

Programación orientada a objetos

TEMA 3

HERENCIA

Cristina Cachero, Pedro J. Ponce de León

versión 3

(Curso 10/11)

(3 sesiones)



Tema 3. HERENCIA

Objetivos

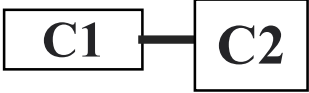


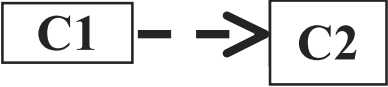
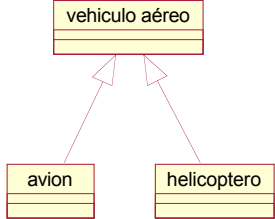


- Entender el mecanismo de abstracción de la herencia.
- Distinguir entre los diferentes tipos de herencia
- Saber implementar jerarquías de herencia en C++
- Saber discernir entre jerarquías de herencia seguras (bien definidas) e inseguras.
- Reutilización de código: Ser capaz de decidir cuándo usar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

Herencia

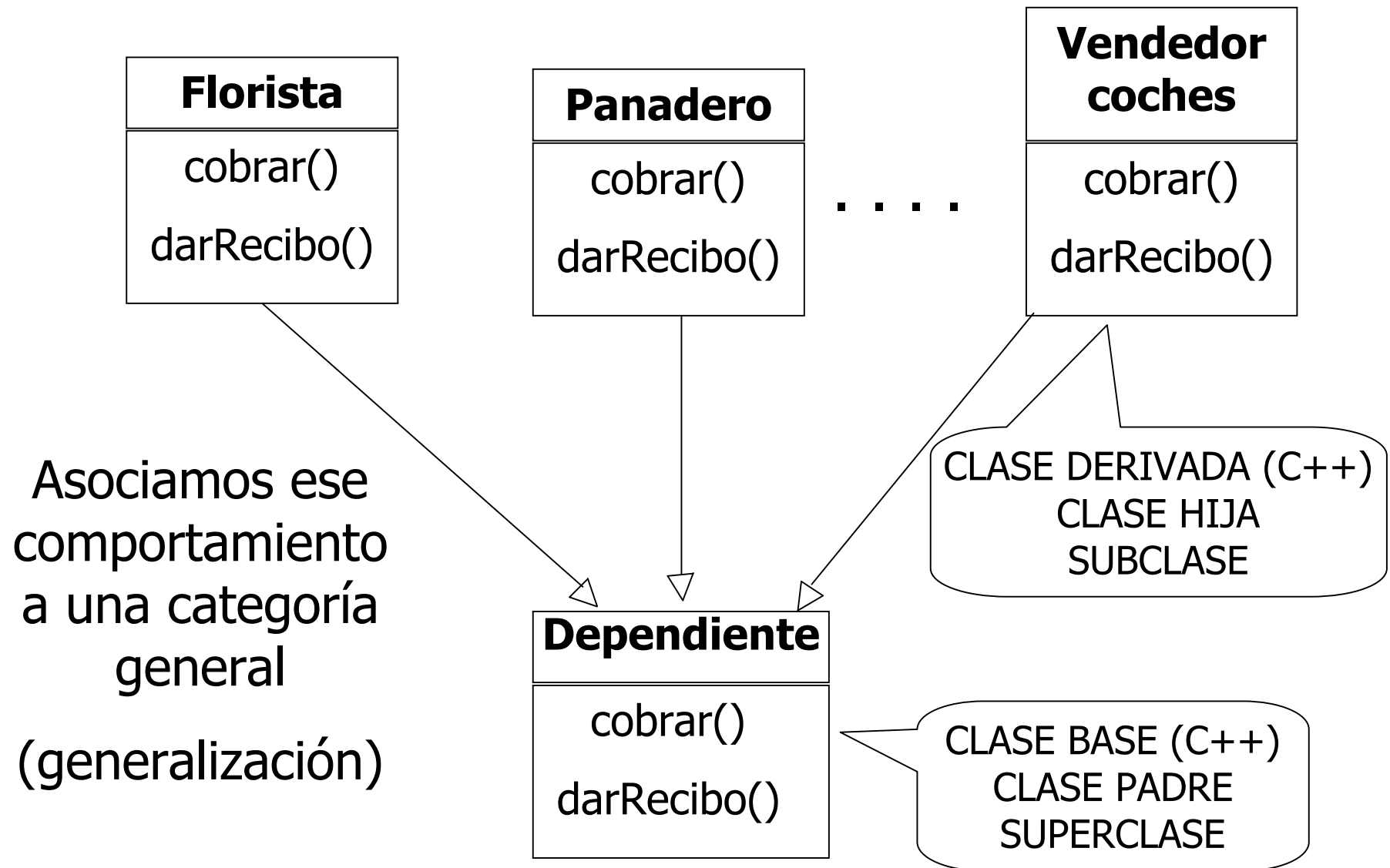
Del tema anterior...



