(2);
cjtoOrdenado.add(9);

System.out.println(cjtoOrdenado.toString());
```

- Uso como framework, nosotros completamos parte del código mediante implementación de un interfaz

```
ArrayList<MiClase> v = // inicialización
// implementación anónima del interfaz comparador
Comparador c = new Comparador<MiClase>() {
 public int compare(MiClase arg0, MiClase arg1) {/* código */}
 public boolean equals(Object arg1) {/* código */}

};
// para ordenar, el framework JCF llama a nuestro método
// de comparación entre objetos MiClase
Collections.sort(v,c);
```

## Ejemplo de un framework



## Simulación de carreras de coches

- Framework propio (de juguete) para la realización de carreras de coches
  - No necesariamente coches
- El framework nos da hecho:
  - La definición de un circuito
  - La simulación del proceso de la carrera
- Cómo usarlo
  - Especificar qué coches van a intervenir y cuantas vueltas hay que dar.
  - Extendiendo e implementando las clases que se nos indica en la documentación (hipotética)

Vehiculo, ICorredor, ICochAuxiliar

# Ejemplo de un framework



## Interfaces

```
interface ICorredor {
 void dar_vuelta ();
}
```

```
interface ICochéAuxiliar {
 boolean en_pista ();
 void toggle ();
}
```

# Ejemplo de un framework



## Clase Vehiculo

```
abstract class Vehiculo {
 public Vehiculo (String m) { marca = m; }
 public String get_marca () { return marca; }

 private String marca;
}
```

# Ejemplo de un framework



## Clase Circuito

(implementada para los interfaces ICorredor e ICochéAuxiliar)

```
class Circuito {
 private int longitudkm;
 private int aforo;
 private String nombre;
 private List<ICorredor> lv;
 private ICochéAuxiliar sc;

 public Circuito (String n) {
 sc = null;
 nombre = n;
 lv = new ArrayList<ICorredor>();
 }
 ...
```

# Ejemplo de un framework



## Clase Circuito

```
...
public int get_nvehiculos ()
 { return lv.size(); }
public int get_longitudkm ()
 { return longitudkm; }
public int get_aforo ()
 { return aforo; }

public void add_vehiculo (ICorredor c)
 { lv.add(c); }

public void add_safetycar (ICochеAuxiliar ca)
 { sc = ca; }
...
```

# Ejemplo de un framework



## Clase Circuito

```
public void simular_carrera (int nv) {
 System.out.println("Bienvenidos al circuito de " + nombre);
 if (sc != null) {
 System.out.println("Comienza la carrera:\n");
 while (nv >0) {
 System.out.println("[");
 for (ICorredor v : lv) {
 if (!sc.en_pista ())
 v.dar_vuelta ();
 else
 System.out.println("SafetyCar en pista");
 }
 System.out.println("]\n");
 nv--;

 if ((new Random()).next_int(100) > 50) {
 sc.toggle ();
 }
 }
 } else
 System.out.println("¡No hay safety-car!");
}
```

# Ejemplo de un framework



## Uso del framework

```
class Coche extends Vehiculo {
 public Coche (String marca) {
 super (marca);
 }
}
```

# Ejemplo de un framework



## Uso del framework

```
class SafetyCar extends Coche implements ICochAuxiliar {

 public SafetyCar (String marca) {
 super (marca);
 m_en_pista = false;
 }
 public boolean en_pista () { return m_en_pista; }
 public void toggle () { m_en_pista = !m_en_pista; }
 public void set_en_pista (boolean v)
 { m_en_pista = v; }

 private boolean m_en_pista;
}
```

# Ejemplo de un framework



## Uso del framework

```
class Formula1 extends Coche implements ICorredor {

 public Formula1 (String marca) {
 super (marca);
 nvueltas = 0;
 }

 public void dar_vuelta () {
 nvueltas++;
 System.out.println(
 "Formula1["+get_marca()+"] , vuelta "+nvueltas);
 }

 protected int nvueltas;
}
```

# Ejemplo de un framework



## Uso del framework

```
class CamionFormula1 extends Formula1 {

 public CamionFormula1 (String marca) {
 super (marca);
 }

 public void dar_vuelta () {
 nvueltas++;
 System.out.println(
 "CamionFormula1[" +get_marca () +
 "], vuelta " +nvueltas);
 }
}
```

# Ejemplo de un framework



## Uso del framework Programa principal

```
class CarreraF1 {
 public static final void main (String[] args) {
 int MAXCOCHES = 3;

 Circuito c = new Circuito("Valencia");
 SafetyCar sc = new SafetyCar ("BMW");

 c.add_safetycar (sc);

 System.out.println("Simulador de carreras");
 }
}
```

# Ejemplo de un framework



## Uso del framework Programa principal

```
...
for (int n = 0; n < MAXCOCHES; n++) {
 int rn = (new Random()).nextInt(100);
 if (rn < 50) { // Formula1
 String marca = "HRT"+n;
 c.add_vehiculo (new Formula1(marca));
 } else {
 String marca = "RENAULT"+n;
 c.add_vehiculo (new CamionFormula1 (marca));
 }
}
c.simular_carrera (7);
} // fin main
} // fin clase
```

# Otro ejemplo de un framework

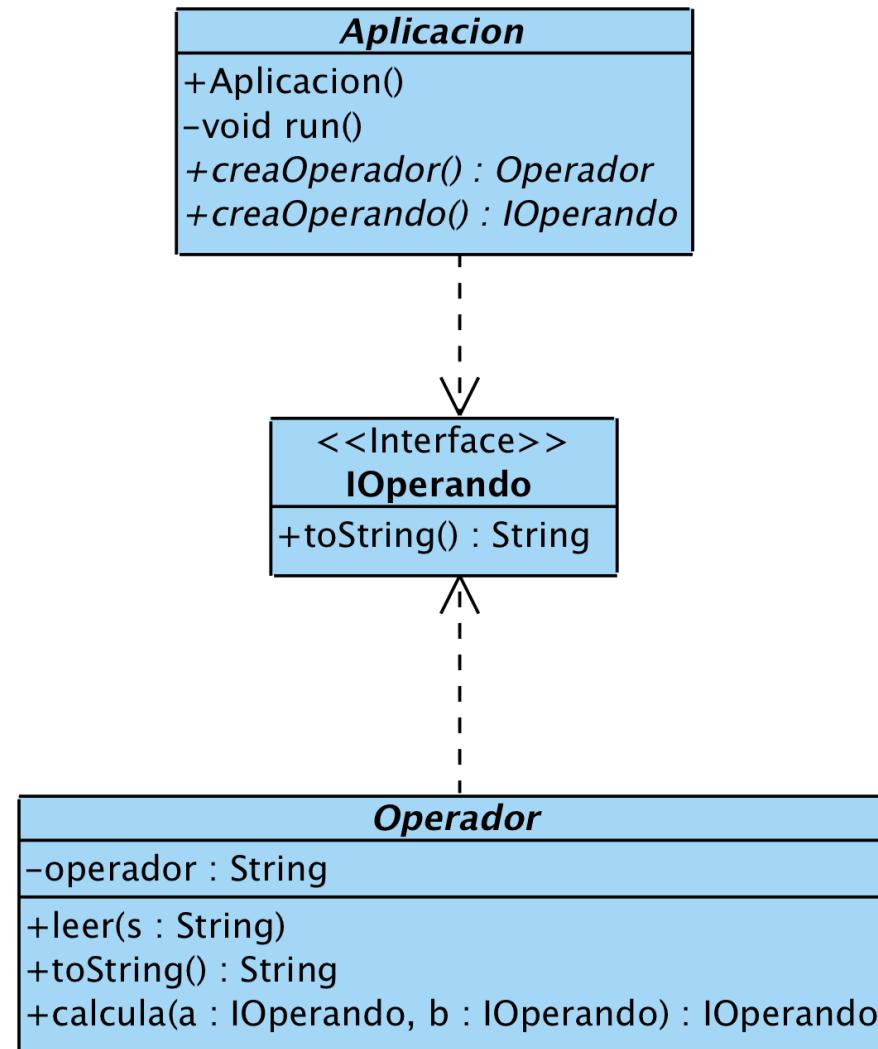


## Operandos y operadores

- Framework propio (de juguete) para la realización de operaciones binarias entre dos operandos
  - No necesariamente números
- El framework nos da hecho:
  - Lectura de operandos
  - Lectura de operador
  - Escritura del resultado
  - Comprobaciones varias
- Cómo usarlo
  - Extendiendo e implementando las clases que se nos indica en la documentación (hipotética)

Aplicacion, IOperando, Operador

# Ejemplo de un framework



# Ejemplo de un framework



```
public interface IOperando {
 String toString();
 void lee(String cadena) throws ExpcionFrameworkOps;
}

public abstract class Operador<TipoOperando extends IOperando> {

 public abstract TipoOperando calcula(TipoOperando a, TipoOperando b) throws
 ExpcionFrameworkOps;
}

public abstract class Aplicacion
 <TipoOperando extends IOperando, TipoOperador extends Operador<TipoOperando>> {
 public abstract TipoOperador creaOperador();
 public abstract TipoOperando creaOperando();

```

# Ejemplo de un framework



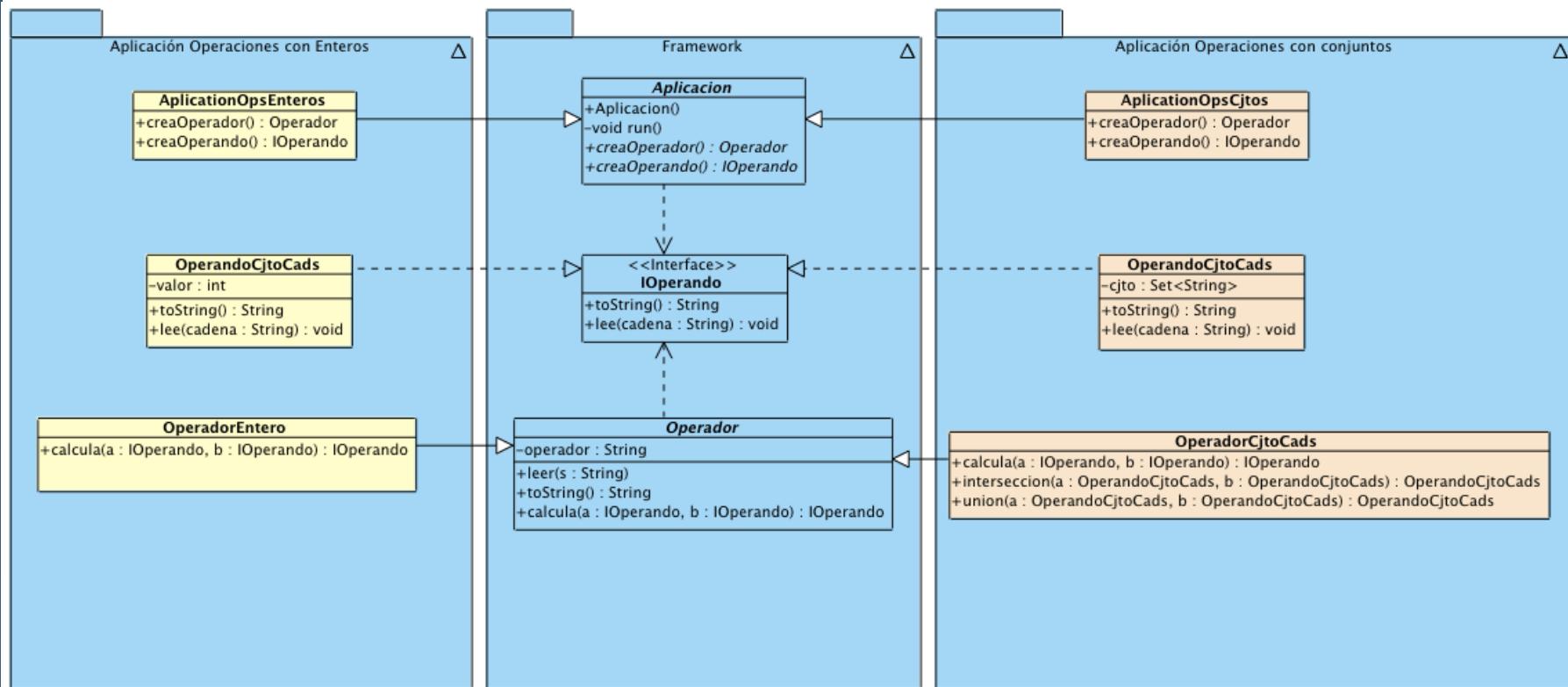
- El framework llama a nuestras clases (las que heredan de Aplicacion, Operador e implementan IOperando)

```
private void run() {
 try {
 Scanner scanner = new Scanner(System.in);
 System.out.println("Introduce el primer operando:");
 TipoOperando a = creaOperando();
 a.lee(scanner.nextLine());
 TipoOperador op = creaOperador();
 System.out.println("Introduce la operación:");
 op.lee(scanner.nextLine());
 System.out.println("Introduce el segundo operador:");
 TipoOperando b = creaOperando();
 b.lee(scanner.nextLine());
 TipoOperando resultado = op.calcula(a, b);
 System.out.println("El resultado de la operación ");
 System.out.print(a.toString()+op.toString()+b.toString());
 + " = " + resultado.toString());
 } catch (ExcepcionFrameworkOps e) {
 System.err.println("Error: " + e.getMessage());
 }
}
```



# Ejemplo de un framework

- Dos aplicaciones hechas con este framework: operaciones con enteros y operaciones con conjuntos



# Ejemplo de un framework



- Creación de una aplicación para operar con enteros:

```
public class ApplicationOpsEnteros extends Aplicacion<OperandoEntero,
OperadorEntero> {
 public static void main(String[] args) {
 new ApplicationOpsEnteros();
 }
 @Override
 public OperadorEntero creaOperador() {
 return new OperadorEntero();
 }

 @Override
 public OperandoEntero creaOperando() {
 return new OperandoEntero();
 }
}
```

# Ejemplo de un framework



- Extendemos Operador

```
public class OperadorEntero extends Operador<OperandoEntero> {

 @Override
 public OperandoEntero calcula(OperandoEntero a, OperandoEntero b)
 throws ExpcionFrameworkOps {
 if (toString().equals("+")) {
 return new OperandoEntero(a.getInt() + b.getInt());
 } else if (toString().equals("-")) {
 return new OperandoEntero(a.getInt() - b.getInt());
 } else {
 throw new ExpcionFrameworkOps("Operando inválido: " +
 toString());
 }
 }
}
```

# Ejemplo de un framework



- Implementamos el interfaz IOperando

