

U.D. 5

HERENCIA

Cristina Cachero, Pedro J. Ponce de León

versión 20111015



Tema 3. HERENCIA

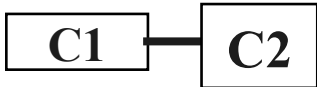


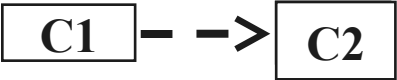
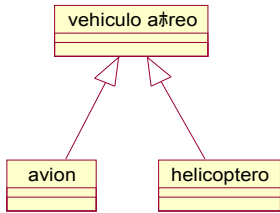
Objetivos



- 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.

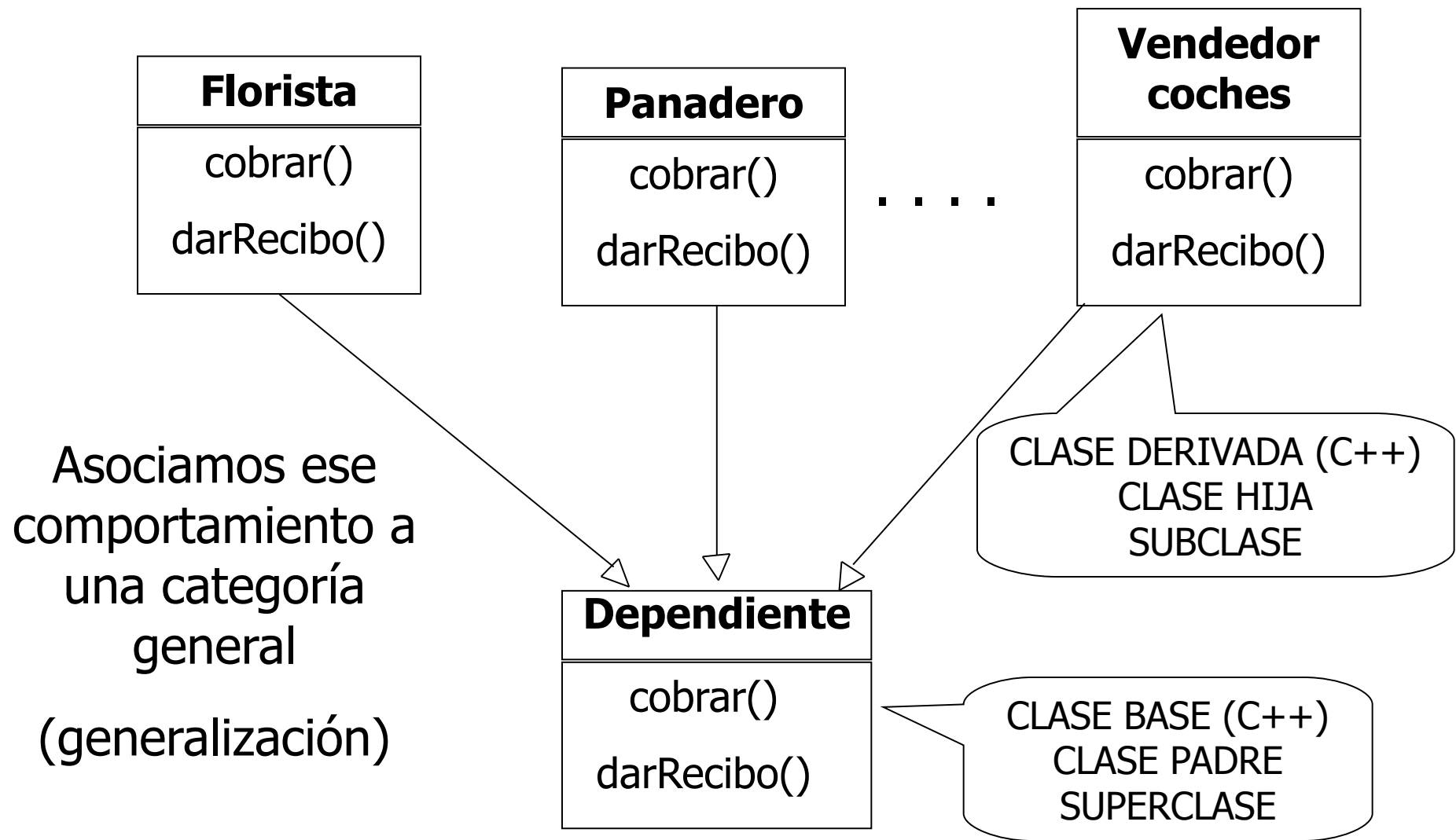
Herencia

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



- 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*.

- 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

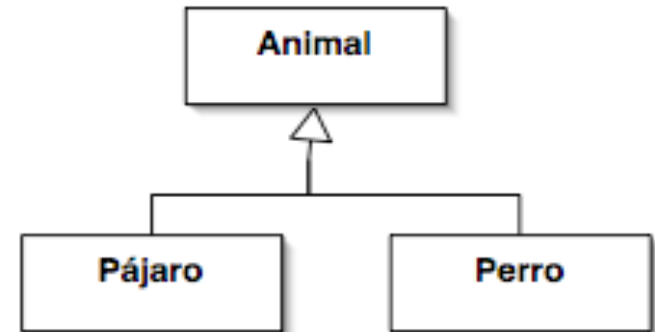
- *La herencia como reutilización de código:* Una clase derivada puede heredar comportamiento de una clase base, por tanto, el código no necesita volver a ser escrito para la derivada.
 - **Herencia de implementación**

- *La herencia como reutilización de conceptos:* Una clase derivada **sobrescribe** 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**

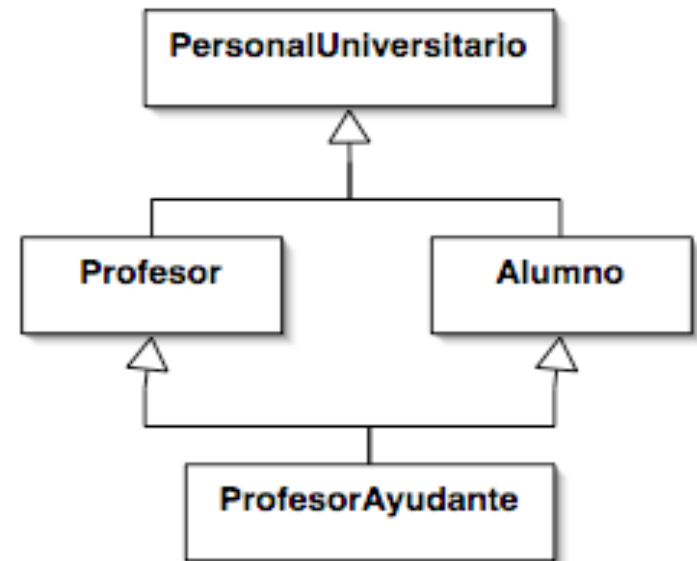
- Simple/Múltiple
- De implementación/de interfaz

- 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 sobreescribirse 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**

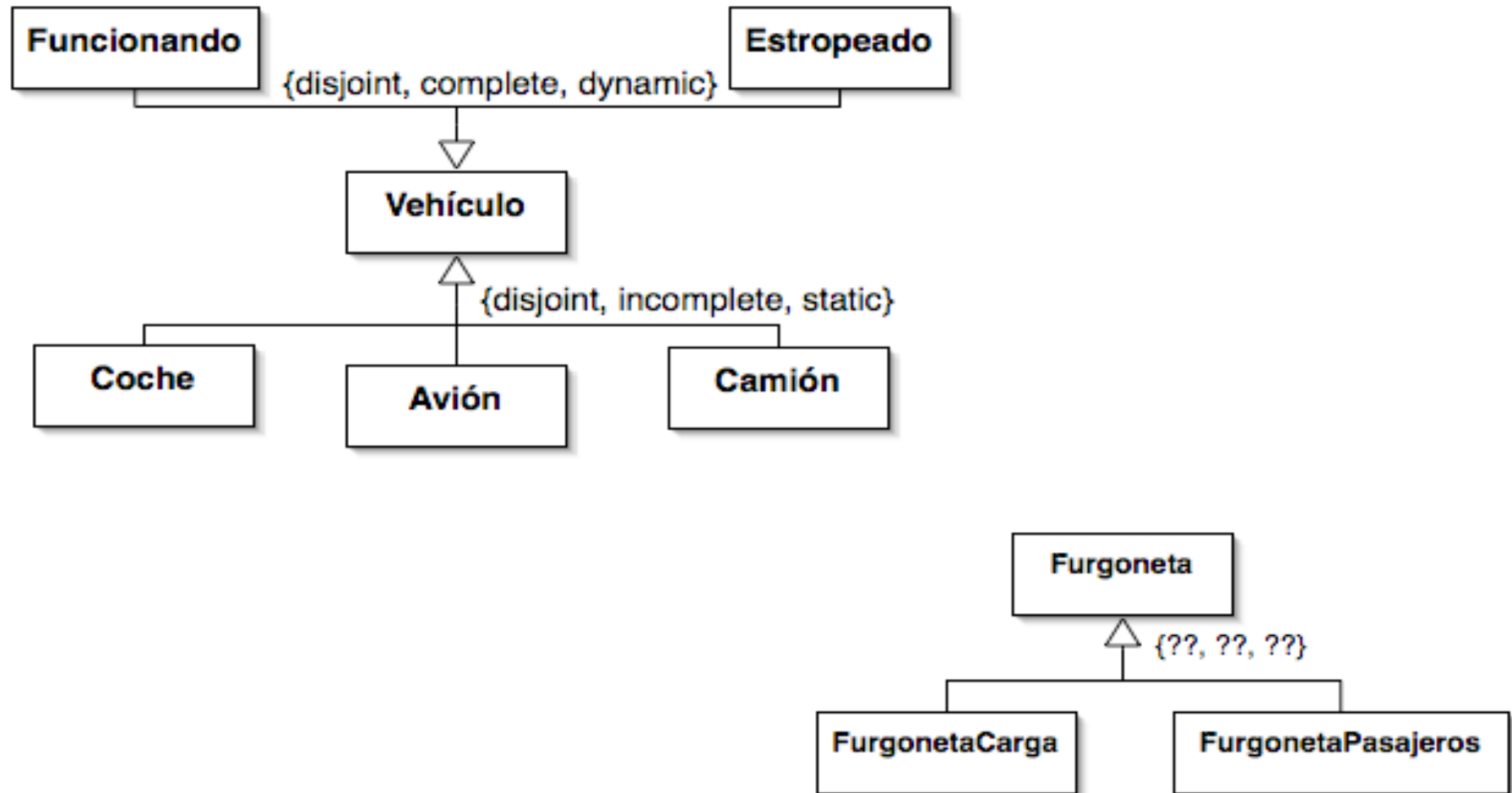
- Determina si todas las instancias de la clase padre son *a la vez* instancias de alguna de las clases hijas (completa) o, por el contrario, hay objetos de la clase padre que no pertenecen a ninguna subcategoría de las reflejadas por las clases hijas (incompleta).