	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 tipos de datos (abstracciones) por variedad, acercando un poco más el mundo de la programación al 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 interfaz de la subclase debe incluir todas las propiedades públicas y privadas 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

HERENCIA

Definición



- 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

HERENCIA

Test "ES-UN"



- 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: Esto ocurre cuando 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**

Tipos de Herencia



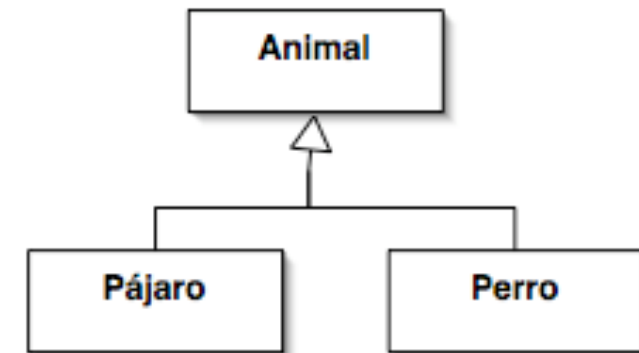
- Simple/Múltiple
- De implementación/de interfaz

Tipos de Herencia

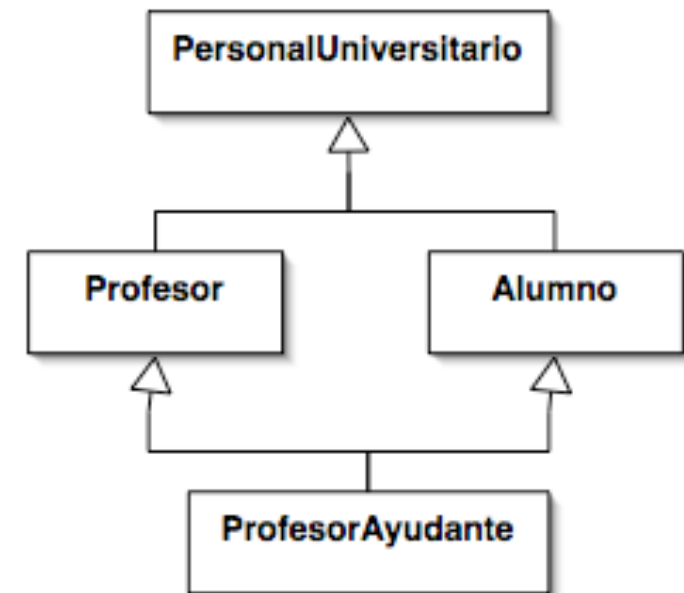


- Simple/Múltiple

- **Simple:** única clase base



- **Múltiple:** Más de una clase base





- De implementación/de interfaz
 - **De implementación:** La implementación de los métodos es heredada. Puede sobrescribirse 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.
- C++ no soporta la herencia solapada (tipado fuerte)

■ **Completa/Incompleta**

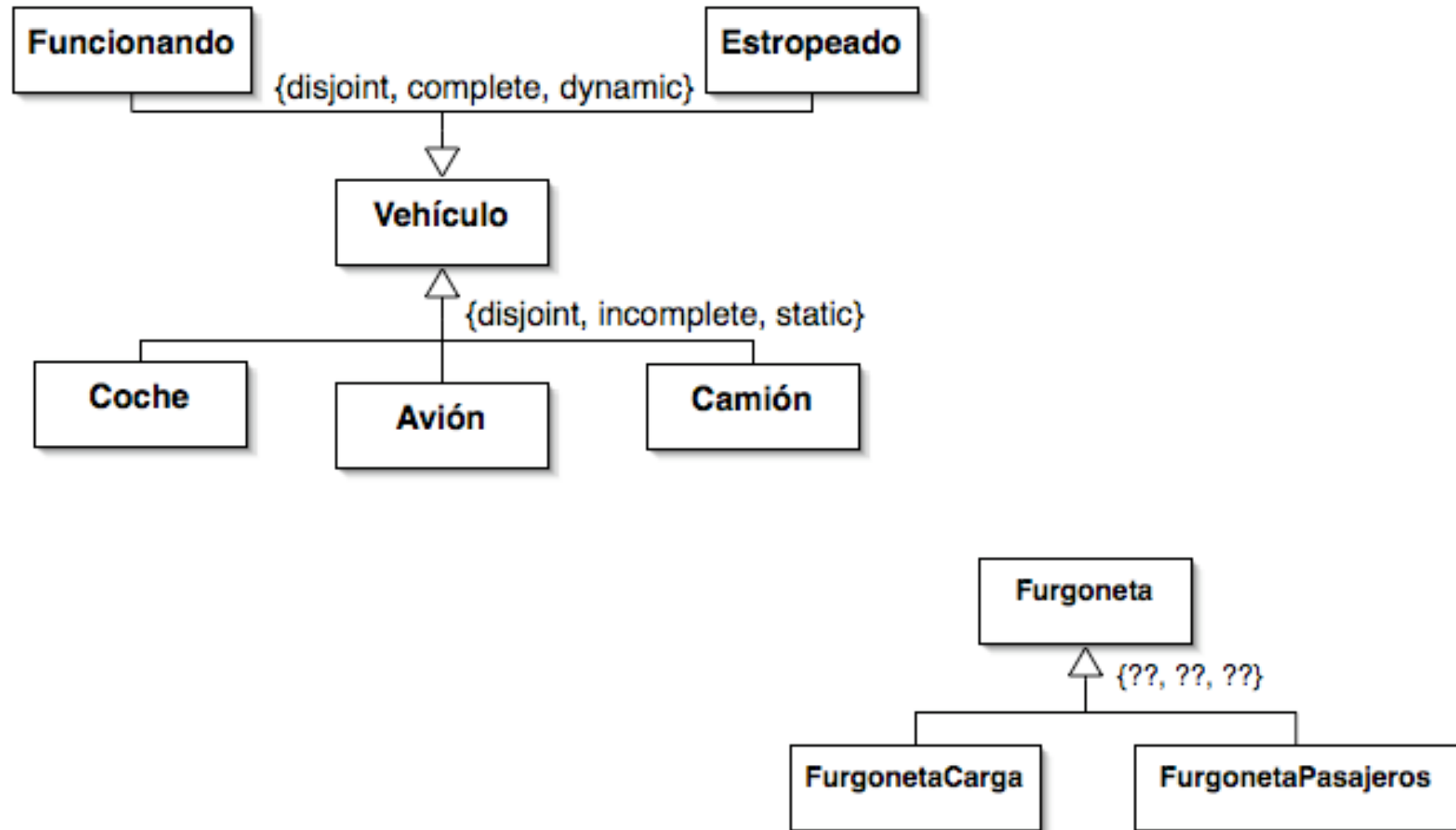
- Determina si todas las instancias de la clase padre son *a la vez* instancias de alguna de las clases hijas (completa) o, por el contrario, hay objetos de la clase padre que no pertenecen a ninguna subcategoría de las reflejadas por las clases hijas (incompleta).