```
public class OperandoEntero implements IOperando {
 int valor;
 public OperandoEntero(int valor) {
 this.valor = valor;
 }
 public OperandoEntero() {
 }
 @Override
 public void lee(String cadena) throws ExpcionFrameworkOps {
 try {
 valor = new Integer(cadena).intValue();
 } catch (NumberFormatException e) {
 throw new ExpcionFrameworkOps("Error leyendo el operador
 entero: " + e.getMessage());
 }
 }
 public int getInt() {
 return valor;
 }
 public String toString() {
 return new Integer(valor).toString();
 }
}
```

# Ejemplo de un framework



- Ejemplo de ejecución:

Introduce el primer operando:

10

Introduce la operación:

-

Introduce el segundo operador:

4

El resultado de la operación

10-4=6



# Ejemplo de un framework

- Creación de una aplicación para operar con **conjuntos de cadenas**:

```
public class ApplicationOpsCjtos extends Aplicacion<OperandoCjtoCads,
 OperadorCjtoCads> {
 public static void main(String[] args) {
 new ApplicationOpsCjtos();
 }
 @Override
 public OperadorCjtoCads creaOperador() {
 return new OperadorCjtoCads();
 }
 @Override
 public OperandoCjtoCads creaOperando() {
 return new OperandoCjtoCads();
 }
}
```



# Ejemplo de un framework

## Extendemos Operador

```
public class OperadorCjtoCads extends Operador<OperandoCjtoCads> {
 @Override
 public OperandoCjtoCads calcula(OperandoCjtoCads a, OperandoCjtoCads b)
 throws ExcepcionFrameworkOps {
 if (toString().equals("U")) {
 return union(a,b);
 } else if (toString().equals("I")) {
 return interseccion(a,b);
 } else {
 throw new ExcepcionFrameworkOps("Operando inválido: " + toString());
 }
 }
 private OperandoCjtoCads interseccion(OperandoCjtoCads a,
 OperandoCjtoCads b) {
 Set<String> res = new TreeSet<String>(a.getCjto());
 res.retainAll(b.getCjto());
 return new OperandoCjtoCads(res);
 }
 private OperandoCjtoCads union(OperandoCjtoCads a, OperandoCjtoCads b) {
 Set<String> res = new TreeSet<String>(a.getCjto());
 res.addAll(b.getCjto());
 return new OperandoCjtoCads(res);
 }
}
```



# Ejemplo de un framework

- Implementamos el interfaz IOperando (1/2)

```
public class OperandoCjtoCads implements IOperando {
 Set<String> cjto;

 public final Set<String> getCjto() {
 return cjto;
 }
 public OperandoCjtoCads(Set<String> valores) {
 this.cjto = valores;
 }
 public OperandoCjtoCads() {
 this.cjto = new TreeSet<String>();
 }
```



# Ejemplo de un framework

- Implementamos el interfaz IOperando (2/2)

```
/**
 * Lee la cadena de elementos separados por comas
 */
@Override
public void lee(String cadena) throws ExpcionFrameworkOps {
 try {
 String [] cads = cadena.split(",");
 for (String string : cads) {
 cjto.add(string.trim());
 }
 } catch (NumberFormatException e) {
 throw new ExpcionFrameworkOps("Error leyendo el operador: "
 + e.getMessage());
 }
}
public String toString() {
 return cjto.toString();
}
}
```



# Ejemplo de un framework

- Ejemplo de ejecución:

Introduce el primer operando:

a,b,c

Introduce la operación:

U

Introduce el segundo operador:

b,d

El resultado de la operación

[a, b, c]U[b, d]=[a, b, c, d]

## Librerías y toolkits JDK



- Conjunto de clases que incorporamos a nuestras aplicaciones
- No requieren que implementemos ni heredemos nada
- Son como cajas negras
- El fabricante nos proporciona una API (application program interface)
  - Conjunto de clases y sus métodos
- Podemos considerar librerías a la implementación de referencia del JCF incluido en el JDK:
  - `java.util.ArrayList`
  - `java.util.Stack`
  - `java.util.TreeSet`
  - ...
- En C++ disponemos de librerías similares: STD



*Los toolkits son frameworks*

|     |                              | Java Language           |                    |                       |           |                         |                     |
|-----|------------------------------|-------------------------|--------------------|-----------------------|-----------|-------------------------|---------------------|
| JDK | Toolkits                     | Deployment              |                    | Java Web Start        |           | Java Plug-in            |                     |
|     |                              | IDL                     | JDBC™              | JNDI™                 | RMI       | RMI-IIOP                | Scripting           |
| JRE | User Interface Toolkits      | AWT                     |                    |                       | Swing     |                         | Java 2D             |
|     |                              | Accessibility           | Drag n Drop        | Input Methods         | Image I/O | Print Service           | Sound               |
|     | Integration Libraries        | IDL                     | JDBC™              | JNDI™                 | RMI       | RMI-IIOP                | Scripting           |
|     |                              | Beans                   | Intl Support       |                       | I/O       | JMX                     | JNI                 |
|     | Other Base Libraries         | Networking              | Override Mechanism |                       | Security  | Serialization           | Extension Mechanism |
|     |                              | lang and util           | Collections        | Concurrency Utilities |           | JAR                     | Logging             |
|     | lang and util Base Libraries | Preferences API         | Ref Objects        | Reflection            |           | Regular Expressions     | Management          |
|     |                              | Java Hotspot™ Client VM |                    |                       |           | Java Hotspot™ Server VM |                     |
|     | Java Virtual Machine         | Solaris™                |                    | Linux                 |           | Windows                 |                     |
|     |                              | Platforms               |                    | Other                 |           |                         |                     |

# PROG3

## JDBC



- Java DataBase Conection
- Realmente es un framework que utilizan los fabricantes de RDBMS para conectar Java con sus sistemas de bases de datos
- El JDK proporciona una serie de interfaces y clases abstractas que los fabricantes deben especializar
- Los desarrolladores utilizamos directamente las librerías de los fabricantes, a las que les denominamos *drivers* o conectores. P.ej.
  - MySQL: mysql-connector-java-5.1.18-bin
  - Oracle: ojdbc6.jar
  - SQLServer: sqljdbc4.jar
  - etc....
- Una lista completa de drivers se puede encontrar en  
<http://developers.sun.com/product/jdbc/drivers>



- Estructura básica:
  - 1º crear conexión
  - 2º usar la conexión para consultar / manipular la BBDD
    - Mediante la clase Statement
  - 3º cerrar la conexión

```
Class.forName("com.mysql.jdbc.Driver"); // cargamos driver
// Cadena de conexiónn
String dbURL = "jdbc:mysql://localhost/mibbdd";
// Conectamos
Connection con = DriverManager.getConnection(dbURL, "milogin", "mipassword");
// CONSULTAMOS, INSERTAMOS, BORRAMOS usando con
con.close(); // cerramos conexión al terminar
```



## ■ Inserción / actualización

```
Statement stmt = conexion.createStatement(); // usamos java.sql.Statement y no la de MySQL
String sqlInsercion = "INSERT INTO equipo (nombre, abreviatura) values ('Valencia', 'VAL'), ('Levante', 'LEV')";
int filasInsertadas = stmt.executeUpdate(sqlInsercion);
System.out.println("Se han insertado " + filasInsertadas + " registros");
```

## ■ Borrado

```
Statement stmt = conexion.createStatement();
String sqlBorrado = "delete from equipo where abreviatura = 'ALC' or abreviatura = 'HER'";
int filasBorradas = stmt.executeUpdate(sqlBorrado);
System.out.println("Se han borrado " + filasBorradas + " registros");
```

## ■ Consulta

```
String sqlConsulta = "SELECT abreviatura, nombre from equipo order by abreviatura";
ResultSet rstEquipos = stmt.executeQuery(sqlConsulta);
while(rstEquipos.next()) {
 String abrv = rstEquipos.getString("abreviatura");
 String nombreCompleto = rstEquipos.getString("nombre");
 System.out.println(abrv + "\t" + nombreCompleto);
}
```



- JDBC tiene mecanismos para mejorar el rendimiento
- Por ejemplo: PreparedStatement
- La sentencia SQL es compilada por el SGBD previamente.
- Se crea con una conexión
- La sentencia puede contener variables marcadas con ?.

```
PreparedStatement preparedSQL =
 conexion.prepareStatement("select * from equipo where abreviatura = ?");

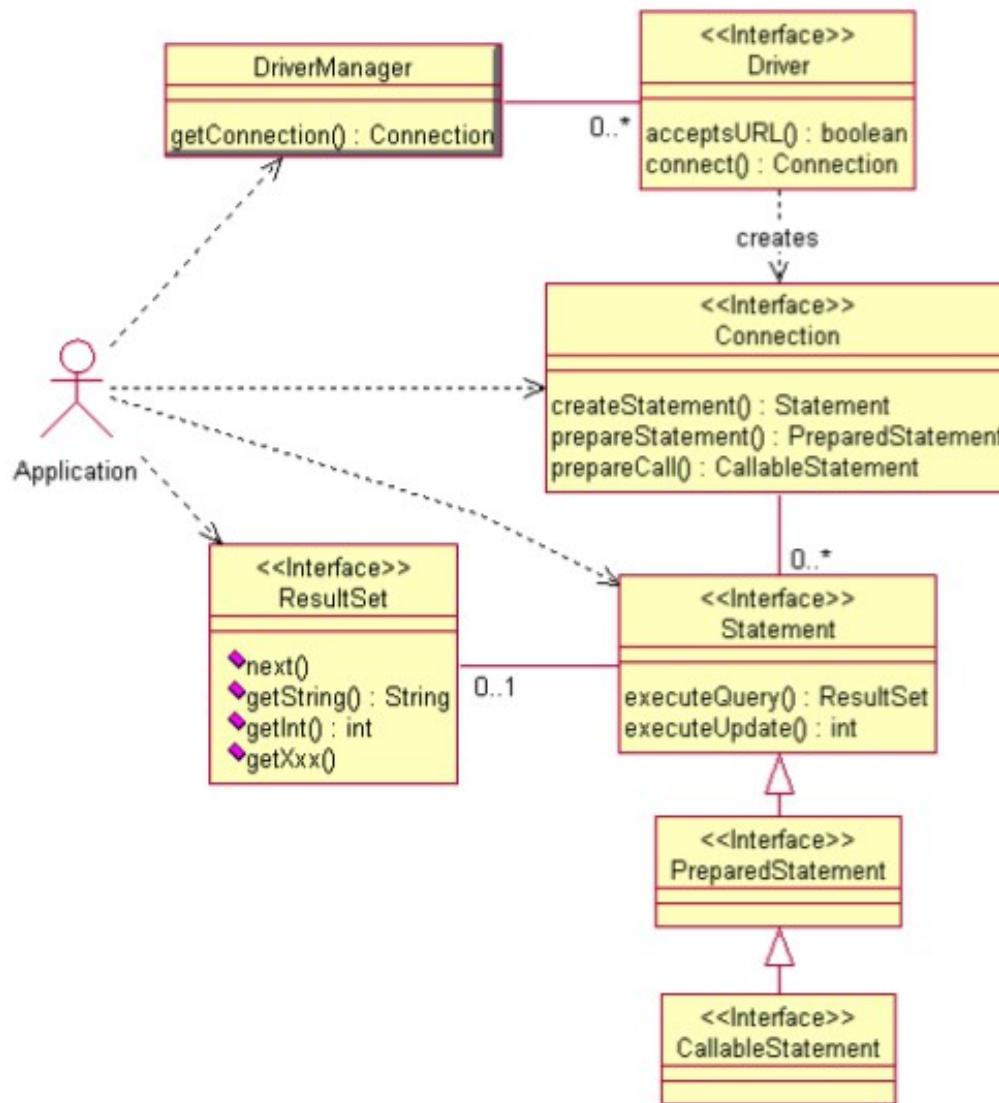
 preparedSQL.setString(1, "ALC");

 rstEquipos = preparedSQL.executeQuery();
 while(rstEquipos.next()) {

 String abrv = rstEquipos.getString("abreviatura");

 String nombreCompleto = rstEquipos.getString("nombre");

 System.out.println(abrv + "\t" + nombreCompleto);
 }
```



## Tratamiento de XML

# Tratamiento de XML



- El JDK incluye dos métodos para analizar XML
  - DOM Parser: lee todo el XML formando el árbol de elementos en memoria
    - Funciona como librería
  - SAX Parser: usado para grandes documentos
    - Funciona como framework. Cada vez que se abre y cierra un elemento se invoca a un *callback* (puntero a una función proporcionada por nosotros)

```
<svg width="1000.0" height="500.0">
 <desc>Esto es una cadena</desc>
 <circle cx="100.0" cy="300.0" fill="red">
</svg>
```

- `svg`, `desc`, `circle` son elementos
- `width`, `height`, `cx`, `cy`, `fill` son atributos

# DOM Parser



```
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
factory.setValidating(true);
factory.setIgnoringElementContentWhitespace(true);
try {
 DocumentBuilder builder = factory.newDocumentBuilder();
 File file = new File("test.xml");
 Document doc = builder.parse(file);

 // TRATAMOS AQUÍ EL DOCUMENTO USANDO LOS MÉTODOS DE LECTURA
 // DE ELEMENTOS DE Document

} catch (ParserConfigurationException e) {
 // TRATAR EXCEPCIÓN
} catch (SAXException e) {
 // TRATAR EXCEPCIÓN
} catch (IOException e) {
 // TRATAR EXCEPCIÓN
}
```

# SAX Parser



- MiElementHandler es una clase nuestra que hereda de DefaultHandler

```
SAXParserFactory factory = SAXParserFactory.newInstance();
factory.setValidating(true);
try {
 SAXParser saxParser = factory.newSAXParser();
 File file = new File("prueba.xml");
 saxParser.parse(file, new EMiElementHandler());
}
catch(ParserConfigurationException e1) {
 //TRATAR EXCEPCION
}
catch(SAXException e1) {
 //TRATAR EXCEPCION
}
catch(IOException e) {
 //TRATAR EXCEPCION
}
```

- Cada vez que se lee el inicio de un elemento XML se invoca al método

```
startElement(String uri, String localName, String qName, Attributes
attributes)
```

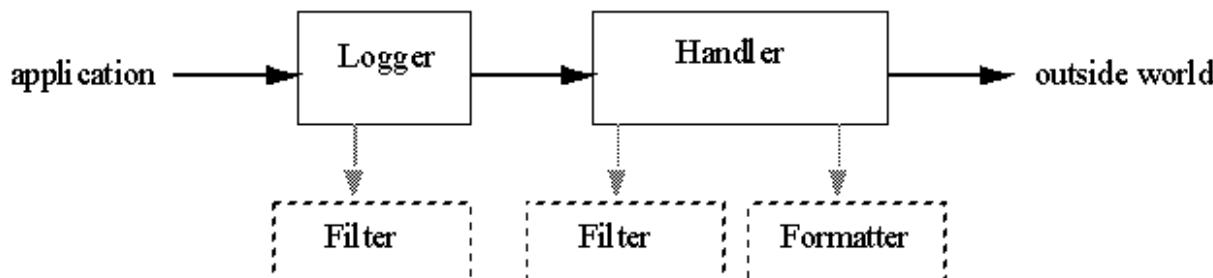
que tendremos definido en nuestra clase MiElementHandler

## Logging



# Logging

- Framework para la emisión eficiente de logs: configurable mediante código o con ficheros
  - Opción no eficiente ni parametrizable
    - .....
    - System.out.println("paso por aquí y x=" + x);
    - .....
    - Además, se mezclan en consola todos los "logs"
  - Opción eficiente y parametrizable
    - logger.info("paso por aquí y x=" + x);
- Actualmente el JDK incorpora el framework Logging





# Logging

- Uso:

```
public class MiClase {
 static final Logger logger = Logger.getLogger(MiClase.class.getName());

 void F() {
 logger.fine("Mensaje de depuración");
 logger.info("Mensaje de monitorización de funcionamiento");
 logger.warning("Mensaje de advertencia");
 logger.severe("Mensaje de error grave");
 }
}
```

- Salida por consola:

```
18-nov-2011 19:24:09 MiClase F
INFO: Mensaje de monitorización de funcionamiento
18-nov-2011 19:24:09 MiClase F
ADVERTENCIA: Mensaje de advertencia
18-nov-2011 19:24:09 MiClase F
GRAVE: Mensaje de error grave
```



# Logging

- Si queremos que sólo nos salgan los logs a partir de un nivel, ponemos en el main, al principio:

```
Logger.getLogger("").getHandlers()[0].setLevel(Level.SEVERE);
System.setProperty("java.util.logging.ConsoleHandler.level", "Level.OFF");
```

- O mejor, usamos un fichero de configuración (*properties* en la terminología Java)

```
java -Djava.util.logging.config.file=./logger.properties -cp
./classes:./lib/* ClassMain
```

- -D pasa una propiedad definida a la máquina virtual
  - que también podríamos cargar así

```
LogManager.getLogManager().readConfiguration(new
FileInputStream("./logger.properties"));
```



# Logging

```
especificacion de detalle de log
nivel de log global
.level = WARNING

manejadores de salida de log
se cargaron un manejador de archivos y
manejador de consola
handlers = java.util.logging.FileHandler,
 java.util.logging.ConsoleHandler

configuración de manejador de archivos
nivel soportado para archivos
java.util.logging.FileHandler.level = ALL
archivo de almacenamiento de las salidas de log
java.util.logging.FileHandler.pattern = ./log/prog3.log
maximo tamaño de archivo en bytes
java.util.logging.FileHandler.limit = 10485760
maximo numero de archivos de logs
java.util.logging.FileHandler.count = 3
clase para formatear salida hacia el archivo de log
java.util.logging.FileHandler.formatter =
 java.util.logging.XMLFormatter
añadir entrada al ultimo archivo (si es falso escribirá al
inicio del archivo cuando la aplicación sea ejecutada)
java.util.logging.FileHandler.append = true
```



# Logging

- Esta configuración genera un fichero prog3.log con este contenido:

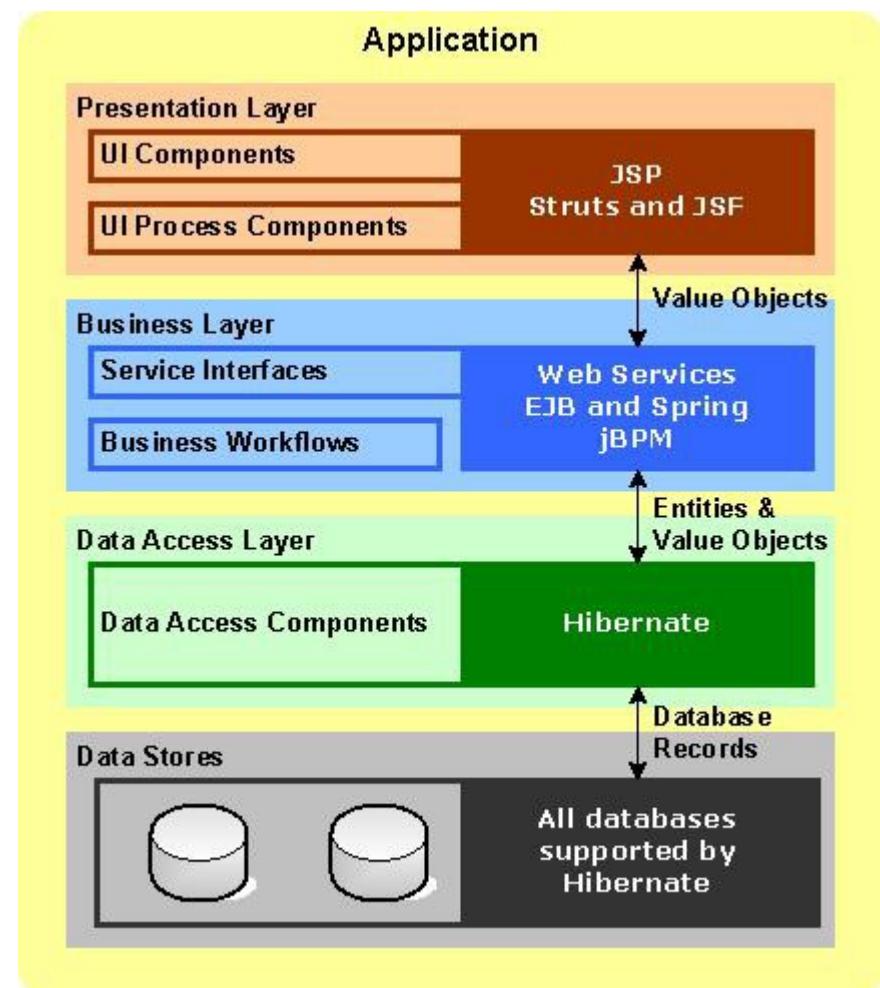
```
<?xml version="1.0" encoding="MacRoman" standalone="no"?>
<!DOCTYPE log SYSTEM "logger.dtd">
<log>
<record>
 <date>2011-11-18T19:37:08</date>
 <millis>1321641428407</millis>
 <sequence>0</sequence>
 <logger>MiClase</logger>
 <level>WARNING</level>
 <class>MiClase</class>
 <method>F</method>
 <thread>10</thread>
 <message>Mensaje de advertencia</message>
</record>
<record>
 <date>2011-11-18T19:37:09</date>
 <millis>1321641429282</millis>
 <sequence>1</sequence>
 <logger>MiClase</logger>
 <level>SEVERE</level>
 <class>MiClase</class>
 <method>F</method>
 <thread>10</thread>
 <message>Mensaje de error grave</message>
</record>
</log>
```

## Hibernate

# Hibernate



- **Mapeo Objeto / Relacional (O/R)**
- A partir de unos ficheros de configuración genera clases que gestionan las operaciones CRUD (create, retrieve, update, delete) que hacen persistentes los objetos en BBDD
- Si usamos JDBC tenemos que escribirlas nosotros manualmente
  - (insert into ...., delete from, .... update, ...., select ...)
  - Con Hibernate nos liberamos de esa tarea





- En el fichero de configuración especificamos parámetros como el nombre de las tablas que asignamos a nuestras clases o cómo se traducen nuestros tipos de datos Java a SQL:

```
<class name="entidades.Temporada" table="temporada" catalog="mibbdd">
<id name="temporadaId" type="java.lang.Integer">
```

- Guardar objetos es tan sencillo como:

```
Session session = HibernateUtil.getSessionFactory().getCurrentSession();
session.beginTransaction();
Temporada temporada = new Temporada(2011, 2012);
session.save(temporada);
session.getTransaction().commit();
```

- Que internamente genera la correspondiente sentencia SQL  
`insert into ...`

## Apache Commons

# Apache Commons



- Conjunto de librerías incluídas en el proyecto Apache
  - <http://commons.apache.org/>
- Especialmente útiles son:
  - CLI: parámetros para la utilidades en línea de comando
  - Collections: añade más tipos abstractos de datos
  - Configuration: ficheros de configuración
  - Email
  - FileUpload: subir ficheros a nuestro servidor
  - Math: para operaciones estadísticas
- Ejemplo: Multimap

```
MultiKeyMap mapa=new;
MultiKey key = new MultiKey(partido.getLocal(), partido.getVisitante());
// para añadir
this.mapa.put(key, partido);
.....
// para recuperar
MultiKey key = new MultiKey(local, visitante);
Partido p = (Partido) mapa.get(key);
```

## Otros frameworks Java

# Otros Frameworks Java



Frameworks de Java en Wikipedia

[http://es.wikipedia.org/wiki/Categor%C3%ADa:Frameworks\\_de\\_Java](http://es.wikipedia.org/wiki/Categor%C3%ADa:Frameworks_de_Java)

- **GWT (Google Web Toolkit)**: generación de aplicaciones ricas web mediante la generación de Javascript a partir de Java
- **Spring**: generación de aplicaciones web empresariales
  - HTML5, REST, AJAX, soporte móviles
  - Mapeo O/R
  - Integración con redes sociales
  - Integración con sistemas basados en mensajería asíncrona
- **Apache Struts**: framework para la generación de aplicaciones web
- **JUnit**: pruebas unitarias
- **JavaFX**: nueva estrategia de Oracle para GUI RIA (rich interface applications)
- **Apache Lucene**: motor de búsqueda para recuperación de información

UD 9

## REFACTORIZACIÓN

*Pedro J. Ponce de León*

Versión 20141203





1. Introducción
  - Qué es la refactorización
  - Un primer ejemplo
2. Principios de refactorización
3. Cuando refactorizar: código sospechoso
4. Pruebas unitarias y funcionales
5. Técnicas de refactorización
  - Refactorizaciones simples
  - Refactorizaciones comunes
  - Refactorización y herencia
6. Bibliografía

# Introducción. Qué es la refactorización

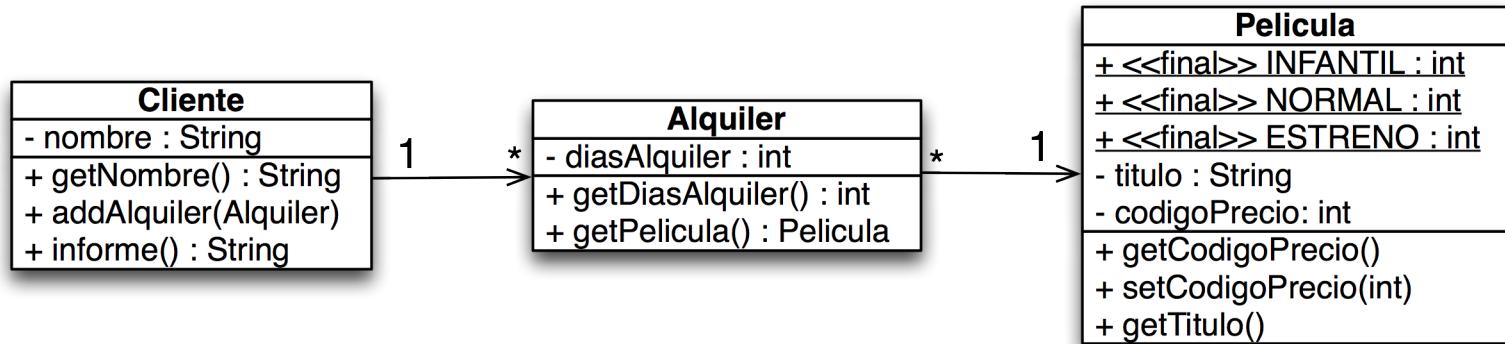


- Proceso de mejora de la estructura interna de un sistema software de forma que su comportamiento externo no varía.
  - Es una forma sistemática de introducir mejoras en el código que minimiza la posibilidad de introducir errores (bugs) en él.
  - Consta básicamente de dos pasos
    - Introducir un cambio simple (refactorización)
    - Probar el sistema tras el cambio introducido
  - Consiste en realizar modificaciones como
    - Añadir un argumento a un método
    - Mover un atributo de una clase a otra
    - Mover código hacia arriba o hacia abajo en una jerarquía de herencia, etc.



# Introducción. Un primer ejemplo

- Tenemos una aplicación de un video-club
  - Calcula e imprime el cargo a realizar a un cliente, a partir de las películas que ha alquilado y el tiempo de alquiler.
  - Hay tres tipos de películas: normales, infantiles y estrenos.
  - La aplicación también bonifica con puntos a los clientes que más alquilan.





# Introducción. Un primer ejemplo

```
public String informe() {
 double totalAmount = 0;
 int frequentRenterPoints = 0;
 Enumeration rentals = _rentals.elements();
 String result = "Rental Record for " + getName() + "\n";

 while (rentals.hasMoreElements()) {
 double thisAmount = 0;
 Alquiler each = (Alquiler) rentals.nextElement();

 //determine amounts for each line
 switch (each.getPelicula().getCodigoPrecio()) {
 case Pelicula.NORMAL:
 thisAmount += 2;
 if (each.getDiasAlquiler() > 2)
 thisAmount += (each.getDiasAlquiler() - 2) * 1.5;
 break;
 case Pelicula.ESTRENO:
 thisAmount += each.getDiasAlquiler() * 3;
 break;
 case Pelicula.INFANTIL:
 thisAmount += 1.5;
 if (each.getDiasAlquiler() > 3)
 thisAmount += (each.getDiasAlquiler() - 3) * 1.5;
 break;
 }
 }
 . . .
```



# Introducción. Un primer ejemplo

```
 . . .
 // add frequent renter points
 frequentRenterPoints++;
 // add bonus for a two day new release rental
 if ((each.getPelicula().getCodigoPrecio() == Pelicula.ESTRENO)
 && each.getDiasAlquiler() > 1)
 frequentRenterPoints++;

 // show figures for this rental
 result += "\t" + each.getPelicula().getTitulo()
 + "\t" + String.valueOf(thisAmount) + "\n";

 totalAmount += thisAmount;
 } // END WHILE

 // add footer lines
 result += "Amount owed is " + String.valueOf(totalAmount) + "\n";
 result += "You earned " + String.valueOf(frequentRenterPoints)
 + " frequent renter points";

 return result;
}
```

- Cliente.informe() hace demasiadas cosas. Muchas de ellas realmente deberían hacerlas otras clases.
- Existe alta probabilidad de introducir errores en el código al modificarlo.



# Introducción. Un primer ejemplo

- Los usuarios solicitan nuevas funcionalidades: imprimir el informe en formato HTML, introducir nuevas formas de clasificar las películas, etc...
- Imposible reutilizar el código de Cliente.informe()
- Solución: copiar/pegar y crear nuevo método Cliente.informeHTML()
- Pero, ¿qué sucedería si las reglas de cargo a clientes cambian?
  - nuevos tipos de pelis, etc



# Introducción. Un primer ejemplo

## Dos 'reglas de oro' de la refactorización:

1. *Cuando necesitamos añadir una nueva funcionalidad a una aplicación, si la estructura de la aplicación no es adecuada para introducir los cambios necesarios, primero refactoriza el código para que el cambio sea fácil de introducir y luego añade la nueva funcionalidad.*
2. *Antes de aplicar técnicas de refactorización, debemos asegurarnos de que disponemos de una batería de pruebas robusta y completa. Estas pruebas deben ser auto-comprobantes (como las pruebas unitarias)*

Refactorizar durante todo el ciclo de vida de una aplicación ahorra tiempo e incrementa la calidad del proyecto.

Objetivo último: Mantener un código conciso y claro, fácil de comprender, modificar y extender (incluso por terceros). Un aspecto clave para ello es la total ausencia de código duplicado.



# Introducción. Un primer ejemplo

- Descomponer y redistribuir el método Cliente.informe()
  - Encapsular la sentencia switch que calcula el cargo aplicable a un ítem en un nuevo método.
  - De esta forma podremos reutilizar esta funcionalidad, independientemente del resto de acciones de Cliente.informe()
- No se trata de un cambio trivial:
  - ¿que variables locales o argumentos de Cliente.informe() se usan en ese trozo de código?
  - ¿Cuáles de ellas simplemente se leen y cuáles son modificadas?



# Introducción. Un primer ejemplo