- **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

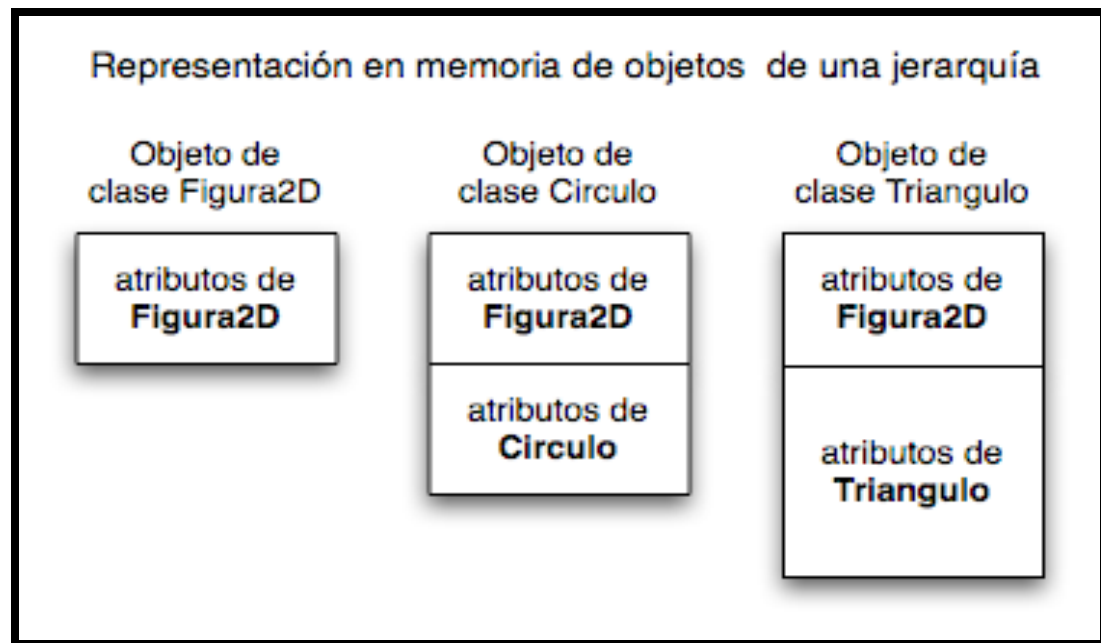
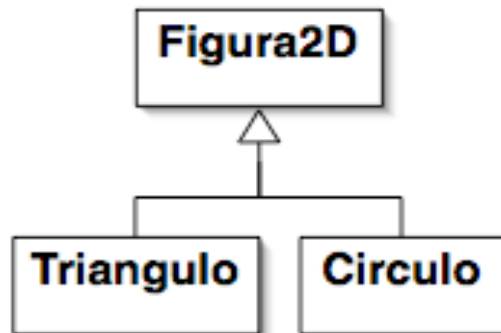
Caracterización: ejemplos



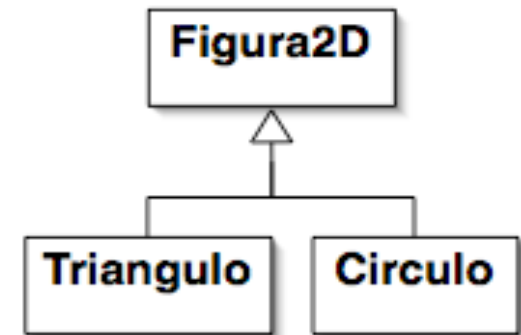
HERENCIA DE IMPLEMENTACIÓN

Herencia Simple

- 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)



```
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();
```

- **Ámbito de visibilidad `protected`**

- Los atributos/métodos *protected* son directamente accesibles desde la propia clase y sus clases derivadas. Tienen visibilidad privada para el resto de ámbitos. En UML, se especifica con `#`.

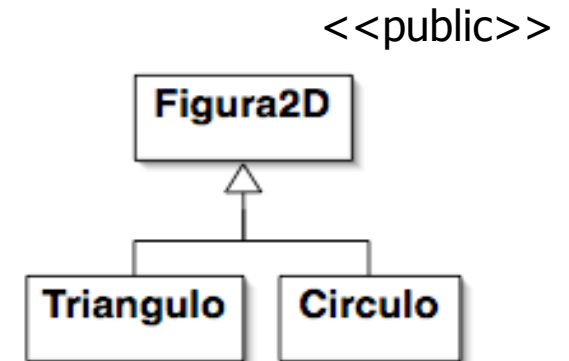
```
class Figura2D {  
    protected Color colorRelleno;  
    ...  
}  
  
class Circulo extends Figura2D {  
    public void vaciarCirculo() {  
        colorRelleno=NINGUNO; //OK, protected  
    }  
    ...  
}
```

```
//código cliente  
Circulo c;  
c.colorRelleno=NINGUNO;  
// ¡ERROR! colorRelleno  
// es privado aquí
```

- Herencia Pública
 - Se hereda interfaz e implementación

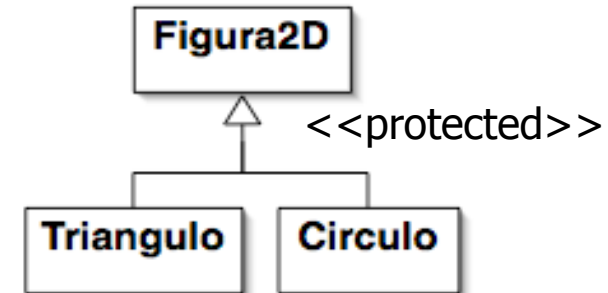
```
// JAVA sólo soporta herencia pública  
class Circulo extends Figura2D  
{  
...  
}
```