■ **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.
- C++ no soporta la herencia dinámica (tipado fuerte)

Herencia

Caracterización: ejemplos



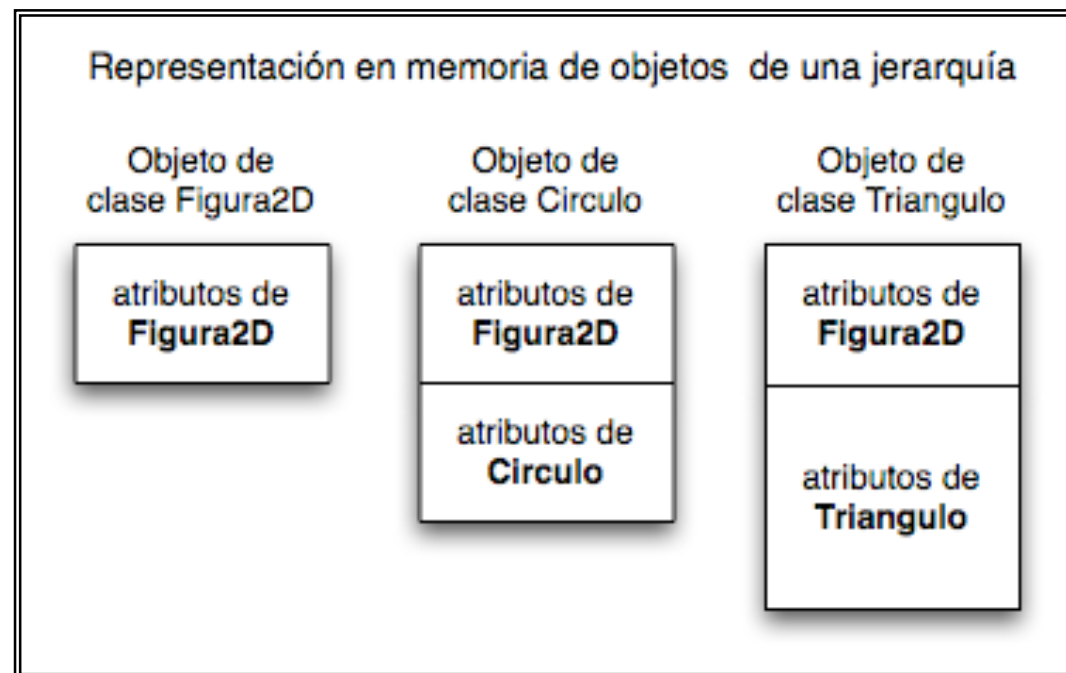
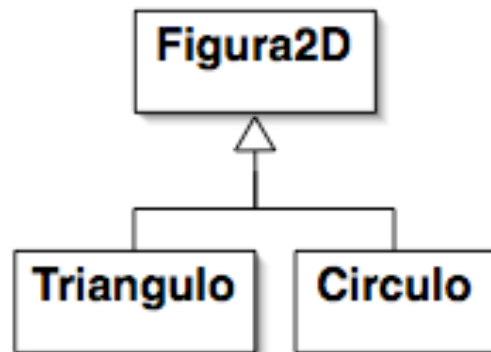
HERENCIA DE IMPLEMENTACIÓN

Herencia Simple

Herencia Simple en C++



- 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)



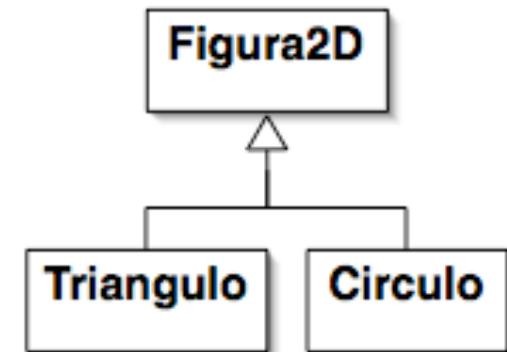
Herencia Simple en C++



```
class Figura2D {  
    public:  
        ...  
        void setColor(Color c);  
        Color getColor() const;  
    private:  
        Color colorRelleno;  
    ... };
```

```
class Circulo : Figura2D {  
    ...  
    public:  
        void vaciarCirculo() {  
            colorRelleno=NINGUNO;  
            // ¡ERROR! colorRelleno es privado  
            setColor(NINGUNO); // OK  
        }  
};
```

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



```
int main() {  
    Circulo c;  
    c.setColor(AZUL);  
    c.getColor();  
    c.vaciarCirculo();  
    ...  
}
```