```
public String informe() {
 ...
 while (rentals.hasMoreElements()) {
 double thisAmount = 0;
 Alquiler each = (Alquiler) rentals.nextElement();

 thisAmount = calculaCargo(each);
 }
}

private double calculaCargo(Alquiler alq) {
 double cargo=0;
 switch (alq.getPelicula().getCodigoPrecio()) {
 case Pelicula.NORMAL:
 cargo += 2;
 if (alq.getDiasAlquiler() > 2)
 cargo += (alq.getDiasAlquiler() - 2) * 1.5;
 break;
 case Pelicula.ESTRENO:
 cargo += alq.getDiasAlquiler() * 3;
 break;
 case Pelicula.INFANTIL:
 cargo += 1.5;
 if (alq.getDiasAlquiler() > 3)
 cargo += (alq.getDiasAlquiler() - 3) * 1.5;
 break;
 }
 return cargo;
}
```



# Principios de refactorización

## Motivos para refactorizar

- Mejorar el diseño del software
- Hacer que el código sea más fácil de entender
- Hacer que sea más sencillo encontrar fallos
- Permite programar más rápidamente



# Principios de refactorización

## ¿Cuando refactorizar?

- Metáfora de los dos sombreros

- ◆ Un programador tiene dos sombreros:
  - ◆ uno para modificar código (refactorizar),
  - ◆ otro para añadir nuevas funcionalidades
- ◆ Cuando trabaja lleva puesto uno (y sólo uno) de los dos sombreros.
- ◆ Cuando añade código nuevo, NO modifica el existente. Si está arreglando el existente, NO añade funcionalidades nuevas.



# Principios de refactorización

## ¿Cuando refactorizar?

Arregla el código con frecuencia – >Refactoriza sistemáticamente.

- Refactoriza al añadir un método/función
- Refactoriza cuando necesites arreglar un fallo
- Refactoriza al revisar código
- Refactoriza cuando 'algo huele mal'



# Principios de refactorización

## Problemas con la refactorización

- *Capa de persistencia:*

Acoplamiento con bases de datos

- *Cambios de interfaz*

- Refactorizar implica a menudo cambios en la interfaz de las clases
- Interfaces publicados (vs. públicos): aquellos utilizados por código cliente al que no tenemos acceso. Se hace necesario mantener la antigua interfaz junto a la nueva. A menudo esto se consigue haciendo que los métodos de la antigua interfaz deleguen en los de la nueva.
- En Java, podemos usar la anotación @deprecated
- Moraleja: no publique interfaces de forma prematura.



# Principios de refactorización

## Cuando no refactorizar

- Cuando el código original es tan 'malo' (por diseño, o múltiples fallos) que merece más la pena reescribirlo desde el principio.
- ¡Cuando se están a punto de cumplir los plazos!

# Cuando refactorizar: **código sospechoso**



A menudo encontramos *código sospechoso*: algo nos dice que ese código podría ser mejor. A menudo a esto se le llama código con mal olor (*bad code smells*, en inglés).

Algunos ejemplos:

Código duplicado: líneas de código exactamente iguales o muy parecidas en varios sitios. Se debe unificar en un sólo sitio. Es el 'mal olor' más común y se debe evitar a toda costa.

Métodos muy largos: Cuanto más largo es el código, más difícil de entender y mantener. Un método muy largo normalmente está realizando tareas que deberían ser responsabilidad de otros. Se deben identificar éstas y descomponer el método en otros más pequeños.



# Cuando refactorizar: código sospechoso

## Clases muy grandes.

Clases con

- demasiados métodos,
- demasiados atributos
- o incluso demasiadas instancias.

La clase está asumiendo, por lo general, demasiadas responsabilidades.

Se debe identificar si realmente todas esas cosas tienen algo que ver entre sí y si no es así, hacer clases más pequeñas, de forma que cada una trate con un conjunto pequeño de responsabilidades bien delimitadas (por ejemplo, ocuparse de la conexión con una base de datos, o manejar cierto tipo de información específica, como fechas, DNI, etc.)



# Cuando refactorizar: código sospechoso

## Métodos con demasiados parámetros.

Los métodos de una clase suelen disponer de la mayor parte de la información que necesitan en su propia clase o clases base. Tener demasiados parámetros puede estar indicando un problema de encapsulación de datos o la necesidad de crear una clase de objetos a partir de varios de esos parámetros y pasar ese objeto como argumento en vez de todos los parámetros. Especialmente si esos parámetros suelen tener que ver unos con otros y suelen ir juntos siempre.

## Sentencias 'switch':

Normalmente un switch se tiene que repetir en el código en varios sitios, aunque en cada sitio sea para hacer cosas distintas. A menudo la solución es usar polimorfismo para evitar tener que repetir el 'switch' en varios sitios, o incluso evitarlo completamente.



# Pruebas unitarias y funcionales

En la práctica, la mayor parte del tiempo de desarrollo se dedica a depurar código.

De ese tiempo, la mayor parte se invierte en encontrar los fallos. El tiempo dedicado a arreglarlo es normalmente ínfimo, en comparación.

Los **test o pruebas unitarios** son una forma de probar el correcto funcionamiento de un módulo de código. Esto sirve para asegurar que cada uno de los módulos funcione correctamente por separado.

La idea es escribir casos de prueba para cada función no trivial o método en el módulo o paquete, de forma que cada caso sea independiente del resto.

Las **pruebas funcionales** tratan a un componente software como una caja negra. Verifican una aplicación comprobando que su funcionalidad se ajusta a los requerimientos o a los documentos de diseño. Se trata a la aplicación o componente software como un todo.



# Pruebas unitarias y funcionales

Para que una prueba unitaria sea buena se deben cumplir los siguientes requisitos:

- Automatizable: no debería requerirse una intervención manual.
- Completas: deben cubrir la mayor cantidad de código.
- Repetibles o Reutilizables: no se deben crear pruebas que sólo puedan ser ejecutadas una sola vez.
- Independientes: la ejecución de una prueba no debe afectar a la ejecución de otra.
- Profesionales: las pruebas deben ser consideradas igual que el código, con la misma profesionalidad, documentación, etc.



# Pruebas unitarias y funcionales

## ***Suites de pruebas unitarias***

Los conjuntos o 'suites' de pruebas unitarias son un potente detector de fallos que reduce drásticamente el tiempo necesario para encontrarlos.

- Son una precondición necesaria para utilizar la refactorización.
- Se asocia una suite de pruebas a cada clase.
- Se deben ejecutar tras cada pequeño cambio o refactorización en el código.



# Pruebas unitarias y funcionales

## Aserciones

Típicamente cada método es probado utilizando **aserciones**, que verifican que el resultado obtenido y el esperado son iguales.

Es especialmente relevante probar

- Las condiciones o valores límite con los que un método debe ejecutarse.
- Las condiciones bajo las cuales un método genera excepciones.



# Pruebas unitarias y funcionales

## ¿Cuándo escribir pruebas unitarias?

Tradicionalmente, se piensa que esto se hace tras haber incluído el nuevo código a probar.

Sin embargo, resulta mucho más útil escribirlas ANTES de escribir el código a probar. A menudo, esto sirve para tener más claro el comportamiento de un método.



# Pruebas unitarias y funcionales

En **Java**, la herramienta **JUnit** (código abierto) es la más usada para pruebas unitarias:

<http://junit.sourceforge.net/>

En **C++**, existen varias librerías similares a Junit en Java:

Cxxtest : <http://cxxtest.tigris.org/> (GNU LGPL)

CppUnit : <https://launchpad.net/cppunit2> (GNU LGPL)

En **C#**,

Nunit : <http://www.nunit.org/> (licencia tipo BSD)

CsUnit : <http://www.csunit.org/> (licencia zLib)



# Técnicas de refactorización

## Refactorizaciones simples

- Añadir un parámetro
- Quitar un parámetro
- Cambiar el nombre de un método



# Técnicas de refactorización

## Añadir un parámetro

Motivo: Un método necesita más información al ser invocado

Solución: Añadir como parámetro un objeto que proporcione dicha información

### Ejemplo

Cliente.getContacto() → Cliente.getContacto(Fecha f)

### Observaciones

Evitar listas de argumentos demasiado largas.



# Técnicas de refactorización

## Quitar un parámetro

Motivo: Un parámetro ya no es usado en el cuerpo de un método

Solución: Eliminarlo

### Ejemplo

```
Cliente.getContacto(Fecha f) → Cliente.getContacto()
```

### Observaciones

Si el método está sobrescrito, puede que otras implementaciones del método sí lo usen. En ese caso no quitarlo. Considerar sobrecargar el método sin ese parámetro.



# Técnicas de refactorización

## Cambiar el nombre de un método

Motivo: El nombre de un método no indica su propósito

Solución: Cambiarlo

### Ejemplo

```
Cliente.getLimCrdFact() →
Cliente.getLimiteCreditoFactura()
```

### Observaciones

- Comprobar si el método está implementado en clases base o derivadas.
- Si es parte de una interface publicada, crear un nuevo método con el nuevo nombre y el cuerpo del método. Anotar la versión anterior como @deprecated y hacer que invoque a la nueva.
- Modificar todas las llamadas a este método con el nuevo nombre.



# Técnicas de refactorización

## Refactorizaciones comunes

- Mover un atributo
- Mover un método
- Extraer una clase
- Extraer un método
- Cambiar condicionales por polimorfismo
- Cambiar código de error por excepciones



# Técnicas de refactorización

## Mover un atributo

Motivo: Un atributo es (o debe ser) usado por otra clase más que en la clase donde se define.

Solución: Crear el atributo en la clase destino

### Observaciones

- Si el atributo es público, encapsularlo primero.
- Reemplazar las referencias directas al atributo por llamadas al getter/setter correspondiente.



# Técnicas de refactorización

## Mover un atributo

### Ejemplo

```
class Cuenta {
 private TipoCuenta tipo;
 private double tipoInteres;

 double calculaInteres(double saldo, int dias) {
 return tipoInteres * saldo * dias /365;
 }

class TipoCuenta {
 private double tipoInteres;

 void setInteres(double d) {...}
 double getInteres() {...}

 double calculaInteres(double saldo, int dias) {
 return tipo.getInteres() * saldo * dias /365;
 }
}
```



# Técnicas de refactorización

## Mover un método

Motivo: Un método es usado más en otro lugar que en la clase actual

Solución: Crear un nuevo método allí donde más se use.

### Ejemplo

```
Cliente.getLimiteCreditoFactura()
Cuenta.getLimiteCreditoFactura(Cliente c)
// arg. opcional, sólo en caso de necesitar info. de
Cliente
```

### Observaciones

- Es una de las refactorizaciones más comunes
- Hacer que el antiguo método invoque al nuevo o bien eliminarlo.
- Comprobar si el método está definido en la jerarquía de clases actual. En ese caso puede no ser posible moverlo.



# Técnicas de refactorización

## Extraer una clase

Motivo: Una clase hace el trabajo que en realidad deberían hacer entre dos.

Solución: Crear una nueva clase y mover los métodos y atributos relevantes de la clase original a la nueva.

### Observaciones

- La nueva clase debe asumir un conjunto de responsabilidades bien definido.
- Implica crear una relación entre la nueva clase y la antigua.
- Implica 'Mover atributo' y 'Mover método'
- Se debe considerar si la nueva clase ha de ser pública o no.



# Técnicas de refactorización

## Extraer una clase

### Ejemplo

Cliente	
- nombre : String	
- prefijoTelefonoTrabajo : String	
- telefonoTrabajo : String	
- prefijoTelefonoCasa : String	
- telefonoCasa : String	
+ getNombre() : String	
+ getTelefonoTrabajo() : String	
+ getTelefonoCasa() : String	

Cliente	
- nombre : String	
+ getNombre() : String	
+ getTelefonoTrabajo() : String	
+ getTelefonoCasa() : String	

Telefono	
- prefijo : String	
- numero : String	
+ getPrefijo() : String	
+ getNumeroSinPrefijo() : String	
+ getNumero() : String	

telefonotrabajo  
1

telefonocasa  
1



# Técnicas de refactorización

## Extraer un método

Motivo: Existe un fragmento de código (quizás duplicado) que se puede agrupar en una unidad lógica.

Solución: Convertir el fragmento en un método cuyo nombre indique su propósito e invocarlo desde donde estaba el fragmento.

Ejemplo: (ejemplo de la introducción de la UD)

### Observaciones

- Se realiza a menudo cuando existen métodos muy largos.
- Las variables locales que sólo se leen en el fragmento se convertirán en argumentos/var. locales del nuevo método
- Si algunas variables locales son modificadas en el fragmento, son candidatas a ser valor de retorno del método
- Considerar si se debe publicar el nuevo método.



# Técnicas de refactorización

## Cambiar condicionales por polimorfismo

Motivo: Una estructura condicional elige entre diferentes comportamientos en función del tipo de un objeto.

Solución: Convertir la estructura condicional en un método ('Extraer método'). Convertir cada opción condicional en una versión del método sobrescrito en la clase derivada correspondiente. Hacer el método en la clase base abstracto.

### Observaciones

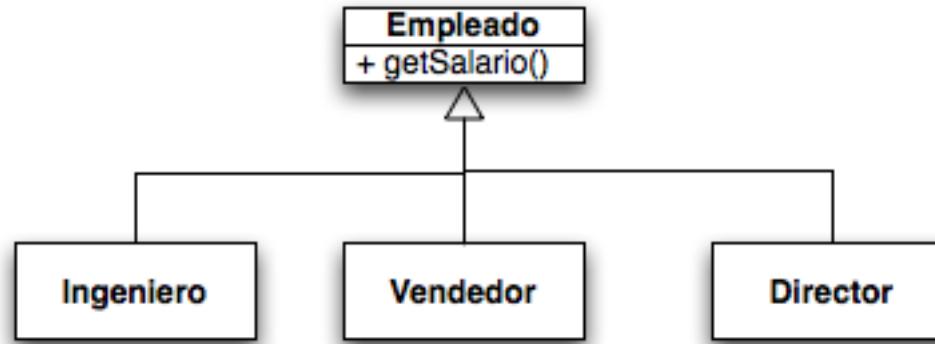
- Con condicionales el código cliente necesita conocer las clases derivadas
- Con polimorfismo el código cliente sólo necesita conocer a la clase base
- Esto permite añadir nuevos tipos sin modificar el código cliente



# Técnicas de refactorización

## Cambiar condicionales por polimorfismo

Ejemplo:



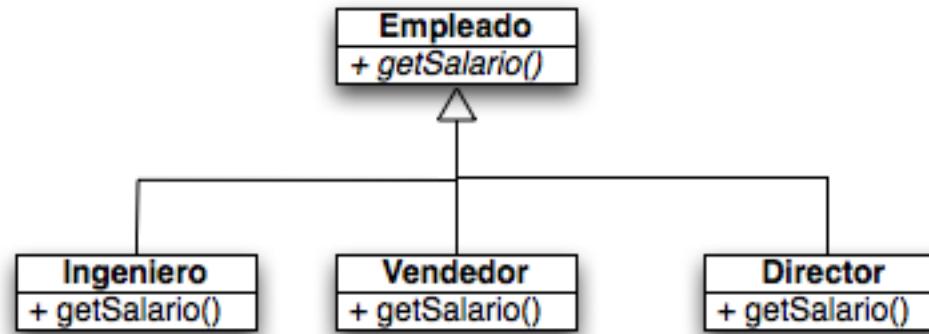
```
double getSalario() {
 switch(tipoEmpleado()) {
 case INGENIERO: return salarioBase + productividad; break;
 case VENDEDOR: return salarioBase + ventas*comision; break;
 case DIRECTOR: return salarioBase + bonificacion+ dietas; break;
 default : throw new RuntimeException("Tipo de empleado incorrecto");
 }
}
```



# Técnicas de refactorización

## Cambiar condicionales por polimorfismo

Ejemplo:



```
class Empleado {
 abstract double getSalario(); ...
}

class Ingeniero {
 @Override double getSalario() { return salarioBase + productividad; }
 ...
}
class Vendedor {
 @Override double getSalario() { return salarioBase + ventas*comision; }
 ...
}
class Director {
 @Override double getSalario() {return salarioBase+bonificacion+dietas;}
 ...
}
```



# Técnicas de refactorización

## Cambiar código de error por excepciones

Motivo: Um método devuelve un valor especial para indicar un error

Solución: Lanzar una excepción en lugar de devolver ese valor.

### Observaciones

- Decidir si se debe lanzar una excepción verificada o no verificada.
- El código cliente debe manejar la excepción.
- ¡Ojo! cambiar la cláusula 'throws' de un método público (excepciones verificadas) en Java equivale a cambiar la interfaz de la clase donde se define.



# Técnicas de refactorización

## Cambiar código de error por excepciones

### Ejemplo

```
int sacarDinero(int cantidad) {
 if (cantidad > saldo) return -1;
 else { saldo -= cantidad; return 0; }
}

int sacarDinero(int cantidad) throws ExcepcionSaldoInsuficiente {
 if (cantidad > saldo) throw new ExcepcionSaldoInsuficiente();
 else { saldo -= cantidad; return 0; }
}
```



- **Refactorización y herencia**

- Generalizar un método
- Especializar un método
- Colapsar una jerarquía
- Extraer una clase derivada
- Extraer una clase base (o interfaz)
- Convertir herencia en composición



# Técnicas de refactorización

## Generalizar un método

Motivo: Existen dos o más métodos con idéntico comportamiento (código duplicado) en las clases derivadas.

Solución: Unificarlos en un único método en la clase base

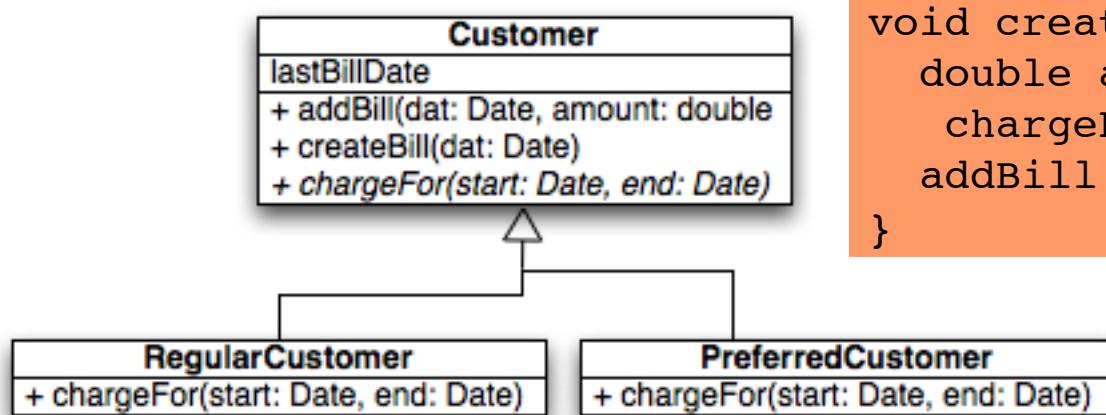
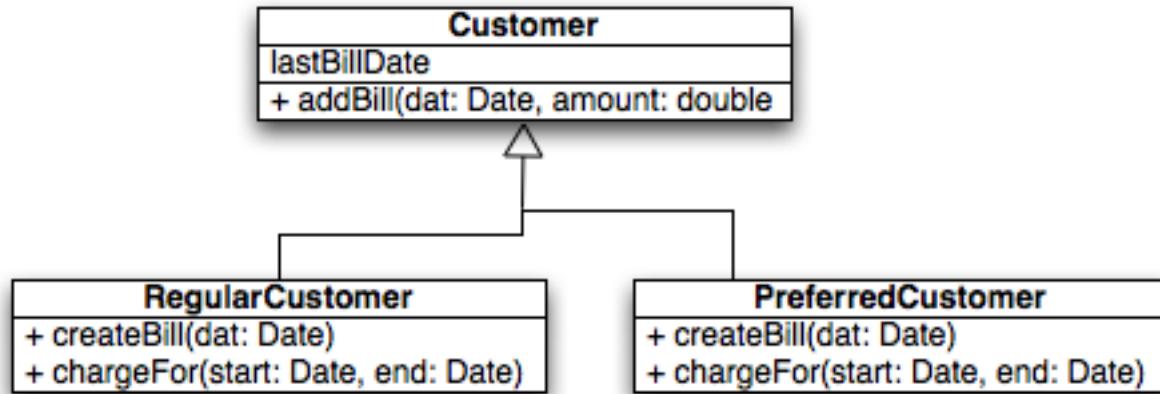
### Observaciones

- Prestar atención a diferencias en el comportamiento (cuerpo de los métodos)
- El código cliente debe manejar la excepción.
- Caso particular: un método en clase derivada sobrescribe uno de la clase base pero hace esencialmente lo mismo.
- Si el método invoca a métodos declarados en las clases derivadas:
  - Primero generalizar los métodos invocados si es posible, o
  - Declarar dichos métodos como abstractos en la clase base



# Técnicas de refactorización

## Generalizar un método