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



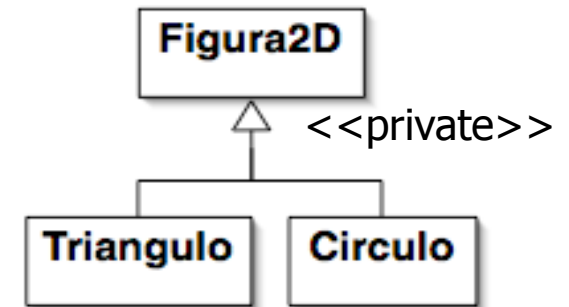
- Herencia Protegida (C++)

```
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 innacesible desde objetos de clase derivada.

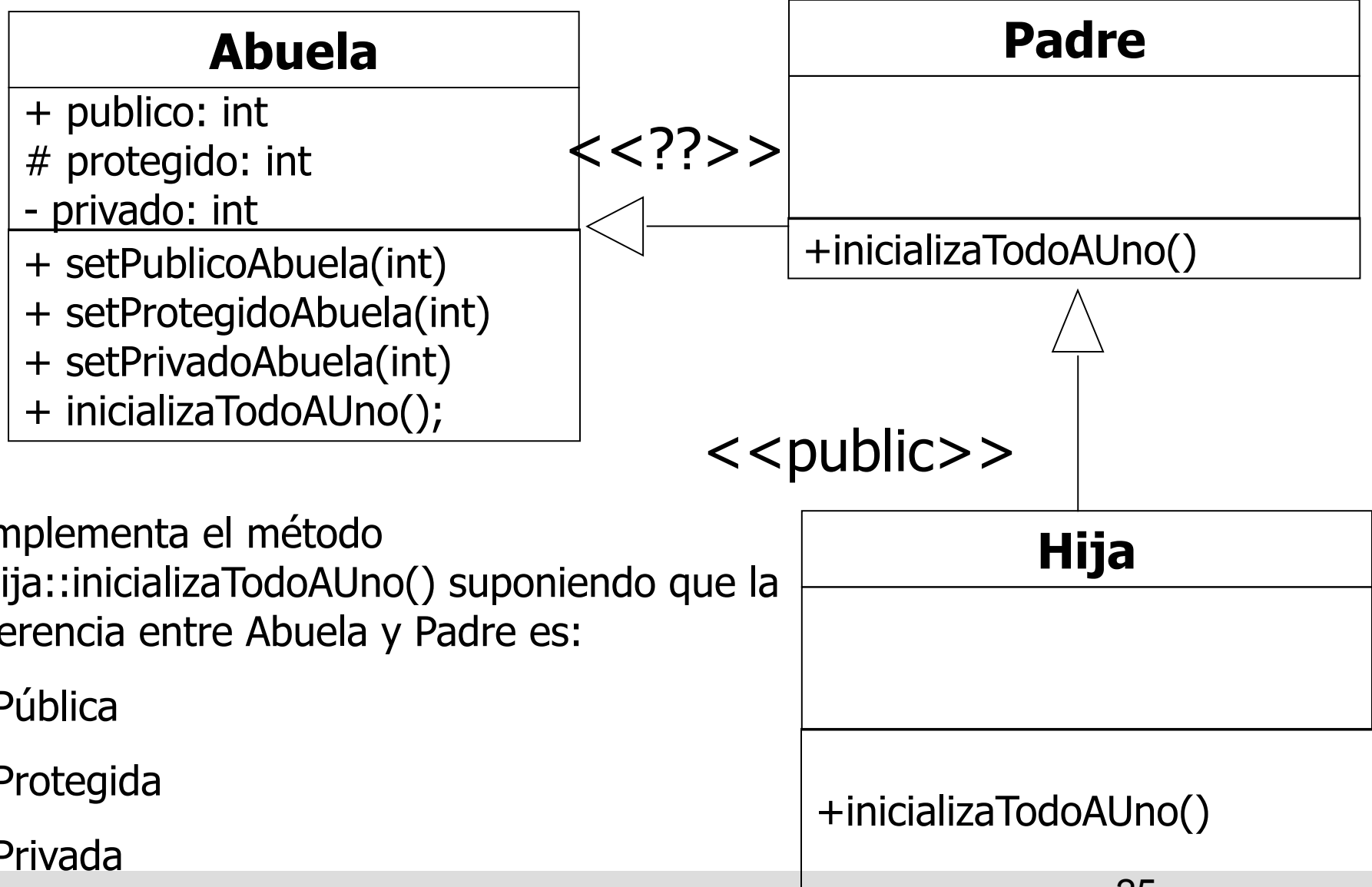
Tipos de Herencia Simple

<div>Ámbito Herencia</div> <div>Visibilidad en clase base</div>	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



Implementa el método
`Hija::inicializaTodoAUno()` suponiendo que la
herencia entre **Abuela** y **Padre** es:

- Pública
- Protegida
- Privada

- 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");  
    }  
}
```

Herencia Simple

Métodos en las clases derivadas

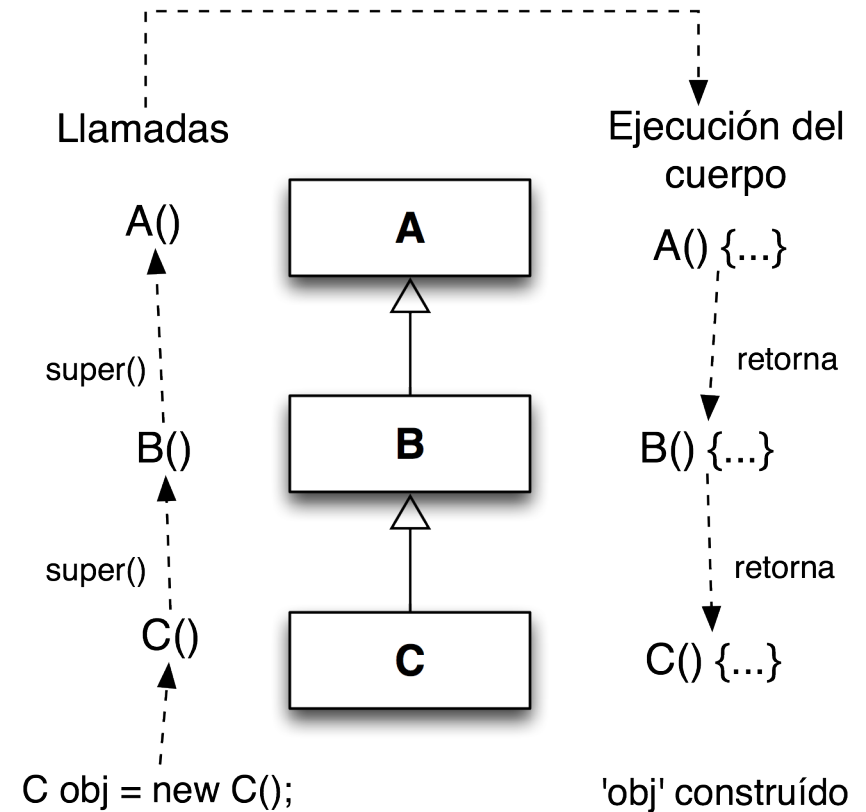


- Ejemplo de **refinamiento** en Java

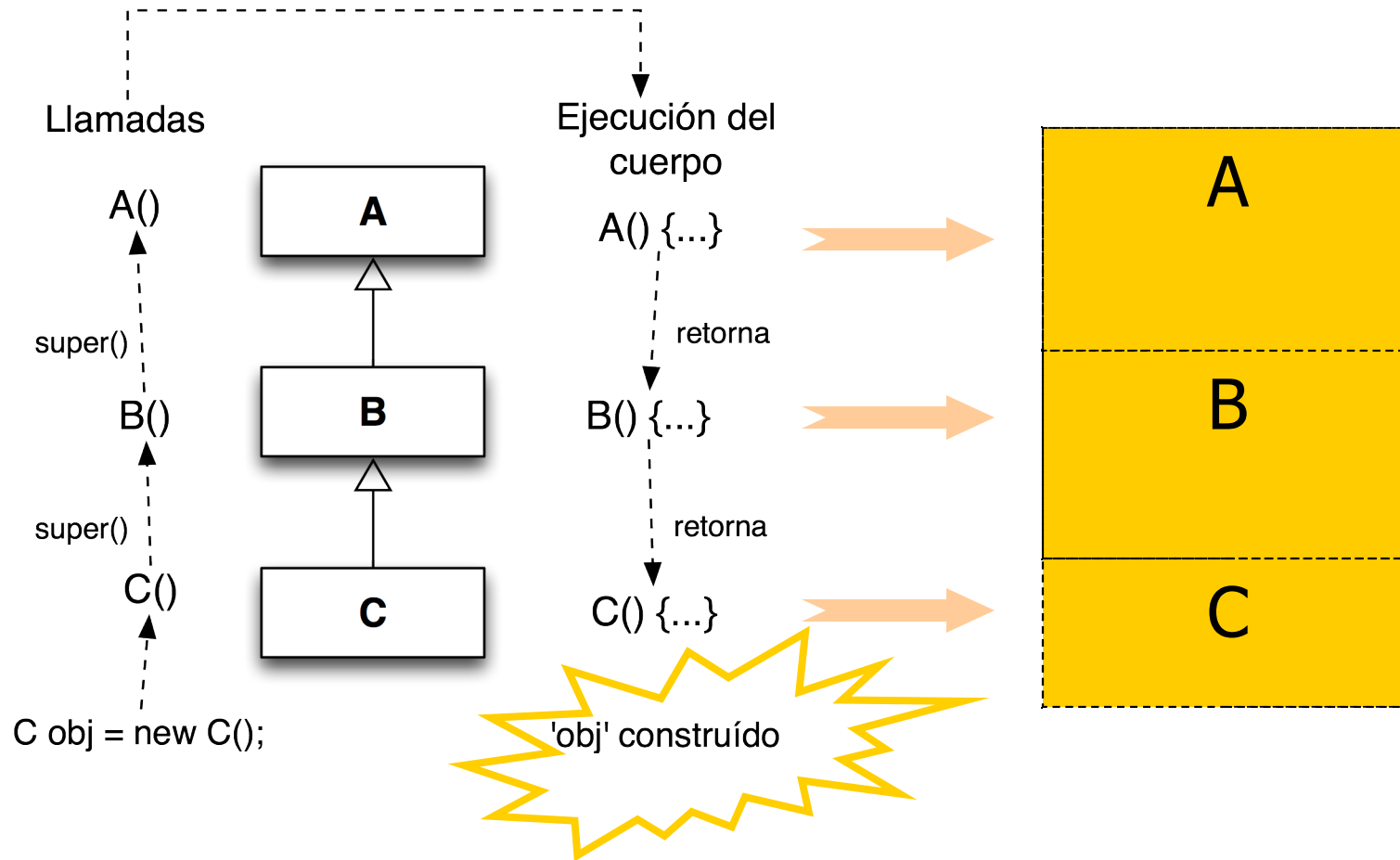
```
class A {  
    public void doIt() { System.out.println("HECHO en A."); }  
    public void doItAgain() {  
        System.out.println("HECHO otra vez en A.");  
    }  
}  
class B extends A {  
    public void doIt() {  
        System.out.println("HECHO en B.");  
        super.doIt(); // impl.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

- 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



- 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.

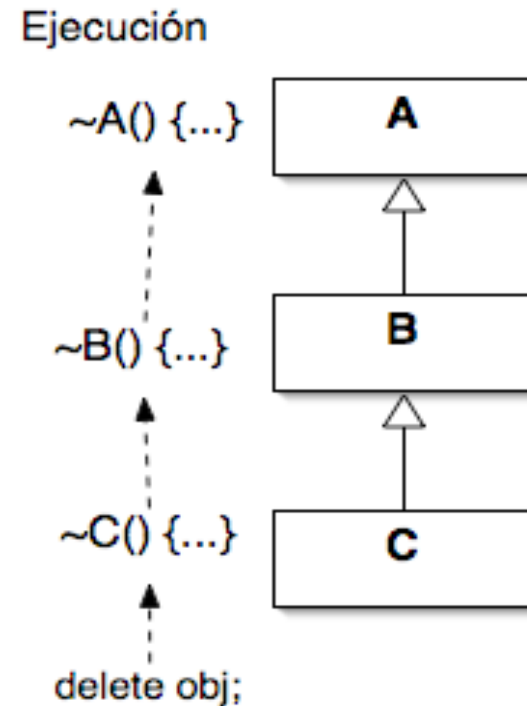
(CONSEJO: Inicialización de atributos de la clase base: en la clase base, no en la derivada)

■ Ejemplo