- **Ámbito de visibilidad `protected`**
 - Los datos/funciones miembros *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 : Figura2D {  
    public:  
    void vaciarCirculo() {  
        colorRelleno=NINGUNO; //OK, protected  
    }  
    ...  
};
```

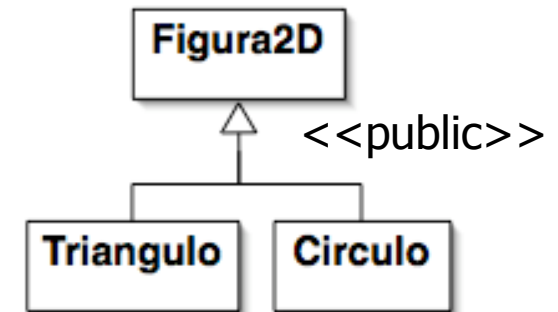
```
int main () {  
    Circulo c;  
    c.colorRelleno=NINGUNO;  
    // ¡ERROR! colorRelleno  
    // es privado aquí  
}
```

Tipos de Herencia Simple (en C++)



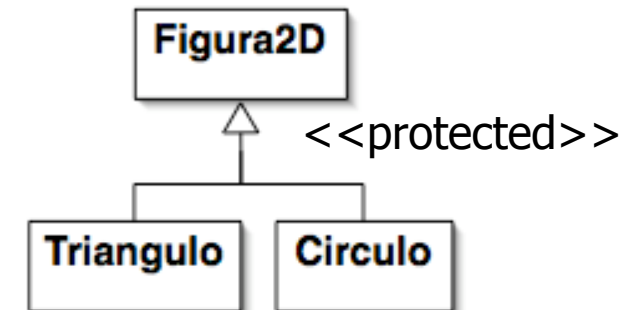
- Herencia Pública (por defecto)

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



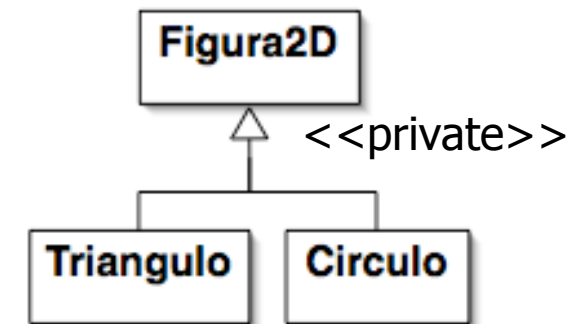
- Herencia Protegida

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



- Herencia Privada

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



Tipos de Herencia Simple

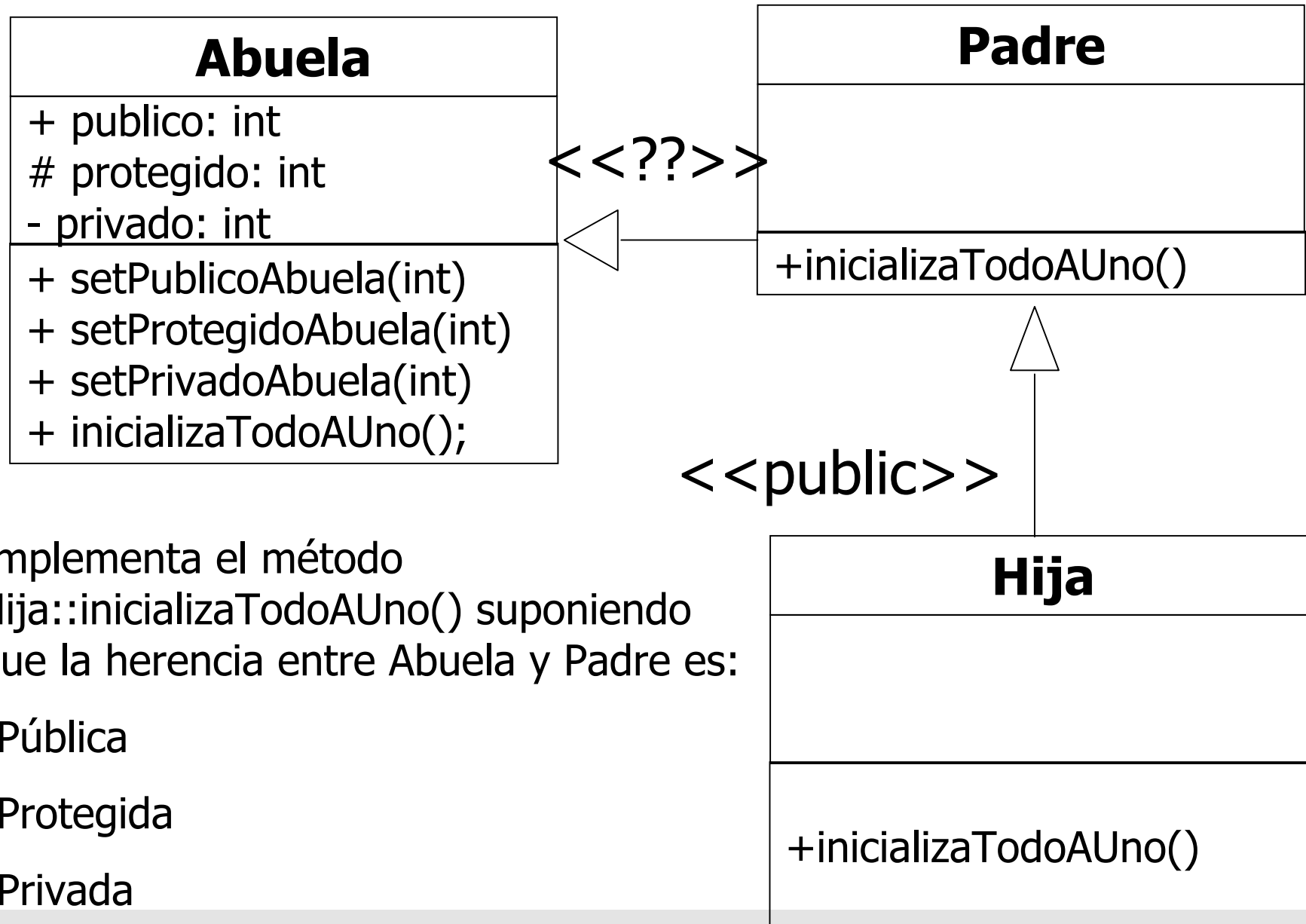


Ámbito Herencia Visibilidad en clase base	CD (*) H. Pública	CD H. Protegida	CD H. privada
Private	No direct. accesible	No direct. accesible	No direct. accesible
Protected	Protected	Protected	Private
Public	Public	Protected	Private

(*) CD: Clase derivada

Tipos Herencia Simple