```
void createBill(Date dat) {
 double amnt =
 chargeFor(lastBillDate, dat);
 addBill(date, amnt);
}
```



# Técnicas de refactorización

## Especializar un método

Motivo: Un comportamiento de la clase base sólo es relevante para algunas clases derivadas.

Solución: Mover el método que lo implementa a las clases derivadas.

### Ejemplo

`Empleado.getCuotaVentas() → Vendedor.getCuotaVentas()`

### Observaciones

- Usado en 'Extraer clase derivada'
- Puede implicar cambiar la visibilidad de ciertos atributos a protegida (o declarar getter/setter públicos/protegidos) en la base.
- Si supone un cambio de interfaz, revisar el código cliente. Otra posibilidad es dejar el método como abstracto en la base



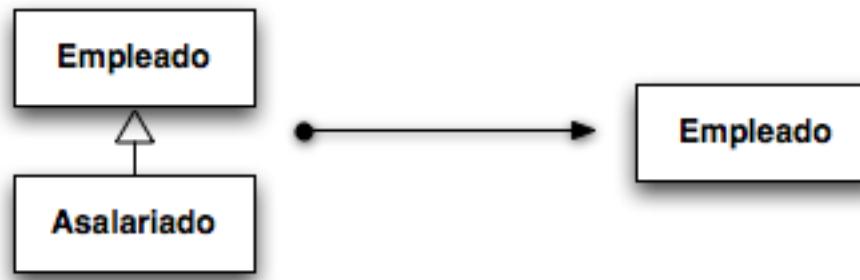
# Técnicas de refactorización

## Colapsar una jerarquía

Motivo: Apenas hay diferencias entre una clase base y una de sus derivadas.

Solución: Juntarlas en una sola clase.

Ejemplo:



Observaciones

- Decidir qué clase eliminar (base o derivada)
- El código cliente de la clase eliminada (si es pública) deberá ser modificado.



# Técnicas de refactorización

## Extraer una subclase

Motivo: Algunas características de una clase son usadas sólo por un grupo de instancias.

Solución: Crear una clase derivada para ese conjunto de características.

### Observaciones

- Es a menudo consecuencia de usar un diseño top-down
- La presencia de atributos como 'tipoCliente' señala la necesidad de este tipo de refactorización.
- El código cliente debe ser revisado, en particular, la construcción de objetos de clase base.
- Relacionado con 'especializar método', 'reemplazar condicional con polimorfismo'



# Técnicas de refactorización

## Extraer una subclase

Ejemplo:

Medico
- cargaPacientes : int
- cargaMaxima : int
+ Medico(nombreAp :string, cargaMax: int)
+ addPaciente(Paciente p) : void
+ quitarPaciente(const Paciente&) : bool
+ getCargaPacientes() : int
+ getCargaMaxima() : int
+ puedeAtenderMasPacientes() : bool

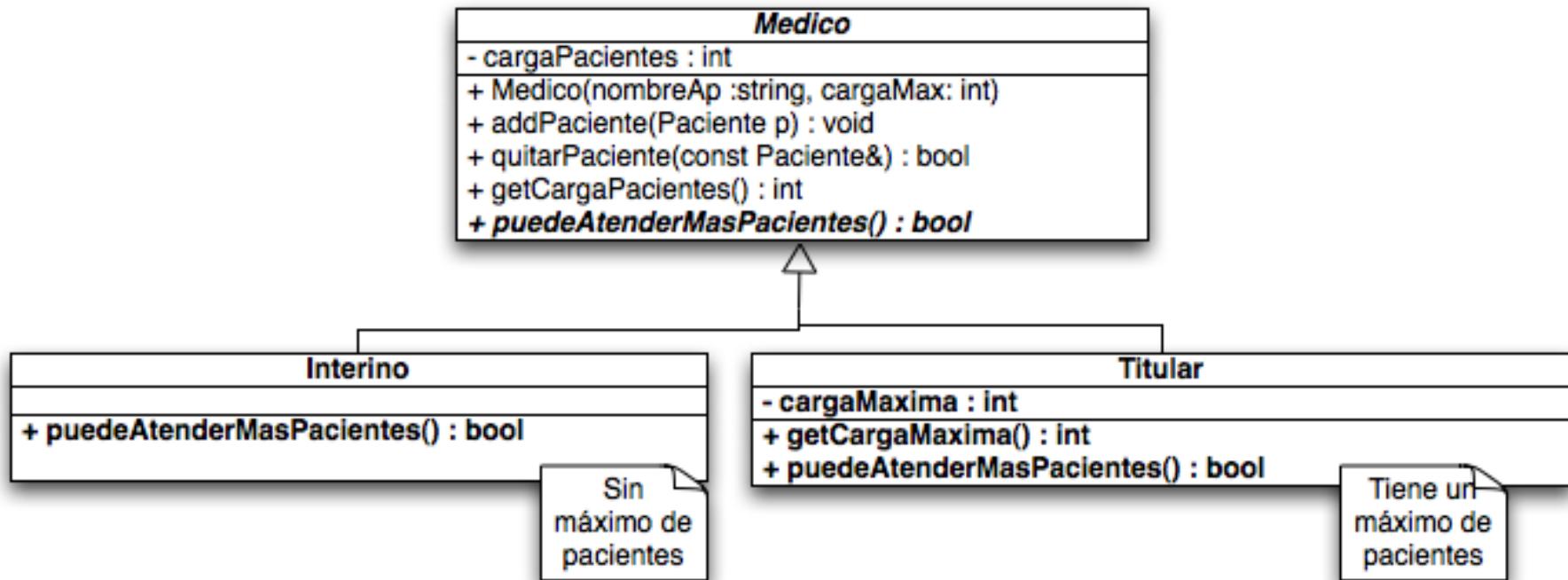
La política del hospital dice que sólo los médicos titulares pueden tener una carga máxima de pacientes. Los interinos pueden atender un número indefinido de pacientes.



# Técnicas de refactorización

## Extraer una subclase

Ejemplo:





# Técnicas de refactorización

## Extraer una superclase

Motivo: Hay dos (o más) clases con características similares.

Solución: Crear una clase base para ellas y mover el comportamiento común a la base.

### Observaciones

- A menudo consecuencia de usar un diseño bottom-up.
- Casi siempre implica eliminar código duplicado.
- La nueva clase base suele ser abstracta o incluso un interfaz.
- Hay que prestar especial atención a qué métodos deben subir a la base ('generalizar método')
- Las referencias en el código cliente pueden tener que ser actualizadas de referencia a derivada a referencia a base.



# Técnicas de refactorización

## Extraer una superclase

Ejemplo:

Círculo
- origen: Coordenada
- radio : float
+ getCentro() : Coordenada
+ getRadio() : float
+ setRadio(pradio : float) : void
+ pintar(ps : PrintStream) : void
+ mover(porigen : Coordenada) : Coordenada

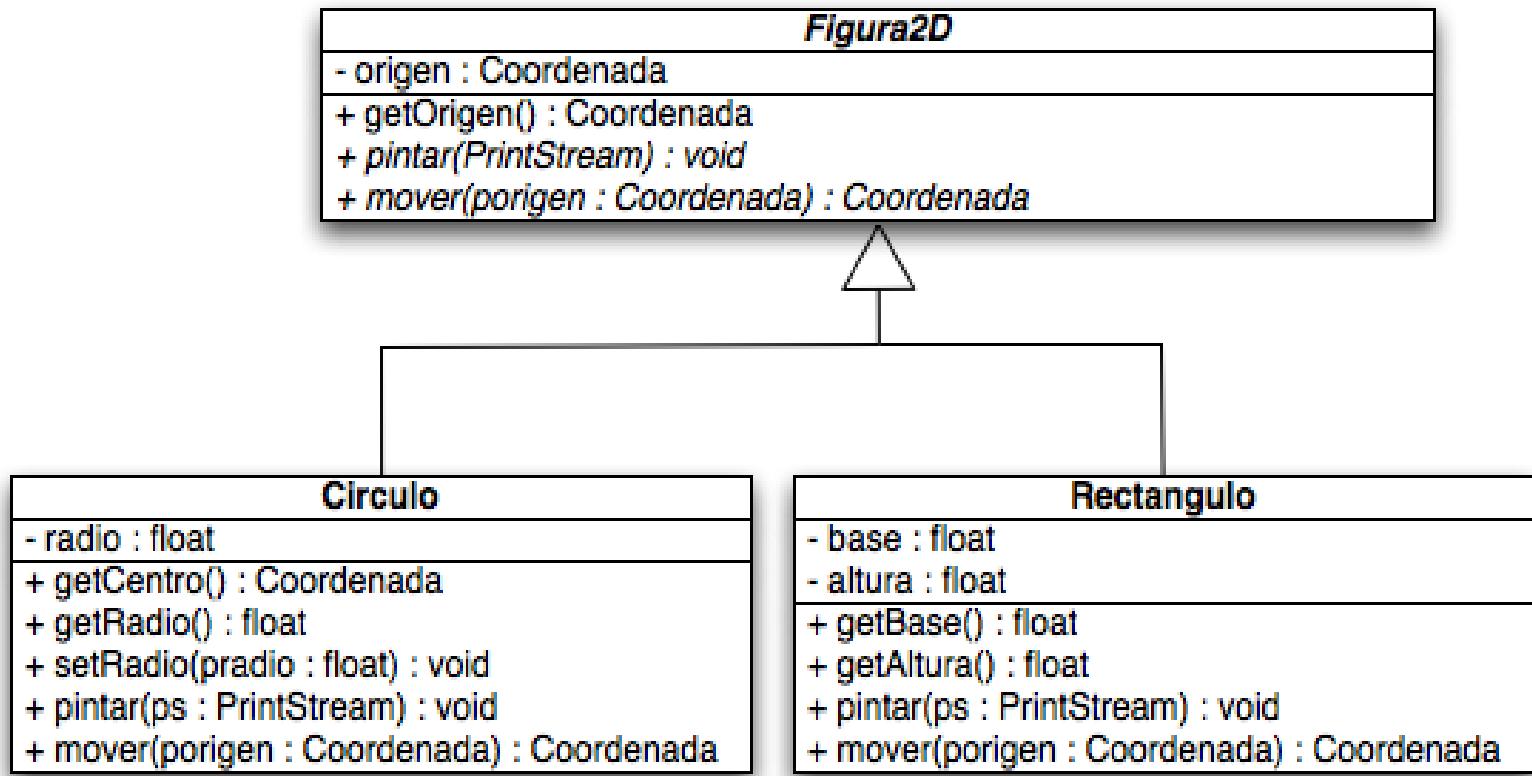
Rectángulo
- origen : Coordenada
- base : float
- altura : float
+ getOrigen() : Coordenada
+ getBase() : float
+ getAltura() : float
+ pintar(ps : PrintStream) : void
+ mover(porigen : Coordenada) : Coordenada



# Técnicas de refactorización

## Extraer una superclase

Ejemplo:





# Técnicas de refactorización

## Convertir herencia en composición

Motivo: El principio de sustitución no se cumple: una clase derivada usa sólo parte de la interfaz de la base o no desea heredar también los atributos.

Solución: Crear un campo para la clase base en la derivada, ajustar los métodos involucrados para delegar en la clase base y eliminar la herencia.

### Observaciones

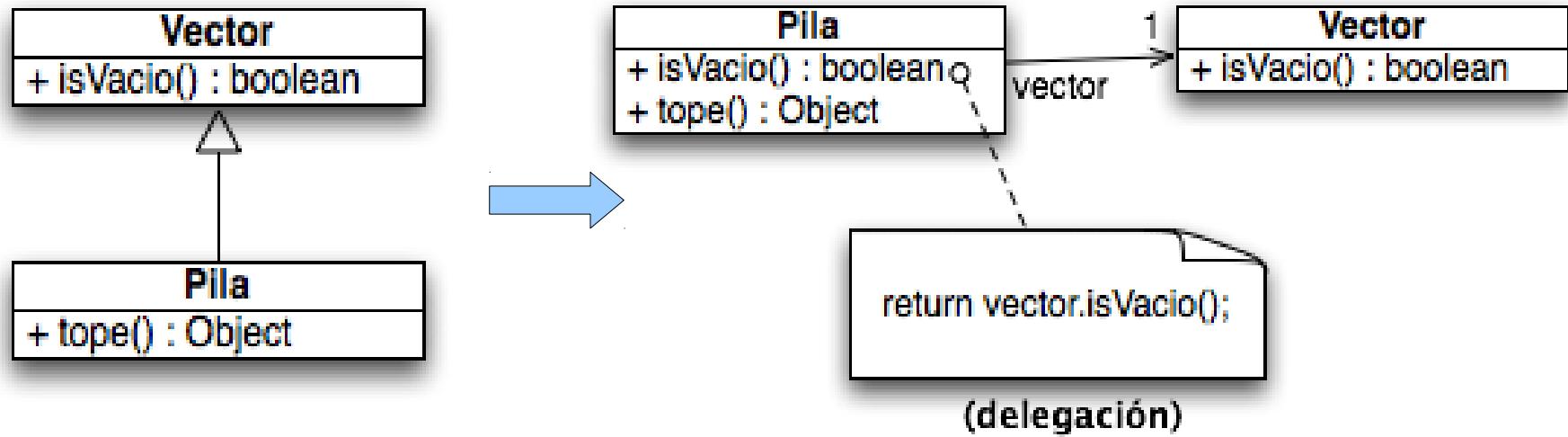
- Los métodos de la clase base usados en la derivada se implementan en la derivada y se convierten en una invocación al método base (delegación)



# Técnicas de refactorización

## Convertir herencia en composición

Ejemplo:





# Técnicas de refactorización

## Otras refactorizaciones

Existen otras muchas refactorizaciones que no se presentan aquí:

- Reemplazar valores por objetos
- Reemplazar paso por valor por paso por referencia
- Reemplazar un array por un objeto
- Cambiar una asociación unidireccional en una bidireccional o viceversa
- Eliminar setters
- Convertir un diseño procedural en orientado a objetos
- Extraer una jerarquía
- Separar la capa de dominio de la capa de presentación
- etc, etc,...



# Técnicas de refactorización

## Conclusiones

- Refactorizar es una forma sistemática y segura de realizar cambios en una aplicación con el fin de hacerla más fácil de comprender, modificar y extender.
- Algunas de las refactorizaciones vistas aquí se pueden hacer de forma automática en algunos entornos de desarrollo (p. ej., Eclipse para Java).
- Lo importante es saber qué y cuando refactorizar.
- La refactorización ha sido identificado como una de las más importantes innovaciones en el campo del software:

<http://www.dwheeler.com/innovation/innovation.html>



# Bibliografía

En esta unidad docente no se ha hecho especial hincapié en los pasos a seguir para realizar las refactorizaciones de forma controlada. Una lectura imprescindible para conocer estos detalles es el libro de Martin Fowler:

Martin Fowler. ***Refactoring. Improving the Design of Existing Code***  
Addison Wesley, 2007

<http://martinfowler.com/refactoring/>

UD 10

## PRINCIPIOS DE DISEÑO ORIENTADO A OBJETOS

*Pedro J. Ponce de León*

Versión 20141210





## 1. Introducción

- Diseño orientado a objetos
- Principios generales de diseño
  - Repasso: Acoplamiento y cohesión

## 2. Principios de diseño orientado a objetos

- Principio abierto-cerrado
- Principio de sustitución (Liskov)
- Principio de segregación de interfaz
- Principio de inversión de dependencia
- Principio de responsabilidad única

## 3. Un vistazo a los patrones de diseño

## 4. Bibliografía

# Introducción. Diseño orientado a objetos



- Diseño de software
  - DISEÑO: En el contexto del software, el diseño es un proceso de solución de problemas cuyo objetivo es encontrar y describir una forma de implementar los requerimientos funcionales de un sistema, respetando las restricciones impuestas por los principios de diseño, la plataforma y los requisitos del proceso tales como el presupuesto y los plazos.

# Introducción. Diseño orientado a objetos



## ■ **Diseño orientado a objetos**

- Forma de diseñar donde se aplican, además de los principios de diseño generales del software, los principios específicos de diseño de software orientado a objetos, utilizando una plataforma de desarrollo (lenguaje, frameworks, librerías, etc.) orientada a objetos.
- Los principios de diseño (generales u orientados a objetos) tienen como objetivo mejorar la calidad, tanto intrínseca como extrínseca del software (ver tema 1).
- Damos por hecho: encapsulación, polimorfismo (enlace dinámico), herencia.



## REPASO: Acoplamiento y cohesión (de UD3)

- **Acoplamiento:** relación entre componentes software
  - Interesa bajo acoplamiento. Éste se consigue moviendo las tareas a quién ya tiene habilidad para realizarlas.
- **Cohesión:** grado en que las responsabilidades de un solo componente forman una unidad significativa.
  - Interesa alta cohesión. Ésta se consigue asociando a un solo componente responsabilidades que están relacionadas en cierta manera, p. ej., acceso a los mismos datos.



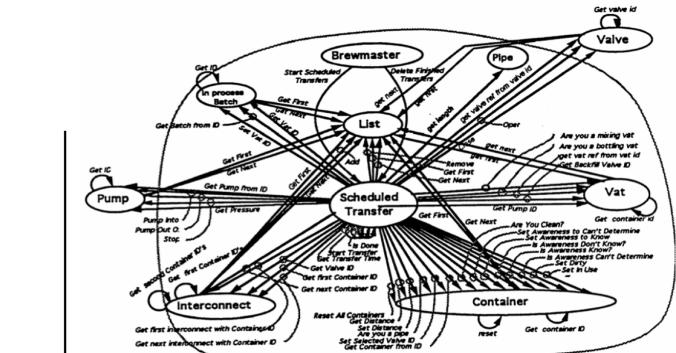
## REPASO (UD3)

- Uno de los principios básicos de la ingeniería del software es **“incrementar la cohesión, reducir el acoplamiento”**.
- Componentes con bajo acoplamiento y alta cohesión facilitan su utilización y su interconexión.



# Principios generales de diseño

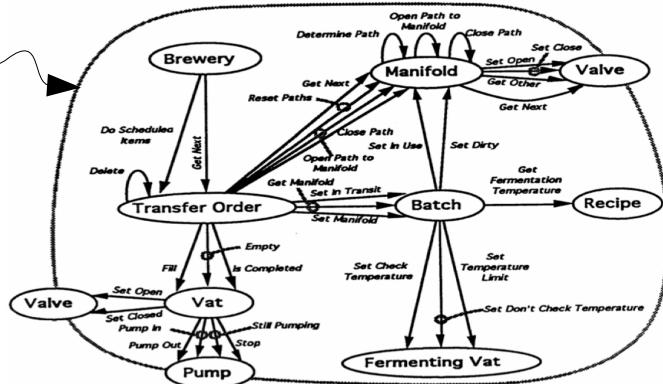
## Acoplamiento vs. cohesión



sistema demasiado acoplado

Acoplamiento

sistema desacoplado y más cohesionado





- 1. Principio abierto-cerrado**
- 2. Principio de sustitución (Liskov)**
- 3. Principio de responsabilidad única**
- 4. Principio de segregación de interfaz**
- 5. Principio de inversión de dependencias**

# 1. Principio abierto-cerrado



- “Todos los sistemas cambian durante su ciclo de vida. Esto debe tenerse en cuenta al desarrollar sistemas que se pretende que duren más allá de la primera versión” (1)
- Principio abierto-cerrado
  - ***“Las entidades software (clases, paquetes, métodos, etc.) deben estar abiertas a su extensión, pero cerradas a su modificación.”*** (2)
  - Lo que ya funciona no se modifica (si acaso se refactoriza). Cuando los requisitos cambian, se extiende el comportamiento de estos módulos añadiendo código nuevo (nuevos métodos, clases, ...)

(1) Object Oriented Software Engineering a Use Case Driven Approach, Ivar Jacobson, Addison Wesley, 1992, p 21.

(2) Object Oriented Software Construction, Bertrand Meyer, Prentice Hall, 1988, p 23

# 1. Principio abierto-cerrado



La herencia de interfaz sirve para esto:

- Las clases abstractas y sus derivadas son una forma de implementar conceptos fijos (cerrados) pero que al mismo tiempo representan un conjunto ilimitado de comportamientos.
- El código cliente que usa la interfaz de la clase abstracta puede cerrarse a la modificación, pero mantenerse abierto a la extensión mediante la creación de clases derivadas.

# 1. Principio abierto-cerrado



## ■ Ejemplo

Una aplicación debe ser capaz de dibujar círculos y cuadrados en un orden determinado. Dada una lista de círculos y cuadrados en el orden apropiado, la aplicación debe recorrerla y dibujar cada figura.

```
//Solución sin RTTI ni enlace dinámico

enum TipoFigura { circulo, cuadrado }
class Figura {
 private TipoFigura tipo;
 public getTipo() { return tipo; }
}
class Circulo extends Figura {
 private Punto centro;
 private float radio;
 public void dibujarCirculo() { ... }
}
class Cuadrado extends Figura {
 private Punto puntoSuperiorIzqda;
 private float lado;
 public void dibujarCuadrado() { ... }
}
```

```
// código cliente (¡apesta!)

void dibujarFiguras(List<Figura> lf) {
 Iterator<Figura> it = lf.iterator();
 while (it.hasNext()) {
 Figura fig = it.next();
 switch(fig.getTipo()) {
 case circulo:
 Circulo ci = (Circulo)fig;
 ci.dibujarCirculo(); break;
 case cuadrado:
 Cuadrado cu = (Cuadrado)fig;
 cu.dibujaCuadrado(); break;
 }
 }
}
```

# 1. Principio abierto-cerrado



- Ejemplo (cont.)
  - *dibujarFiguras()* no cumple el principio abierto-cerrado, porque no está cerrado a la adición de nuevas figuras.
  - Además, la sentencia switch aparecerá allá donde sea necesario discriminar entre diferentes tipos de figuras.

# 1. Principio abierto-cerrado



## ■ Ejemplo (cont.)