```
class Figura2D {
    private Color colorRelleno;
    public Figura2D() { colorRelleno= Color.NINGUNO; }
    public Figura2D(Color c) { colorRelleno=c; }
    public Figura2D(Figura2D f) { colorRelleno=f.colorRelleno; }
    ...}

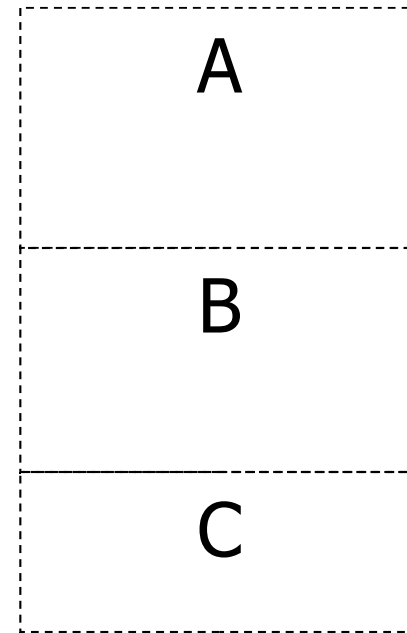
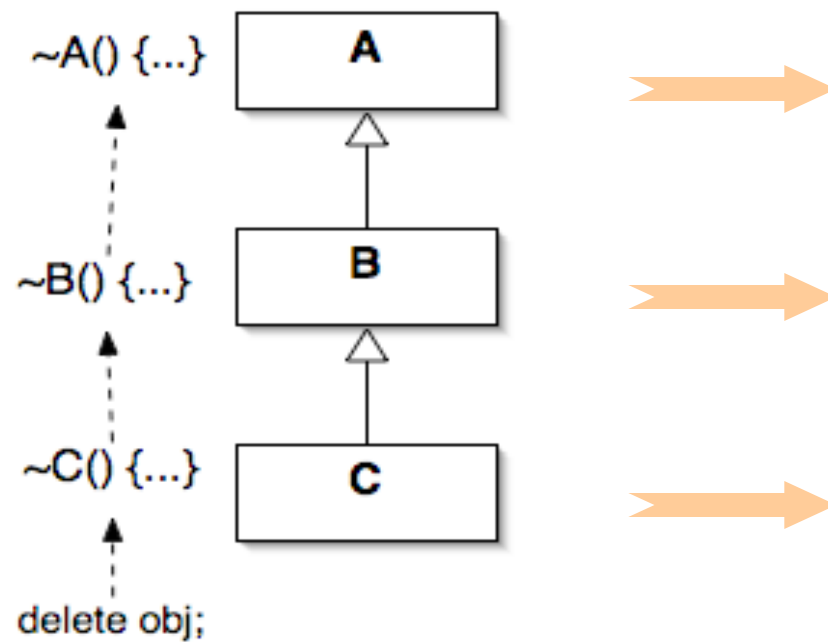
class Circulo extends Figura2D {
    private double radio;
    public Circulo() { radio=1.0; } //llamada implícita a Figura2D()
    public Circulo() { super(); radio=1.0; } //llamada explícita
    public Circulo(Color col, double r) { super(col); radio=r; }
    public Circulo(Circulo cir) { super(cir);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

- 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 destructor de la clase base.

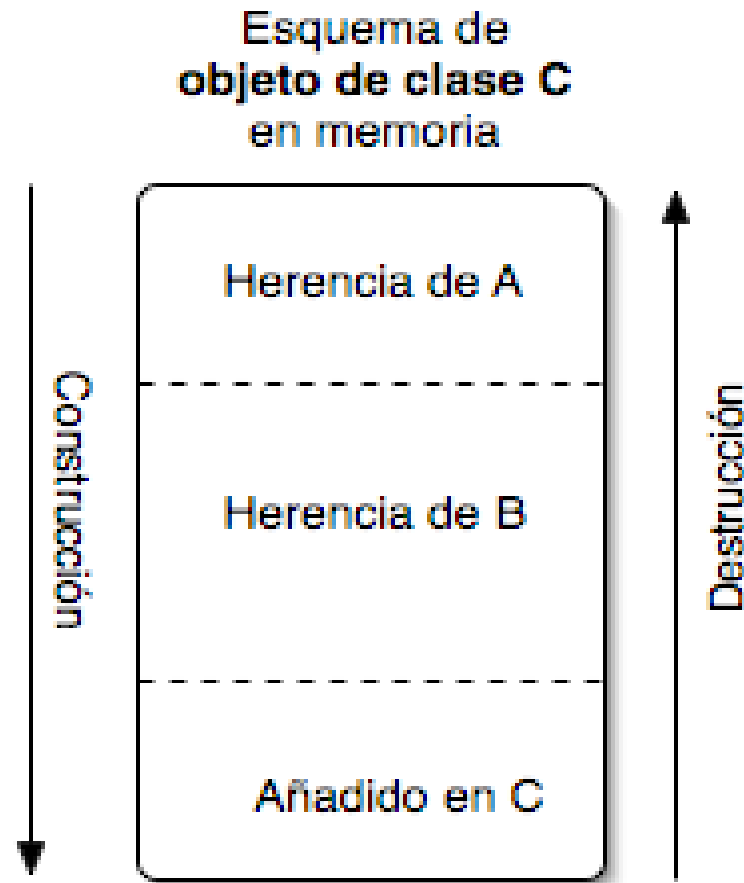


El destructor en herencia simple (C++)

Ejecución



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



- 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.

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,4ª ed.', Bruce Eckl)

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();  
        }  
    }  
}
```

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;  
    // ...  
}
```

Herencia Simple (base): 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.)

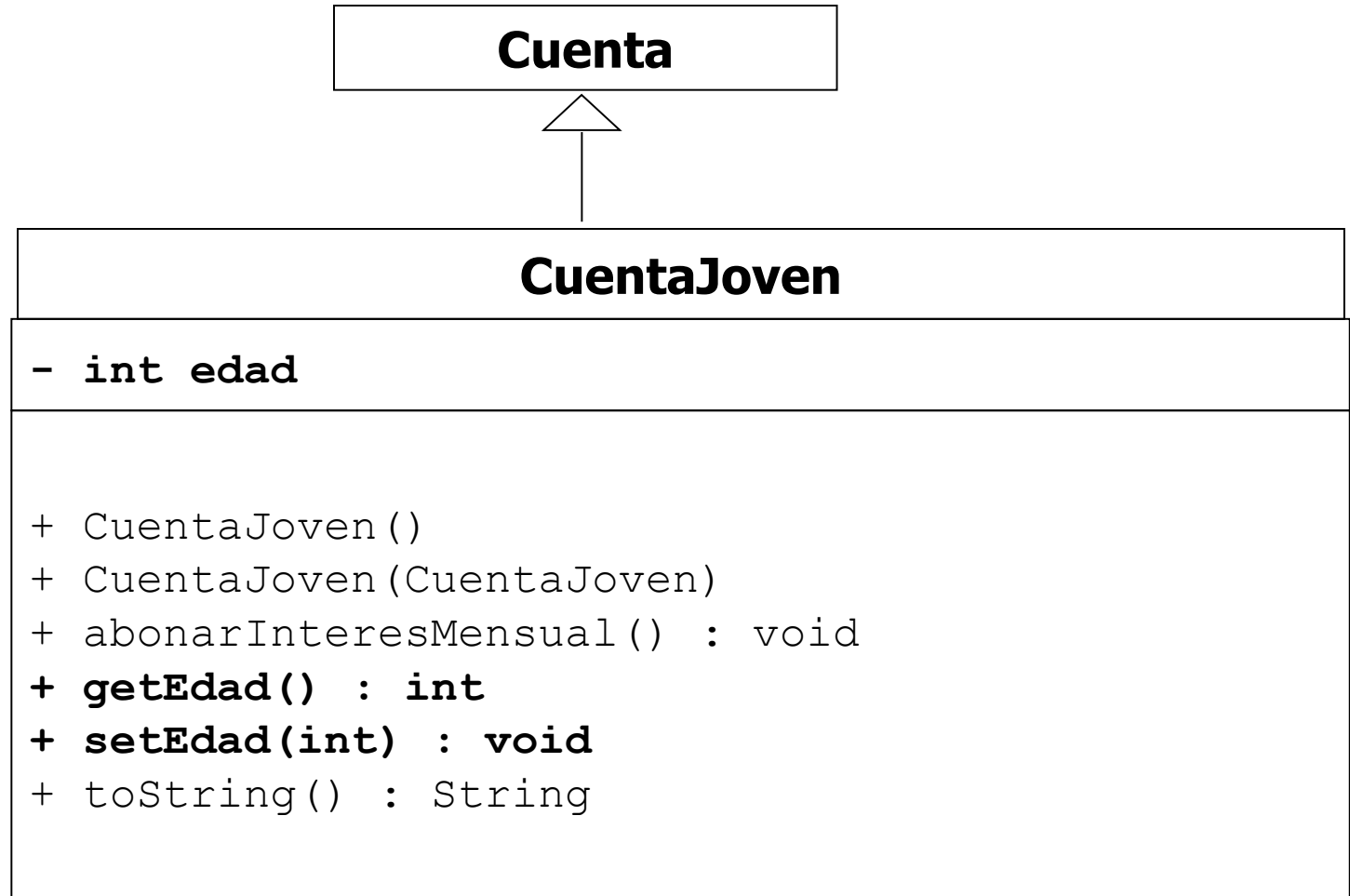
void abonarInteresMensual()

```
{ 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;
```