Ejercicio



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

- Pública
- Protegida
- Privada

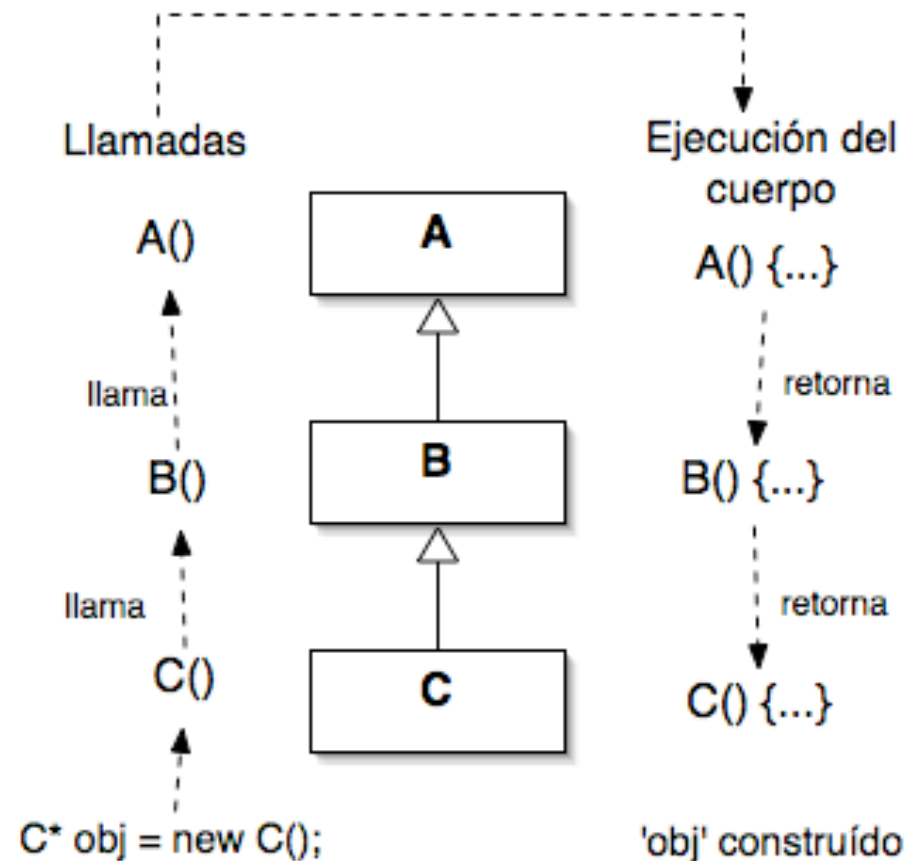


- En la clase derivada se puede:
 - Añadir nuevos métodos/atributos propios de la clase derivada
 - Modificar los métodos heredados de la clase base
 - **REFINAMIENTO:** se añade comportamiento nuevo antes y/o después del comportamiento heredado. (*Simula, Beta*)
(se puede simular en C++, Java)
 - C++,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

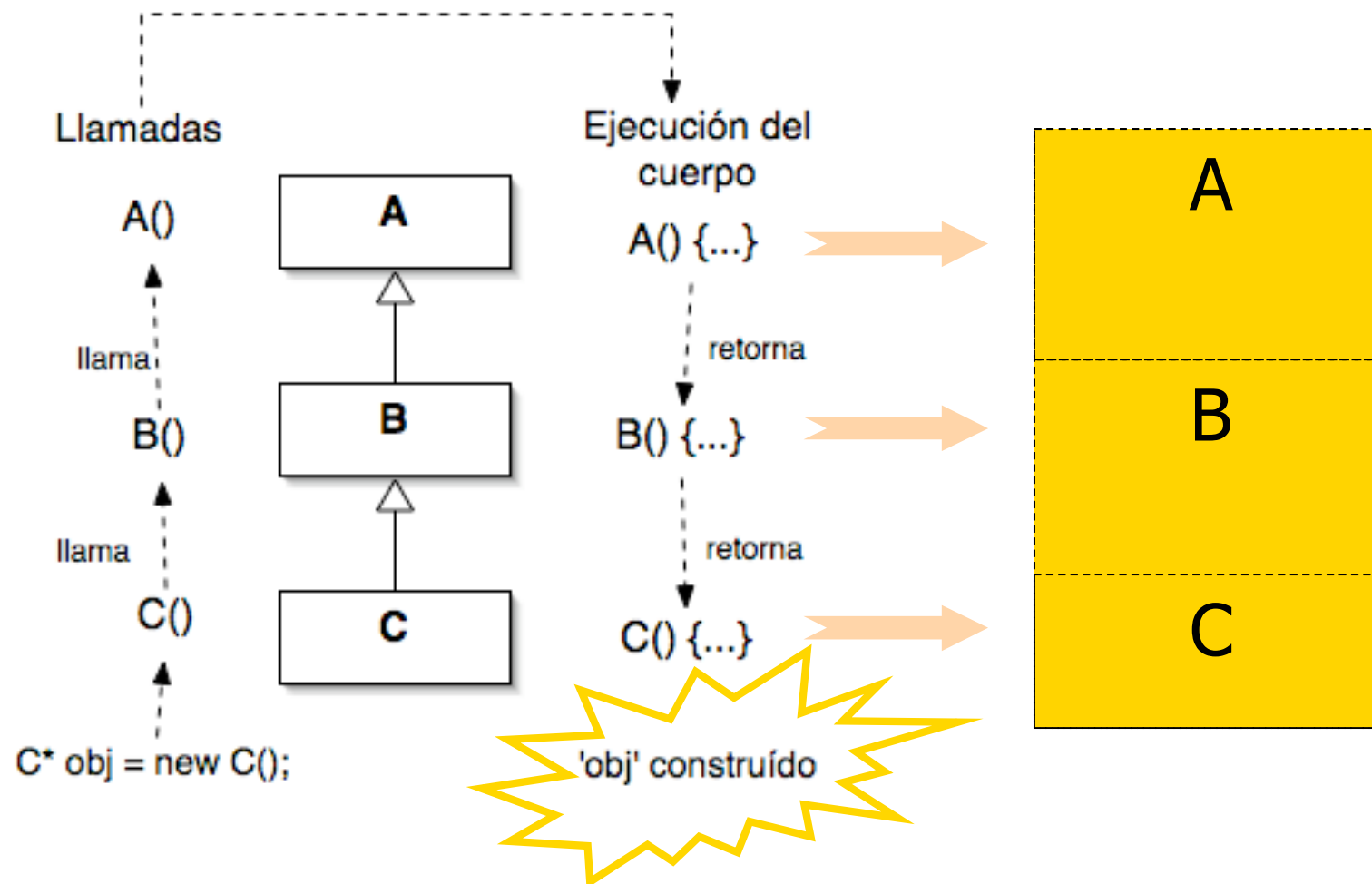
El constructor en herencia simple



- 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 {
    Color colorRelleno;
public:
    Figura2D() : colorRelleno(NINGUNO) {}
    Figura2D(Color c) : colorRelleno(c) {}
    Figura2D(const Figura2D& f)
        : colorRelleno(f.colorRelleno) {}
    ...};

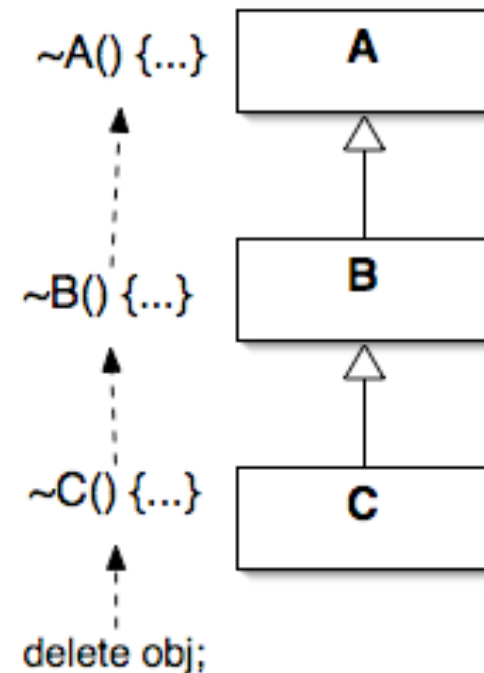
class Circulo : Figura2D {
    double radio;
public:
    Circulo() : radio(1.0) {} //llamada implícita a Figura2D()
    Circulo(Color col, double r) : Figura2D(col), radio(r) {}
    Circulo(const Circulo& cir)
        : Figura2D(cir), radio(cir.radio) {}
    ...};
```