```
// Solución con RTTI // código cliente (sigue apestando!)

class Figura { } void dibujarFiguras(List<Figura> lf) {
class Circulo extends Figura { Iterator<Figura> it = lf.iterator();
 private Punto centro; while (it.hasNext()) {
 private float radio; try {
 public void dibujarCirculo() // downcasting no seguro:
 { ... } Circulo ci = (Circulo)it.next();
 } Cuadrado cu = (Cuadrado)it.next();
class Cuadrado extends Figura { ci.dibujarCirculo();
 private Punto puntoSupIzqda; cu.dibujarCuadrado();
 private float lado; } catch (ClassCastException cce) { ... }
 public void dibujarCuadrado() }
 { ... } }
} }
```

*dibujarFiguras()* sigue sin cumplir el principio abierto-cerrado

(Sólo debemos usar RTTI cuando no viole el principio abierto-cerrado)

# 1. Principio abierto-cerrado



## ■ Ejemplo (cont.)

Solución que cumple el principio abierto-cerrado

```
abstract class Figura { // código cliente
 abstract public void dibujar();
}
class Circulo extends Figura {
 private Punto centro;
 private float radio;
 public void dibujar() { ... }
}
class Cuadrado extends Figura { }
 private Punto puntoSupIzqda;
 private float lado;
 public void dibujar() { ... }
}
```

```
void dibujarFiguras(List<Figura> lf) {
 Iterator<Figura> it = lf.iterator();
 while (it.hasNext()) {
 Figura fig = it.next();
 fig.dibujar();
 }
}
```

*dibujarFiguras()* está cerrada frente a la adición de nuevas figuras. Su comportamiento se puede extender sin cambiar su código, añadiendo nuevas clases derivadas de Figura.

# 1. Principio abierto-cerrado



- Una aplicación 'real' no puede estar 100% cerrada.
  - Por ejemplo, considera que sucedería si en el método dibujarFiguras() decidiéramos que los círculos deben ser dibujados antes que los cuadrados.
- Siempre existirá algún tipo de cambio para el cual una aplicación no esté cerrada.
- El diseñador debe decidir, basándose en su experiencia, ante qué tipo de cambios va a cerrar su código.

# 1. Principio abierto-cerrado



- La clausura del código ante cambios se consigue mediante el uso de la abstracción.
- En el caso de nuestro ejemplo, lo que necesitaríamos sería alguna forma de abstraer la forma en que las figuras se ordenan: dadas dos figuras ¿cuál debemos dibujar primero? (\*)

(\*) La solución, aquí: <http://www.objectmentor.com/resources/articles/ocp.pdf>

# 1. Principio abierto-cerrado



- Heurísticas

- Encapsulación: Todos los atributos deben tener visibilidad privada
  - Ningún método que dependa de un atributo puede ser cerrado con respecto a éste.
- Variables globales no, gracias.
  - Ningún método que dependa de una variable global puede ser cerrado con respecto a ésta.
  - Existen ciertas excepciones, como `System.out`
- Usar RTTI es peligroso (ejemplo pág. 18)

# 1. Principio abierto-cerrado



## ■ Conclusiones

- *Seguir este principio proporciona los mayores beneficios asociados a la tecnología orientada a objetos: reusabilidad y mantenibilidad.*
- *Esto no se consigue simplemente usando un lenguaje orientado a objetos. Requiere aplicar la abstracción en aquellas partes de una aplicación que se prevee puedan cambiar.*

## 2. Principio de sustitución (Liskov)



### Principio de sustitución (Liskov)

- Ya visto en UD 3, cuando hablamos de 'herencia de interfaz'.
- Enunciado original (de Barbara Liskov):
  - *“Si para cada objeto o1 de tipo S existe un objeto o2 de tipo T tal que, para todos los programas P definidos en términos de T, el comportamiento de P no cambia cuando se sustituye o2 por o1, entonces S es un subtipo de T”* - Barbara Liskov, Data Abstraction and Hierarchy, SIGPLAN Notices, 23,5 (May, 1988).
- *“El comportamiento de P no cambia”* no significa que su efecto es el mismo usemos un objeto de tipo S o T (S puede modificar el comportamiento base), sino que P no basa su interacción con los objetos o1 u o2 en saber si son de tipo S o T.

## 2. Principio de sustitución (Liskov)



- Enunciado alternativo:
  - Los métodos que usan referencias a clases base deben ser capaces de utilizar objetos de clases derivadas sin saberlo.
- Un método que no cumple este principio es aquél que, aún usando referencias a clase base, necesita conocer a las clases derivadas. No cumplirá tampoco el principio abierto-cerrado.
- Implica usar únicamente la interfaz de la clase base y respetar su contrato.

## 2. Principio de sustitución (Liskov)

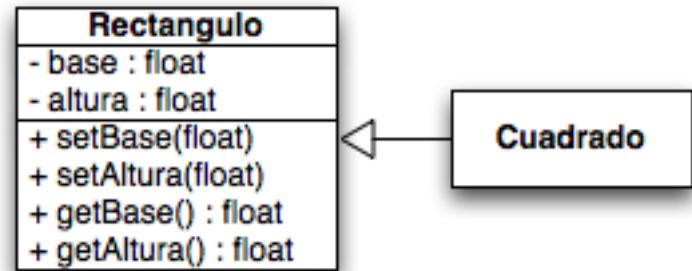


### Un ejemplo sutil...

Un cuadrado ES UN rectangulo cuya base es igual a su altura.

Pero,...

- En realidad no necesitamos dos atributos para el cuadrado.
- *setBase()* y *setAltura()* no son apropiadas para el cuadrado. Podemos sobreescribirlas.



```
void setBase(float b) {
 super.setBase(b);
 super.setAltura(b);
}

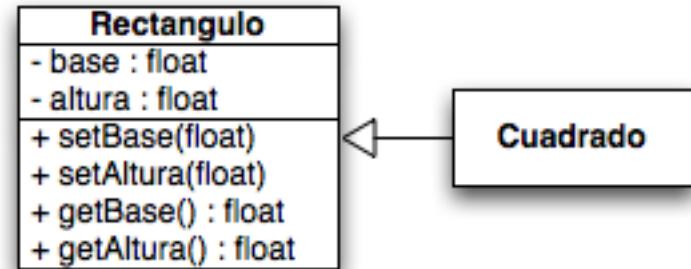
void setAltura(float a) {
 super.setBase(a);
 super.setAltura(a);
}
```

## 2. Principio de sustitución (Liskov)



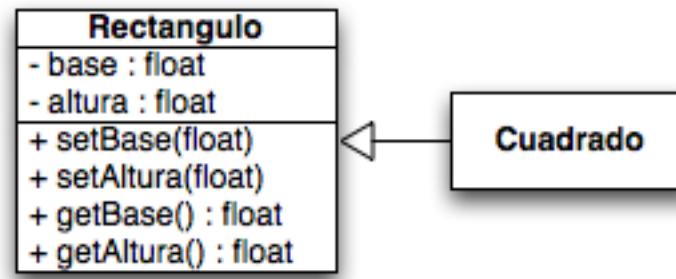
Ahora el modelo parece consistente, pero...

```
// código cliente: prueba unitaria
void testSetters(Rectangulo r) {
 r.setBase(5);
 r.setAltura(4);
 assertEquals("Setters", r.getBase() * r.getAltura(), 20);
}
```



- El problema: El programador del test asumió que cambiar la altura de un rectángulo no modifica su base.
- Por tanto, existen métodos que usan referencias a clase base, Rectangulo, que no funcionan correctamente con clases derivadas como Cuadrado.
- No se respetó el contrato: Un cuadrado no se comporta como un rectángulo. El principio de sustitución indica que en diseño OO, la relación ES-UN debe cumplirse en términos del comportamiento público (el contrato) y no sólo del estado interno.

## 2. Principio de sustitución (Liskov)



### Diseño por contrato (1)

Ofrece mecanismos para asegurar que se cumple el principio de sustitución:

- Los métodos declaran precondiciones y postcondiciones:
  - Las precondiciones deben ser ciertas para que el método se pueda ejecutar
  - Cuando termina, el método garantiza que las postcondiciones se cumplen.
- Una posible postcondición para *Rectangulo.setAltura()* podría ser  
`assert( (altura == a) && (base == old.base) )`

(1) *Object Oriented Software Construction*, Bertrand Meyer, Prentice Hall, 1988

### 3. Principio de inversión de dependencias

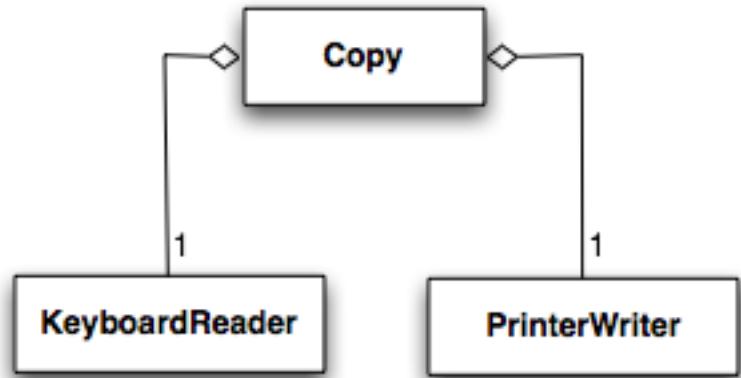


- **Rasgos de un “mal diseño” en el software**
  - Es difícil de cambiar porque cualquier cambio afecta a demasiadas partes del sistema : **Rigidez**
  - Cuando realizamos un cambio, aparecen problemas en sitios inesperados: **Fragilidad**
  - Es complicado de reutilizar en otras aplicaciones, porque no puede ser desacoplado de la aplicación actual: **Inmovilidad**
- La interdependencia entre módulos es la culpable de un mal diseño.

### 3. Principio de inversión de dependencias



- Ejemplo: El módulo 'Copiar'



```
void Copy()
{
 int c;

 while ((c = KeyboardReader.read()) != EOF)
 PrinterWriter.write(c)
}
```

- Copy() no es reutilizable para copiar de algo que no sea un teclado a algo que no sea una impresora.

### 3. Principio de inversión de dependencias



- **Ejemplo: El módulo 'Copiar'**

Supongamos que deseamos poder copiar también de teclado a fichero:

```
enum OutputDevice {printer, disk}

void Copy(OutputDevice dev)
{
 int c;
 while ((c = KeyboardReader.read()) != EOF)
 if (dev == printer)
 PrinterWriter.write(c);
 else DiskWriter.write(c);
}
```

Esto añade más dependencias. Cuanto más tipos de dispositivos añadimos, más dependencias aparecen.

### 3. Principio de inversión de dependencias

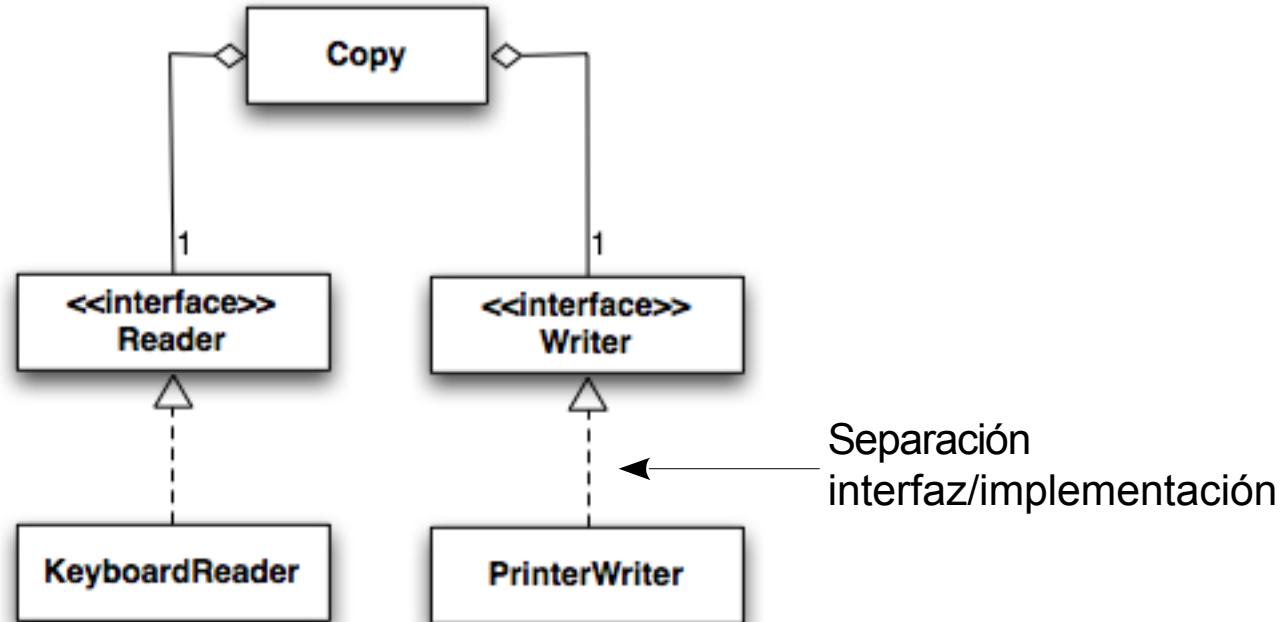


- Ejemplo: El módulo 'Copiar'

#### Inversión de dependencias:

El módulo de alto nivel (Copy()), depende de los módulos de bajo nivel

Hemos de hacer Copy() independiente de los detalles de bajo nivel.



### 3. Principio de inversión de dependencias



- **Ejemplo: El módulo 'Copiar'**

```
interface Reader
{
 char read();
}

interface Writer
{
 void write(char);
}

void Copy(Reader r, Writer w)
{
 int c;
 while((c = r.read()) != EOF)
 w.write(c);
}
```

Las dependencias han sido invertidas:  
Copy() depende de los interfaces  
(abstracciones) y los lectores y  
escritores específicos también.

### 3. Principio de inversión de dependencias



#### ■ **Principio de inversión de dependencias**

A. Los módulos de alto nivel no deben depender de los de bajo nivel.  
Ambos deben depender de abstracciones.

B. Las abstracciones no deben depender de detalles específicos. Éstos  
deben depender de las abstracciones.

En este principio está basado el diseño de 'frameworks'.

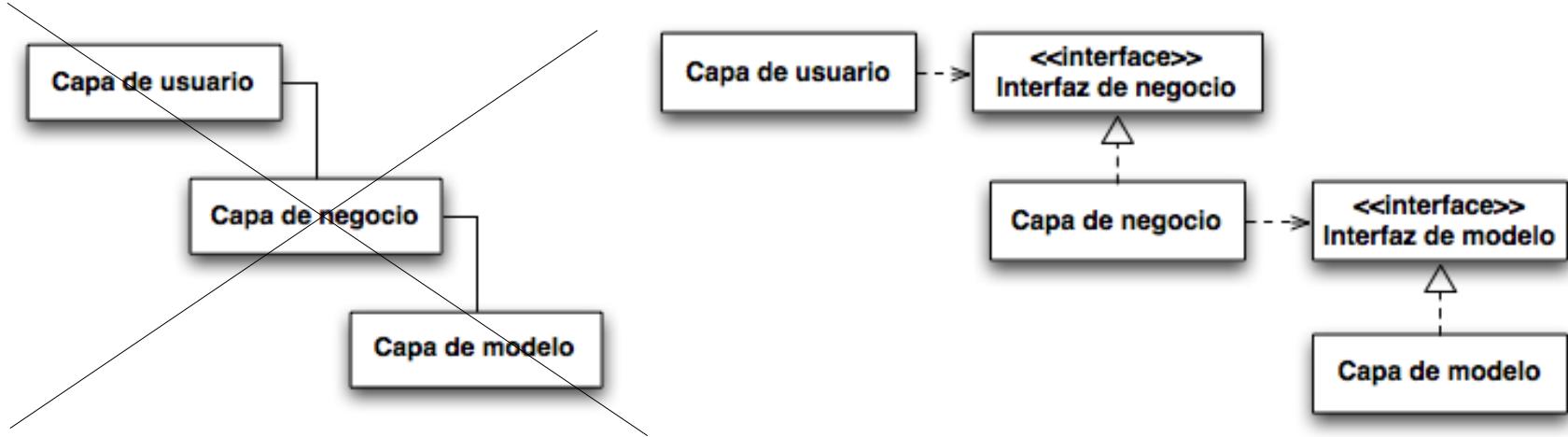
### 3. Principio de inversión de dependencias



- **Principio de inversión de dependencias**

Según Booch(\*):

“Las arquitecturas orientadas a objetos bien estructuradas están diseñadas por capas, donde cada capa proporciona un conjunto coherente de servicios a través de una interfaz bien definida.”



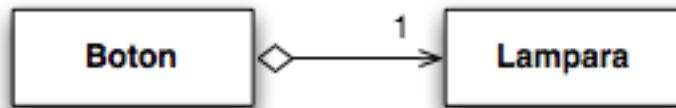
(\*) *Object Solutions*, Grady Booch, Addison Wesley, 1996, p54

### 3. Principio de inversión de dependencias



#### ■ Ejemplo: el Botón y la Lámpara

La inversión de dependencias se puede aplicar allí donde una clase envía un mensaje a otra.