¿Hay que incrementar numCuentas?

```
    public CuentaJoven(String unNombre,int unaEdad,  
        double unSaldo, double unInteres)
```

```
    {
```

```
        super(unNombre,unSaldo,unInteres) ;
```

```
        edad=unaEdad;
```

```
    }
```

Refinamiento

```
    public CuentaJoven(CuentaJove& tcj)
```

```
    // llamada explícita a constructor de copia de Cuenta.
```

```
    {
```

```
        super(tcj) ;
```

```
        edad=tcj.edad) ;
```

```
    }
```

```
    ...
```

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));
```

```
}
```

Reemplazo

```
int getEdad() {return edad;}
```

```
void setEdad(int unaEdad) {edad=unaEdad;}
```

Métodos
añadidos

```
void toString() {
```

```
    String s = super.toString();
```

```
    s = s + "Edad:" + edad;
```

```
}
```

```
} //fin clase CuentaJoven
```

Método
Refinado

Convertir un objeto de tipo derivado a tipo base

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

```
c = (Cuenta)tcj; // explícito  
c = tcj; // implícito
```

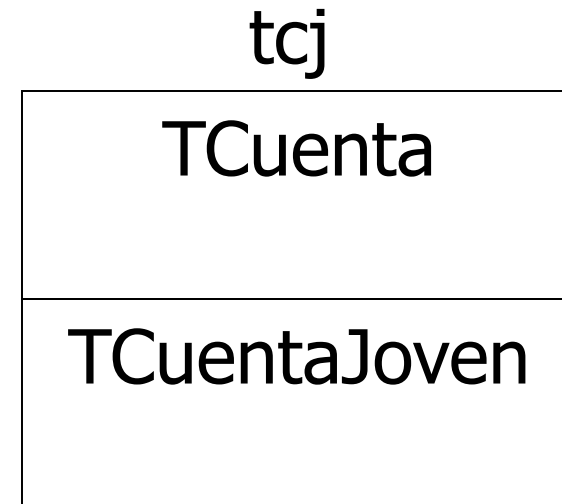
```
tcj.setEdad(18); //OK  
c.setEdad(18); // ¡ERROR!
```

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.

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

```
CuentaJoven tcj;
```

```
(TCuenta)tcj
```

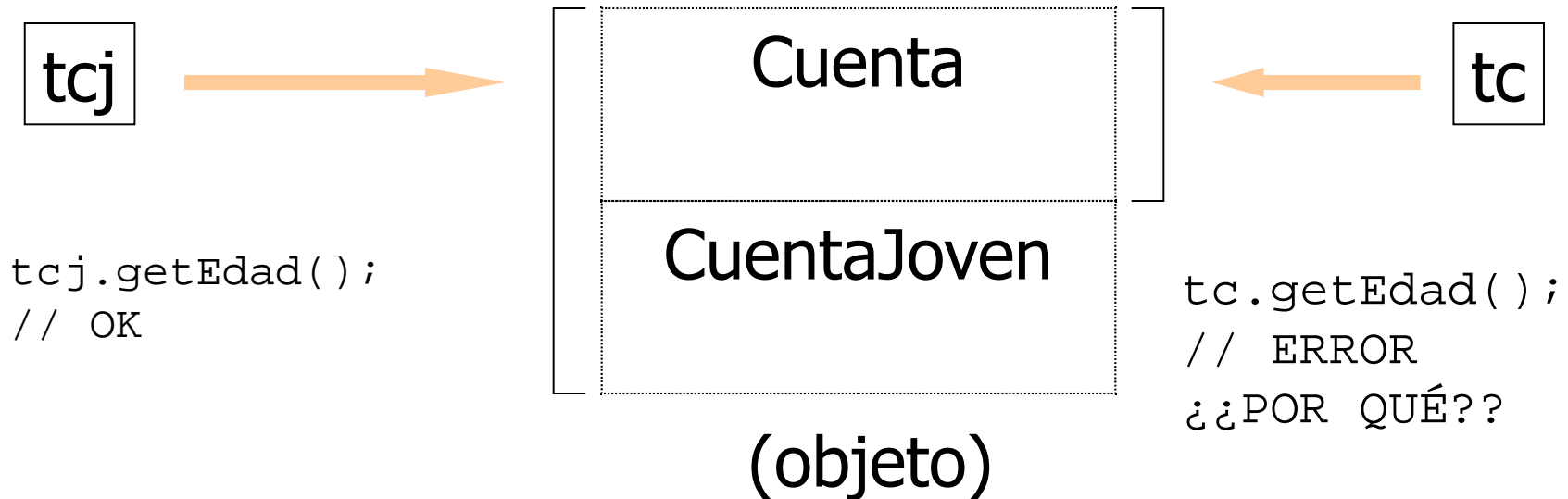


Herencia Simple (derivada): Upcasting



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

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



- 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.

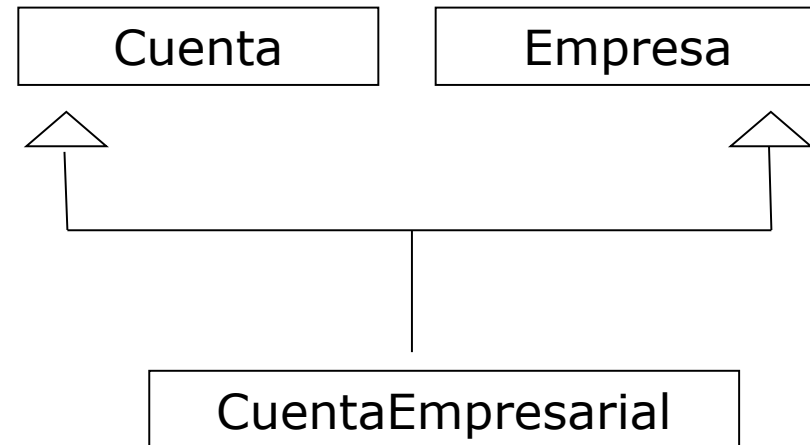
En esta sección tratamos la herencia múltiple de implementación.

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++

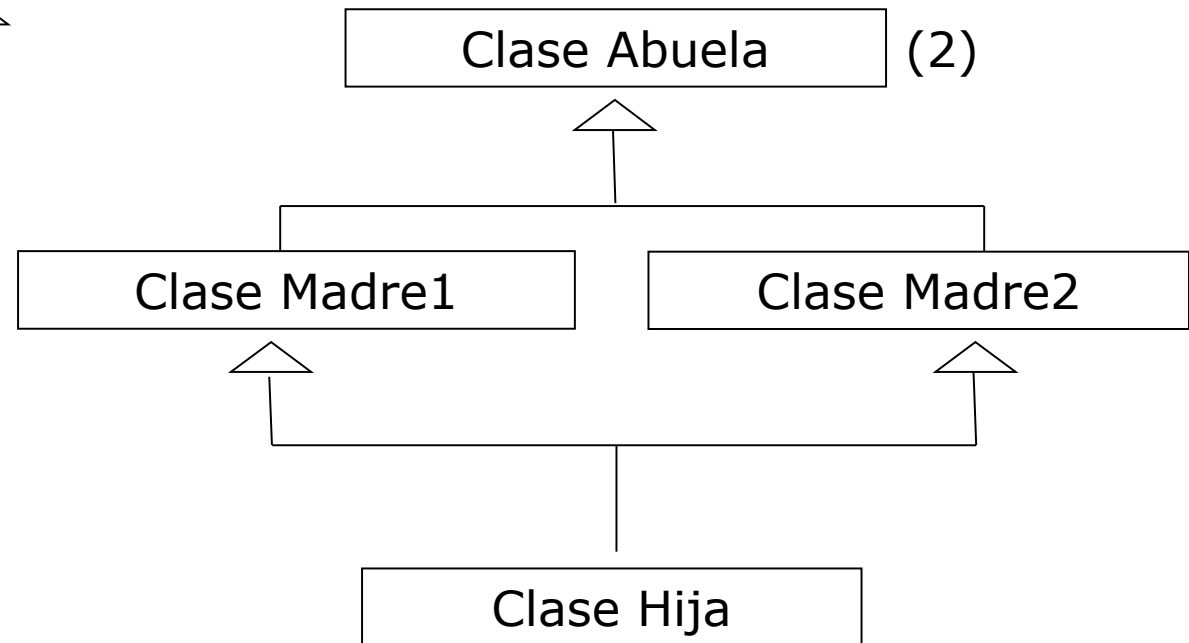
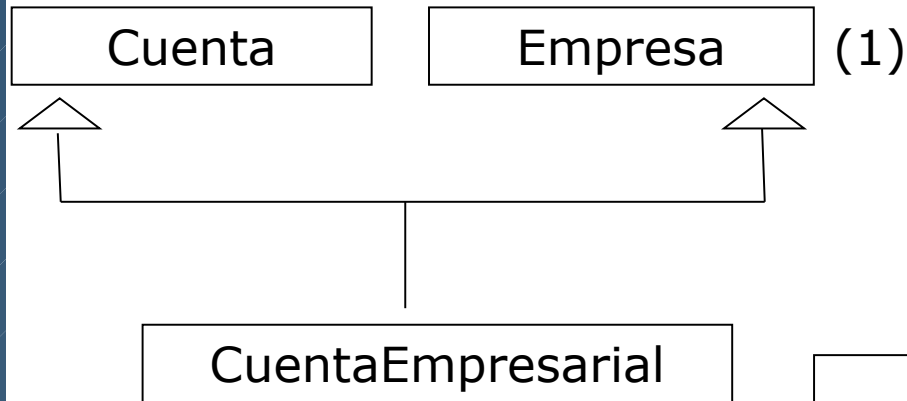