El destructor en herencia simple



- 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.

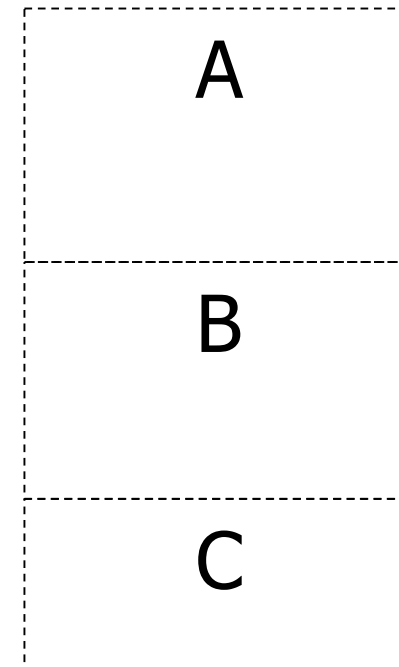
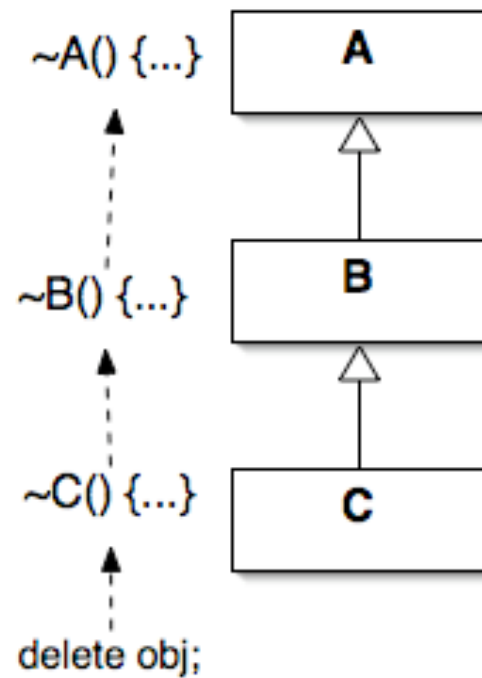
Ejecución