```
class Lampara {
 public void encender() {}
 public void apagar() {}
}
```

(No es posible reusar Boton para controlar un Motor, p. ej.)

```
class Boton {
 public Boton(Lampara l)
 { lamp = l; }
 public void detectar()
 {
 boolean botonOn = getEstado();
 if (botonOn)
 lamp.encender();
 else
 lamp.apagar();
 }
 private Lampara lamp;
}
```

### 3. Principio de inversión de dependencias

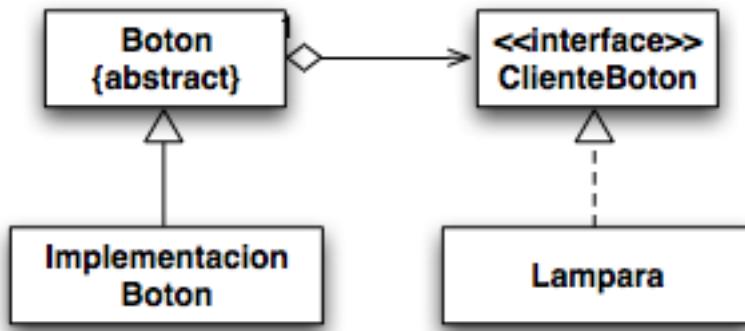


- **Ejemplo: el Boton y la Lampara**

Para aplicar el principio de inversión de dependencias, es necesario encontrar la abstracción subyacente en el diseño.

En el ejemplo Boton/Lampara la abstracción consiste en detectar un estado On/Off y pasar esta información a otro objeto.

El mecanismo usado para detectar el estado, o el tipo de objeto a controlar (la lámpara) es irrelevante.



**Ejercicio:** Escribe código en Java para este diseño que realice la misma función que el de la página anterior.

### 3. Principio de inversión de dependencias



#### ■ Conclusiones

El principio de inversión de dependencias implica la separación de interfaz e implementación en distintos módulos.

Corolario: Las clases abstractas no deben depender de clases concretas. Las clases concretas son las que deben tener dependencias de clases abstractas

## 4. Principio de segregación de interfaz



### ■ Interfaces 'gruesas'

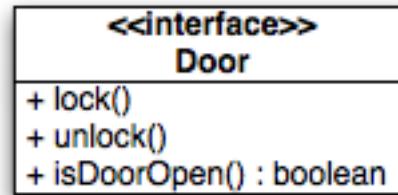
- Son aquellas que tienen muchos métodos públicos. Suelen indicar una falta de cohesión en el interfaz.
- Se pueden dividir en grupos de métodos, cada uno de los cuales sirve a un grupo diferente de clientes (diferentes responsabilidades).
- El principio de segregación de interfaz admite que hay objetos que requieren interfaces no cohesivas, pero sugiere que este interfaz no tiene que estar definido en una única clase.
- El código cliente manipula estos objetos a través de diferentes clases abstractas o interfaces bien cohesionados.

# 4. Principio de segregación de interfaz

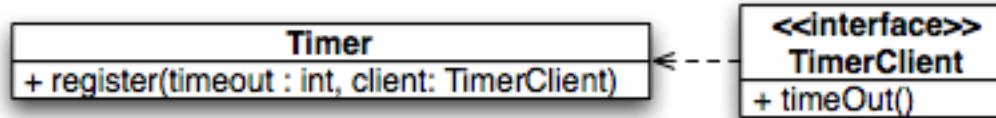


## ■ Polución de interfaz

- Imaginemos un sistema de seguridad. En éste hay puertas (Door) que pueden estar abiertas o cerradas, y que saben si están abiertas o cerradas:



- Supongamos que hay un tipo de puertas (TimedDoor) que hacen sonar una alarma cuando la puerta permanece abierta demasiado tiempo. Para hacerlo, la puerta se comunica con otro objeto (Timer), el cual invoca a un método `timeOut()` cuando el tiempo se agota. Para ello, la puerta debe implementar el interface TimerClient y registrarse en el objeto Timer.

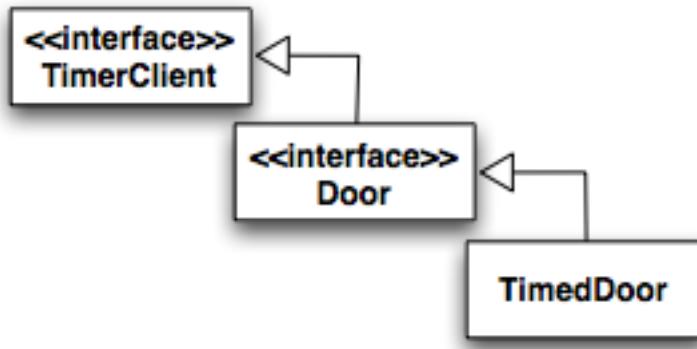


# 4. Principio de segregación de interfaz



## ■ Polución de interfaz

- Solución común:



- Problemas: no todos los tipos de puertas necesitan un Timer. éstas se verán obligadas a proporcionar una implementación vacía del método `timeOut()`.
- Esto se conoce como el síndrome de la polución de interfaz: la interfaz de Door ha sido contaminada por un interfaz que sólo necesitan algunas de sus subclases, convirtiéndose en un interfaz 'grueso'.
- Todos los clientes de Door se ven afectados por el cambio de interfaz.

## 4. Principio de segregación de interfaz



### ■ El principio de segregación de interfaz

El código cliente no debe ser forzado a depender de interfaces que no utiliza.

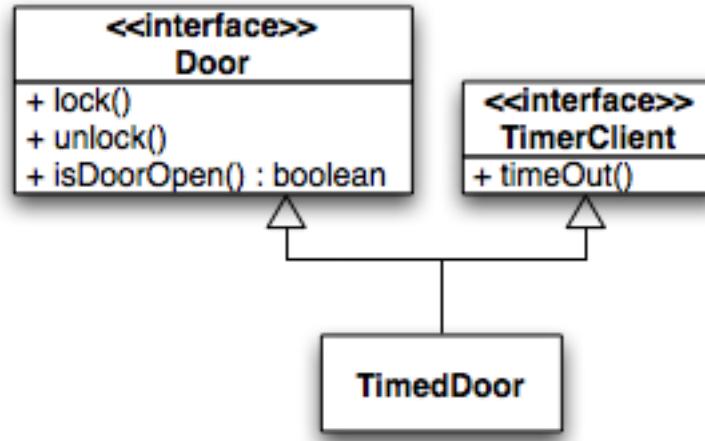
¿cómo separar interfaces cuando deben estar juntos? (como en el caso de Door)

- Los clientes de un objeto no necesariamente acceden a él a través de la interfaz de la clase del objeto.
  - P. ej.: Separación mediante herencia de interfaz múltiple

## 4. Principio de segregación de interfaz



- Separación mediante herencia de interfaz múltiple



- Los clientes tanto de Door como de TimerClient pueden hacer uso de TimedDoor, pero no tienen ninguna dependencia de ella. Usan el mismo objeto a través de interfaces diferentes.

# 4. Principio de segregación de interfaz



## ■ Conclusiones

- Los interfaces 'gruesos' o 'hinchados' conducen a acoplamientos entre clientes que de otra forma deberían estar aislados (desacoplados) entre sí.
- Estos interfaces pueden segregarse en diferentes clases abstractas o interfaces que rompen el acoplamiento no deseado entre los clientes.

## 5. Principio de responsabilidad única



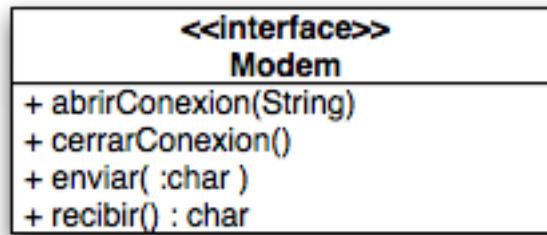
Nunca debe existir más de una razón para que una clase cambie.

- Cada responsabilidad asociada a una clase es una razón para que la clase cambie.
- Si una clase asume más de una responsabilidad, entonces existirá más de una razón para que la clase cambie.
- Si una clase asume dos o más responsabilidades, entonces estas se acoplan: cambios en una de ellas pueden impedir satisfacer alguna de las otras → Fragilidad

# 5. Principio de responsabilidad única



- Ejemplo: Interface Modem

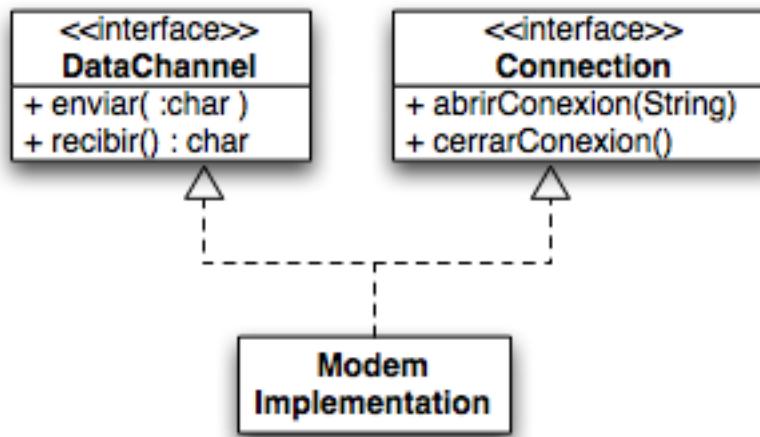


- Parece un diseño razonable, pero Modem está asumiendo dos responsabilidades diferentes.
  - 1. Gestión de la conexión (abrirConexion/cerrarConexion)
  - 2. Comunicación de datos (enviar/recibir)
- Ambos grupos de métodos pueden cambiar por razones diferentes
- Además, son invocados por partes diferentes de la aplicación, que también cambiarán por razones diferentes.

## 5. Principio de responsabilidad única



- Ejemplo: Interface Modem
  - Este diseño separa las responsabilidades en interfaces diferentes:



- Las dos responsabilidades son re-acopladas en *ModemImplementation*. Sin embargo, separando los interfaces hemos separado los conceptos, de forma que ningún código cliente (con la posible excepción de algún método factoría) depende de dicha clase, sino de los interfaces desacoplados.

## 5. Principio de responsabilidad única



### ■ Conclusiones

- El principio de responsabilidad única es uno de los más simples, y a la vez de los más difíciles de cumplir correctamente
- Unir responsabilidades es algo que hacemos de forma natural.
- Encontrarlas y separarlas es, en su mayor parte, de lo que realmente trata el diseño de software.

# Principios de diseño orientado a objetos



- Los principios vistos hasta ahora tienen que ver con el diseño de clases.
- Por sus iniciales en inglés se les conoce como el conjunto de principios **SOLID**:
  - Single Responsibility Principle (SRP)
  - Open-Closed Principle (OCP)
  - Liskov Substitution Principle (LSP)
  - Interface Segregation Principle (ISP)
  - Dependency Inversion Principle (DIP)
- Existen otros principios de diseño OO de más alto nivel que tratan, por ejemplo, de la cohesión y acoplamiento a nivel de paquetes.

# Un vistazo a los patrones de diseño



- Cuando nos enfrentamos a un nuevo problema ¿dónde buscamos inspiración?
- La mayoría busca soluciones a problemas previos que tienen características similares
- **Patrones de diseño**
  - Un patrón de diseño es una solución a un problema de diseño.
  - Para que una solución sea considerada un patrón debe poseer ciertas características.
    - Se debe haber comprobado su efectividad resolviendo problemas similares en ocasiones anteriores.
    - Debe ser reutilizable, lo que significa que es aplicable a problemas de diseño similares en distintas circunstancias.

# Un vistazo a los patrones de diseño



- Los patrones de diseño aceleran el proceso de encontrar una solución, eliminando la necesidad de reinventar soluciones ya probadas y bien conocidas.
- Muchos patrones tienen que ver con el concepto de dependencia, e intentan aumentar la cohesión de la solución al tiempo que reducen su acoplamiento.

# Un vistazo a los patrones de diseño



- Un ejemplo simple: el patrón **Adaptador**
  - Un adaptador es un componente que se usa para conectar un cliente (que necesita algún servicio) con un servidor (que proporciona este servicio)
  - El problema es que el cliente requiere un determinado interfaz. Aunque el servidor proporciona la funcionalidad deseada, no soporta el interfaz requerido.
  - El adaptador cambia la interfaz del servidor, delegando el trabajo en éste.

# Un vistazo a los patrones de diseño



- Un ejemplo simple: el patrón **Adaptador**

```
class MiAdaptadorColeccion implements Collection {

 public boolean isEmpty ()
 { return data.cuantos() == 0; }
 public int size ()
 { return data.cuantos(); }
 public void add (Object nuevo)
 { data.colocarAlFinal(nuevo); }
 public boolean contains (Object test)
 { return data.busca(test) != null; }
 // ... etc

 private Vectorcito data = new Vectorcito();
}
```

- 'Vectorcito' es una clase contenedora que no implementa la interfaz de Collection
- Los adaptadores se usan a menudo para conectar software de diferentes fabricantes.



# Bibliografía

- Robert C. Martin. ***Principles of object oriented design***  
<http://c2.com/cgi/wiki?PrinciplesOfObjectOrientedDesign>
- T.C. Lethbridge, R. Laganiere. ***Object-oriented software engineering : practical software development using UML and Java***
  - Sec. 9.1 (*el proceso de diseño*)
- T. Budd. ***An Introduction to Object-oriented Programming, 3rd ed.***
  - Cap 23 (*acoplamiento y cohesion*)
  - Cap 24 (*patrones de diseño*)
- E. Gamma et al. ***Design Patterns. Elements of Reusable Object-Oriented Software.***