```
class CuentaEmpresarial
: public Cuenta, public Empresa {

    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

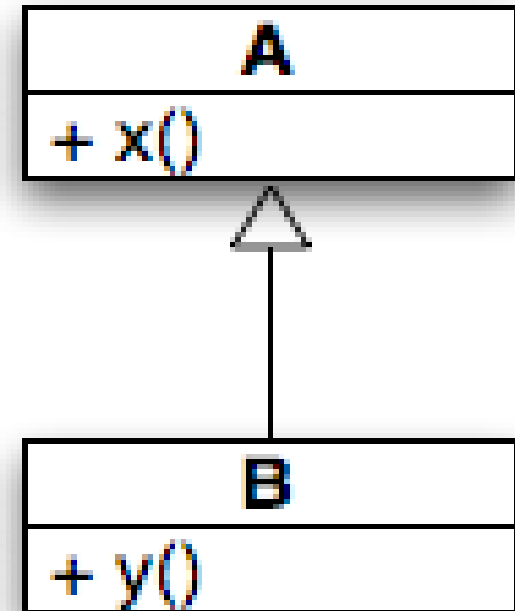
- La herencia de interfaz NO hereda código
- Sólo se hereda la interfaz (a veces con una implementación parcial o por defecto).
- Objetivos
 - Separar la interfaz de la implementación.
 - Garantizar la **sustitución**.

■ 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 debilmente 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.