El destructor en herencia simple



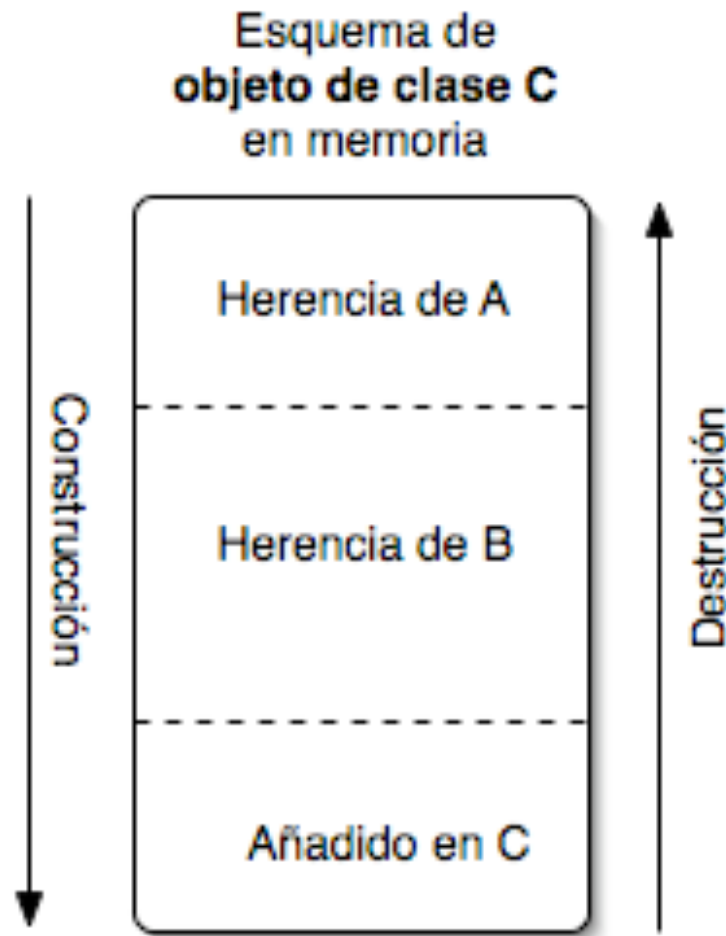
Ejecución



Construcción/destrucción en resumen...



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



Ejemplo Clase Base



TCuenta

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

```
+ TCuenta()  
+ TCuenta(const TCuenta &)  
+ operator=(const Tcuenta &) : TCuenta&  
+ ~TCuenta()  
+ getTitular() : string  
+ getSaldo() : double  
+ getInteres() : double  
+ setSaldo(double) : void  
+ setInteres(double) : void  
+ abonarInteresMensual() : void  
+ mostrar() : void
```

Herencia Simple (base): TCuenta



```
class TCuenta{
public:
    TCuenta(string t, double s=0.0, double i=0.0)
        : titular(t), saldo(s), interes(i)
        { numCuentas++; }
    ...
protected:
    string titular;
    double saldo;
    double interes;
    static int numCuentas;
};
```


Herencia Simple (base): TCuenta (II)



```
TCuenta::TCuenta(const TCuenta& tc)  
: titular(tc.titular), saldo(tc.saldo),  
  interes(tc.interes)  
{ numCuentas++; }
```

```
TCuenta& TCuenta::operator=(const TCuenta& tc) {  
    if (this!=&tc) {  
        titular = tc.titular; saldo=tc.saldo;  
        interes=tc.interes;  
    }  
    return*this;  
}
```

```
TCuenta::~~TCuenta() { numCuentas--; }
```