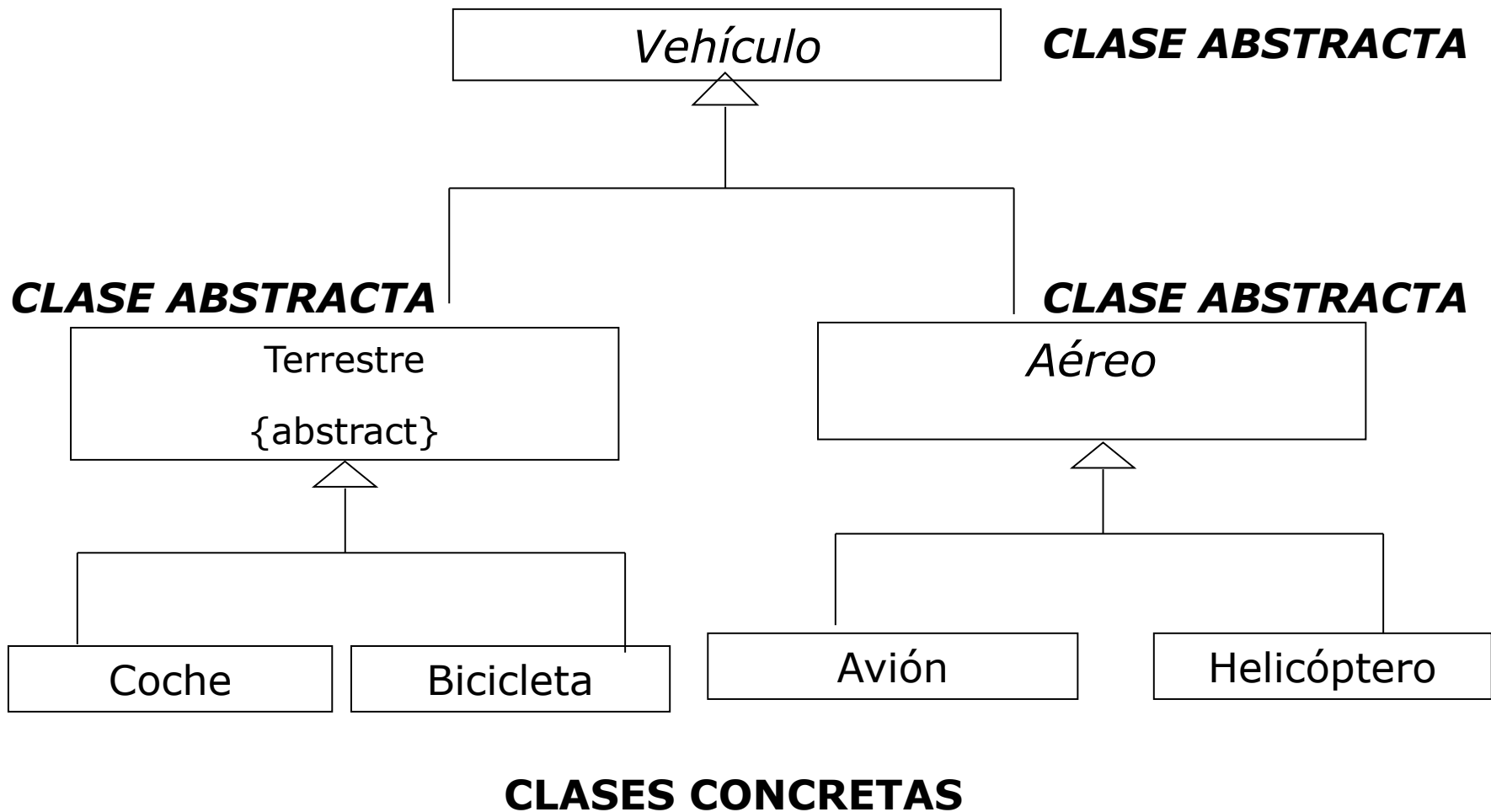
HERENCIA DE INTERFAZ

- **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() {...};
    ...
}
```

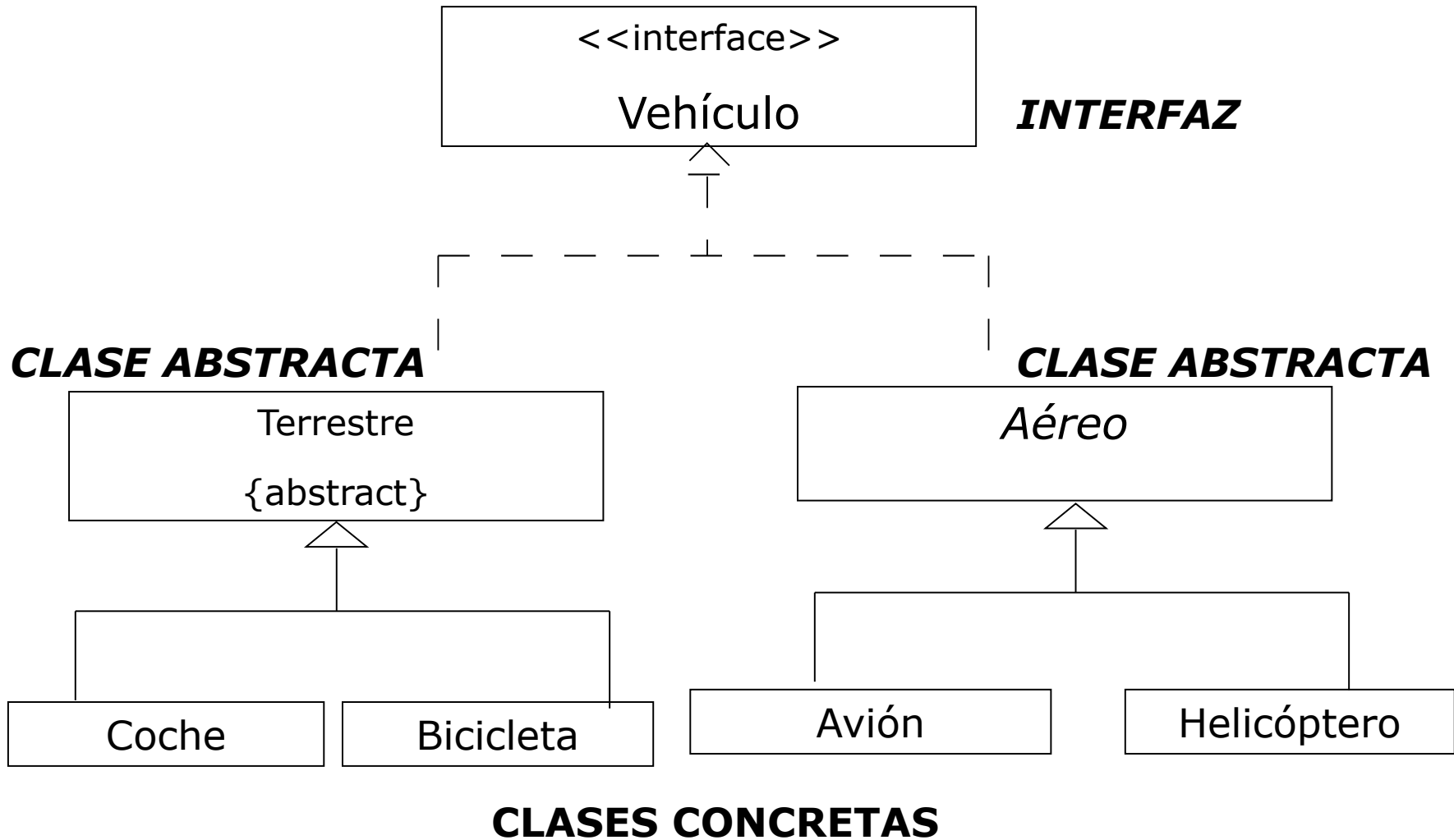
HERENCIA DE INTERFAZ

- **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

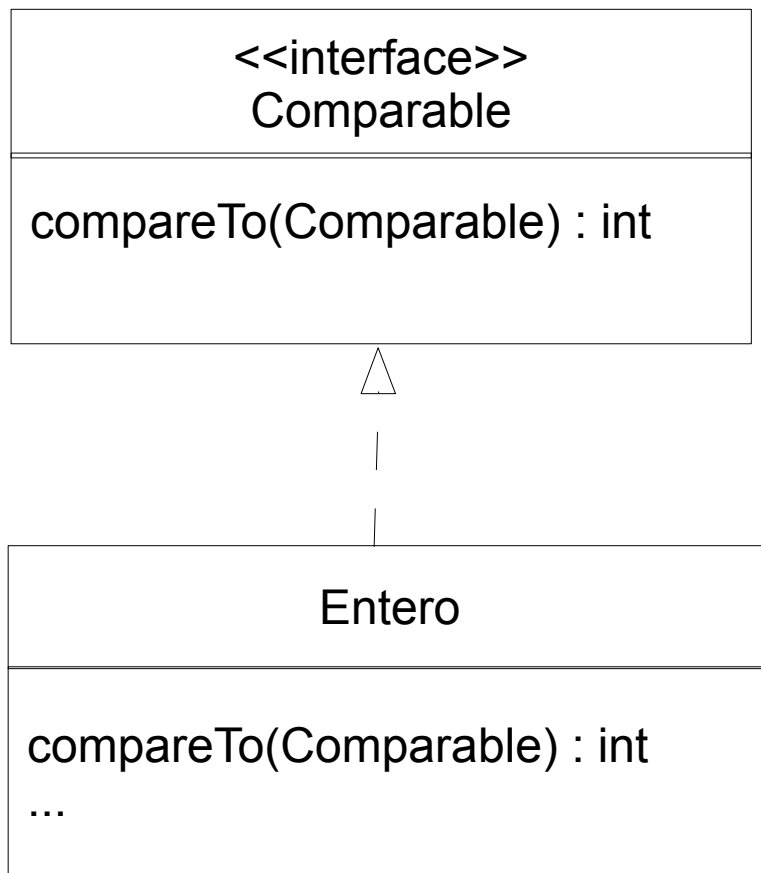
- **Interfaces en C++:** herencia pública de clases abstractas

```
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]; }
}
```

```
// Codigo 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();  
}
```

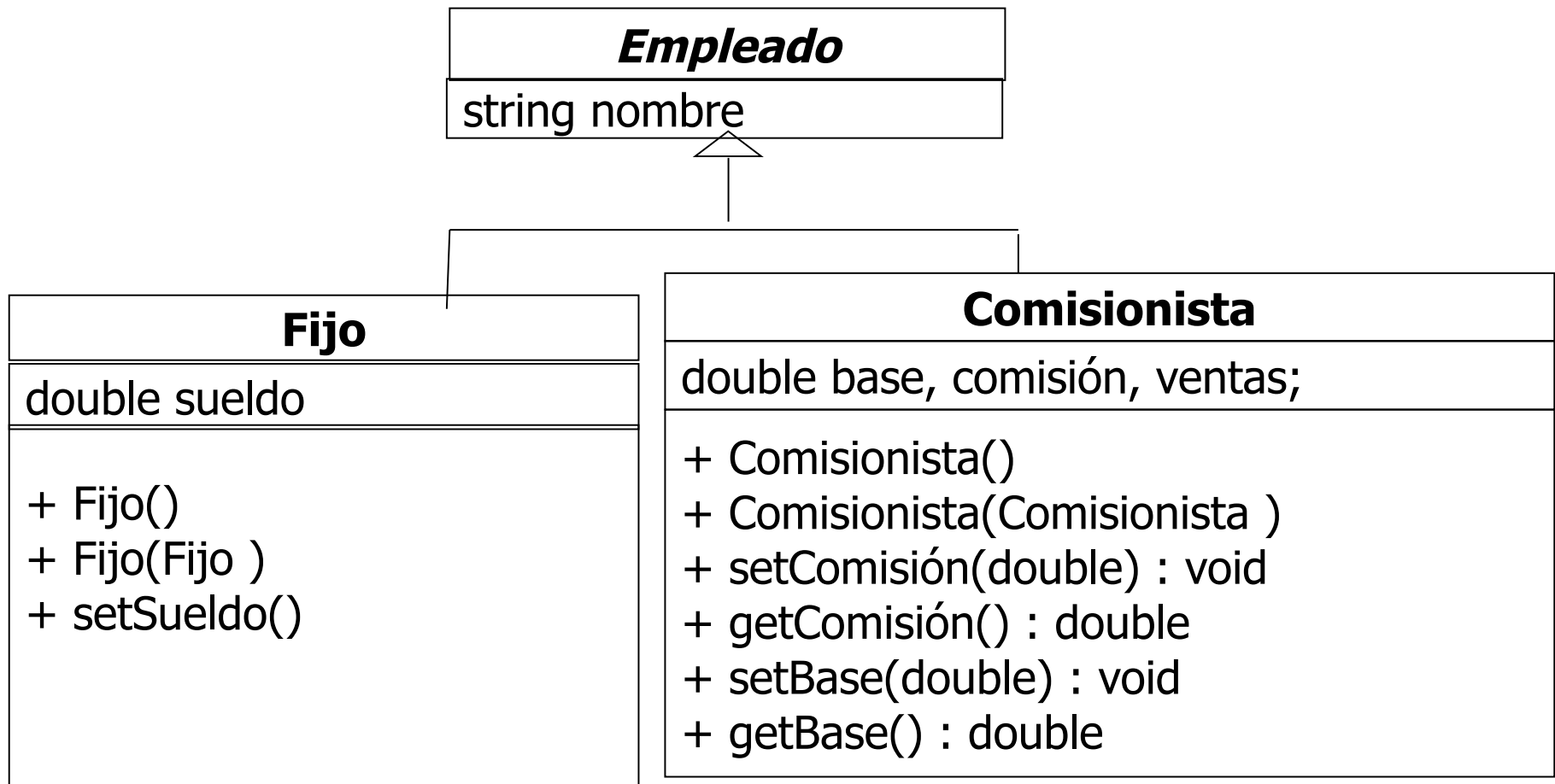
```
class ActionCharacter {  
    public void fight() {}  
}
```

```
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, 4ª 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

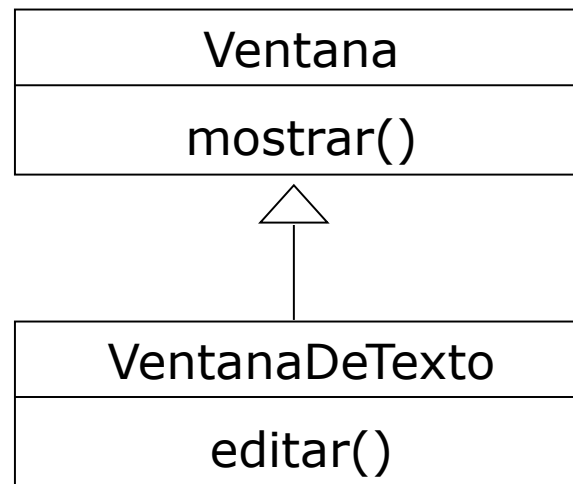
Uso seguro

- 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

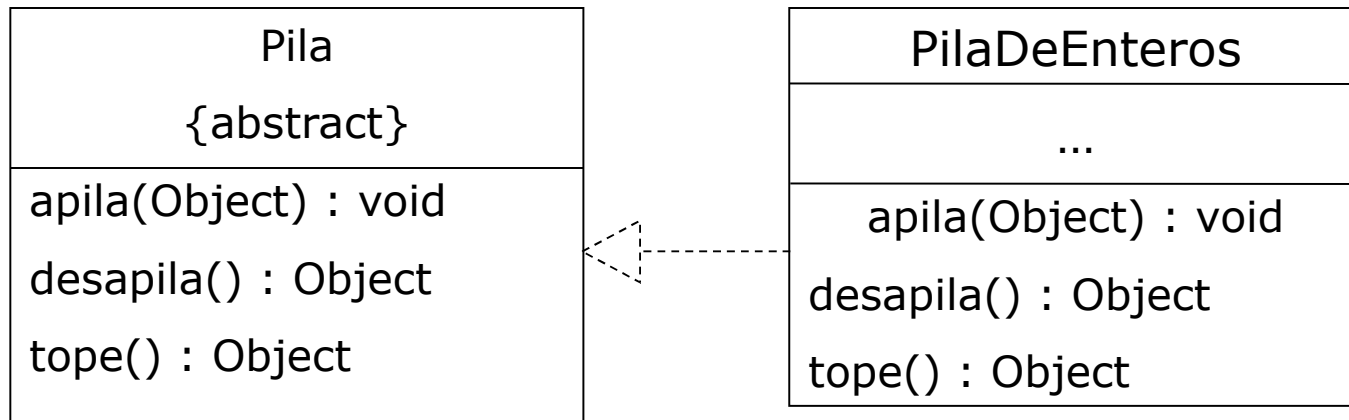
■ **Especialización**

- 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)

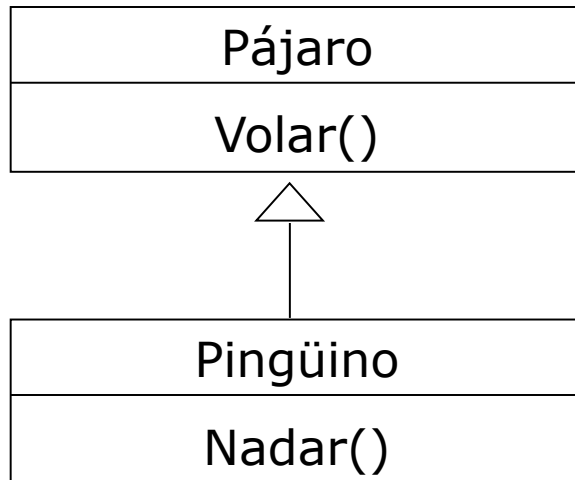


■ Especificación

- 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)

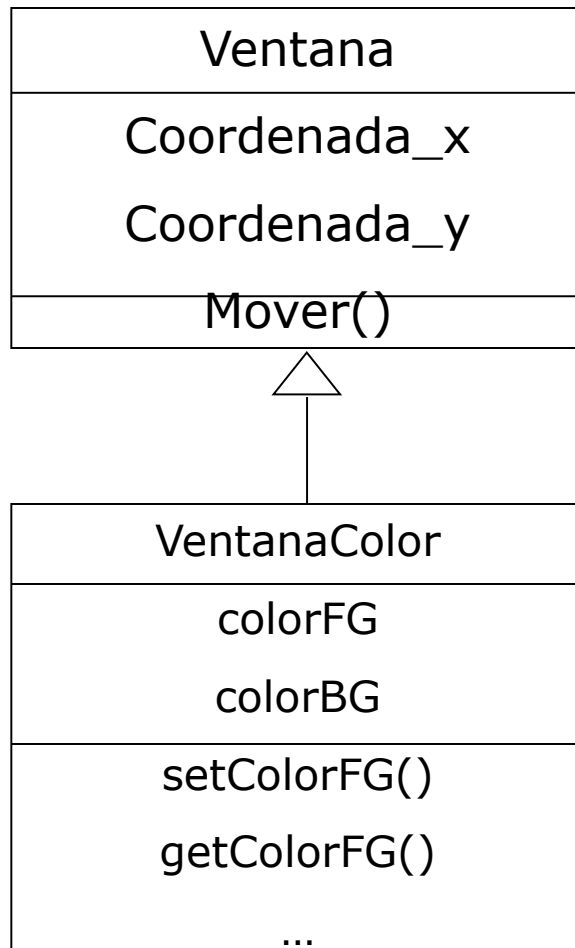


No todo lo de la clase base sirve a la derivada.

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

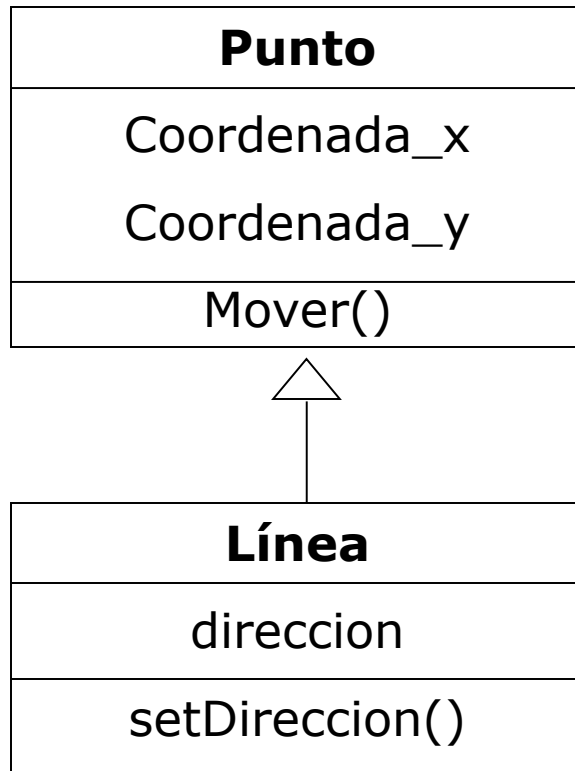
No se cumple el principio de sustitución
(un pingüino no puede volar)

■ 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)**

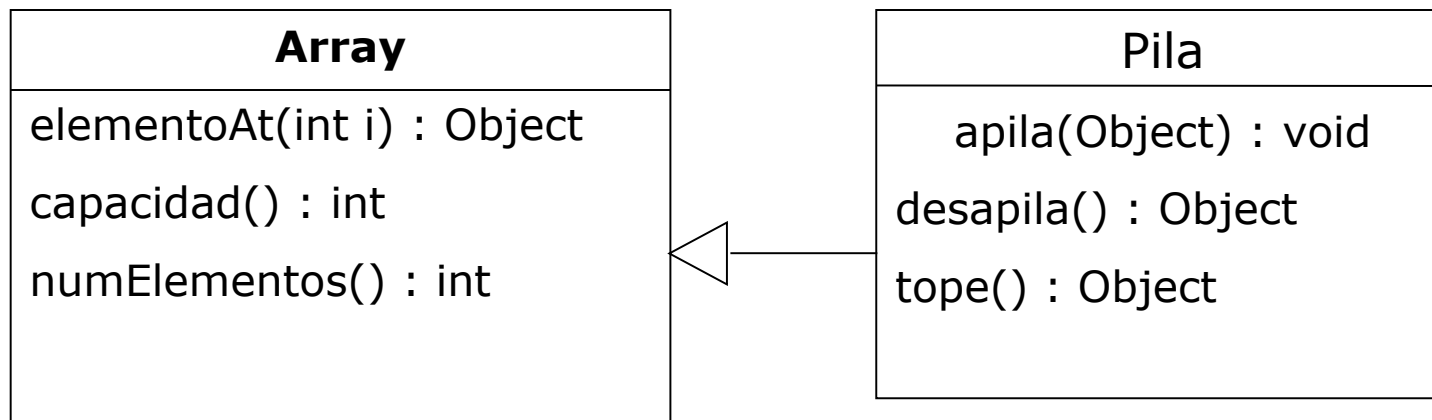


La implementación se parece pero semánticamente los conceptos no están relacionados jerárquicamente (test "es-un").

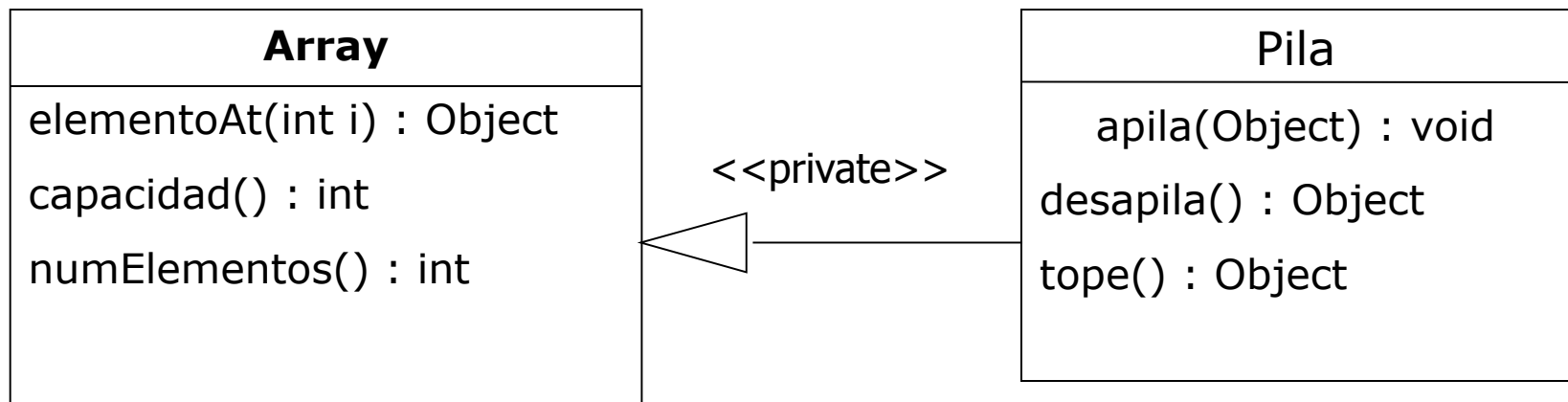
INCORRECTA!!!!

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

- 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



- 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

[BUDD] 8.8

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

[BUDD] 8.9

HERENCIA

Elección de técnica de reuso

- 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;  
}
```

- 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 metodos 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).

- 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.

- 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 4ª 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++)