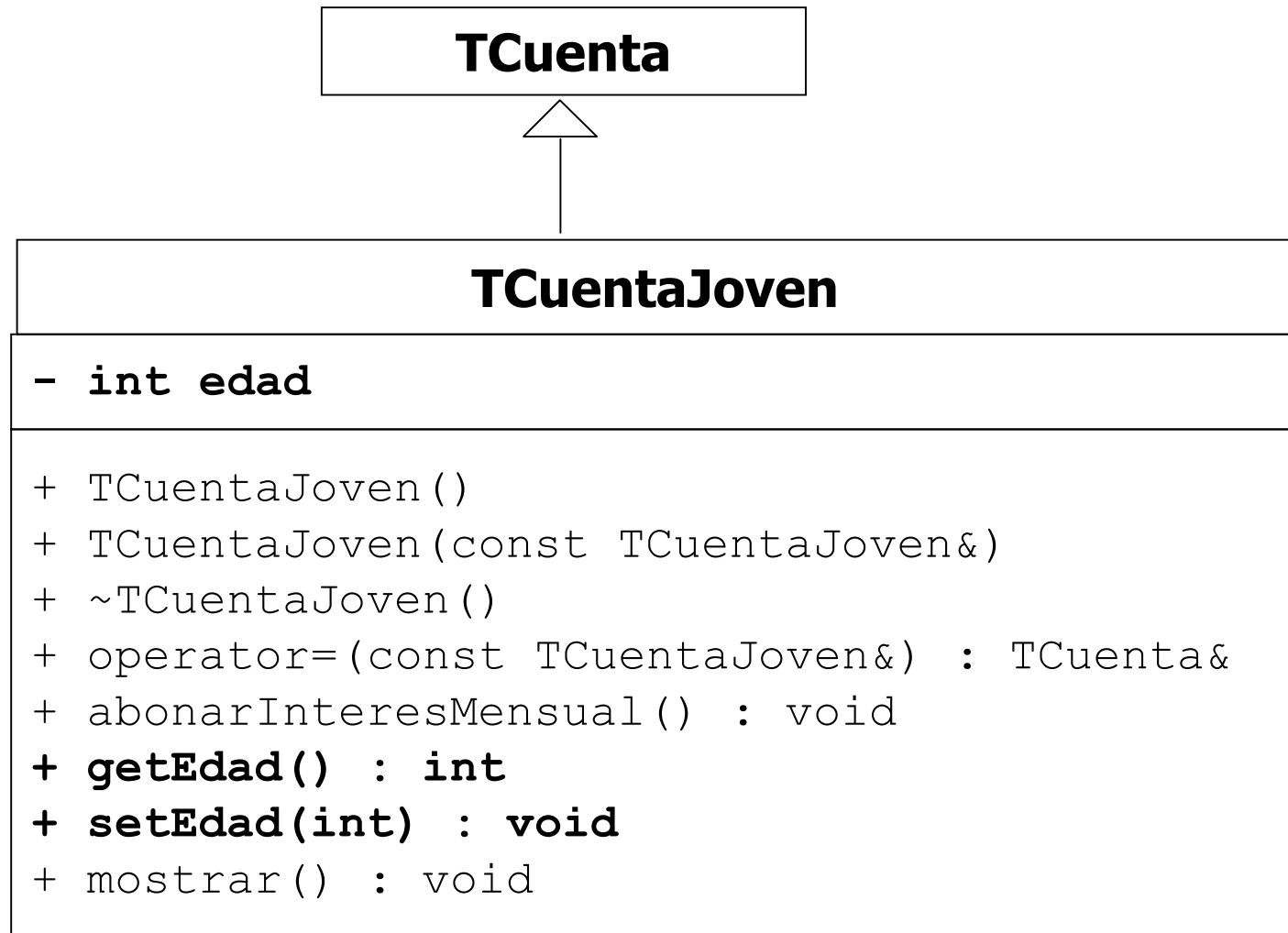
Herencia Simple (base): TCuenta (III)



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

```
void TCuenta::mostrar ()  
{  
    cout << "NumCuentas=" << TCuenta::numCuentas << endl;  
    cout << "Titular=" << unaCuenta.titular << endl;  
    cout << "Saldo=" << unaCuenta.saldo << endl;  
    cout << "Interes=" << unaCuenta.interes << endl;  
}  
}
```

Ejemplo clase derivada



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

Herencia Simple (derivada): TCuentaJoven (I)



```
class TCuentaJoven: public TCuenta {  
    private:  
        int edad;  
    public:  
        TCuentaJoven(string unNombre,int unaEdad,  
            double unSaldo=0.0, double unInteres=0.0)  
            : TCuenta(unNombre,unSaldo,unInteres) , edad(unaEdad)  
            { }
```

¿Hay que incrementar
numCuentas?

```
    TCuentaJoven(const TCuentaJoven& tcj)  
    // llamada explícita a constructor de copia de TCuenta.  
        : TCuenta(tcj) , edad(tcj.edad) { }
```

```
    ~TCuenta() { edad=0; }
```

```
    TCuenta& operator=(const TCuentaJoven& tcj) {  
        if (this!=&tcj) {  
            TCuenta::operator=(tcj) ;  
            edad = tcj.edad;  
        }  
        return *this;  
    }
```

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));  
    }  
}
```

Reemplazo

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

Métodos
añadidos

```
void mostrar() {  
    TCuenta::mostrar();  
    cout<<"Edad:"<<edad<<endl;  
}
```

Método
Refinado

```
}; //fin clase TCuentaJoven
```

Herencia Simple

Upcasting



Convertir un objeto de tipo derivado a tipo base

```
TCuentaJoven tcj;
```

```
cout << (TCuenta)tcj << endl;
```

```
cout << TCuenta(tcj) << endl;
```

```
TCuenta tc;
```

```
tc = tcj; // upcasting implícito
```

```
cout << tc << endl;
```

Herencia Simple (derivada): Upcasting

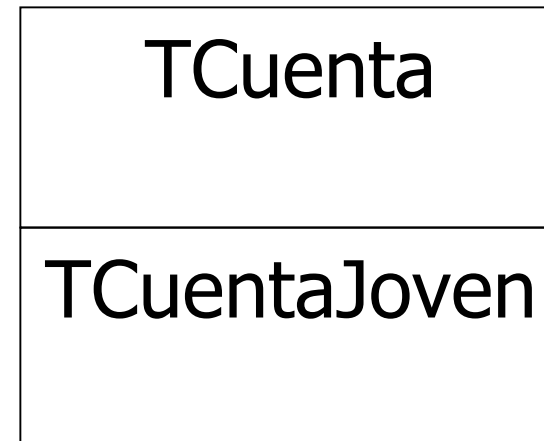


Cuando se convierten objetos en C++,
se hace **object slicing**

```
TCuentaJoven tcj;
```

```
(TCuenta)tcj
```

tcj

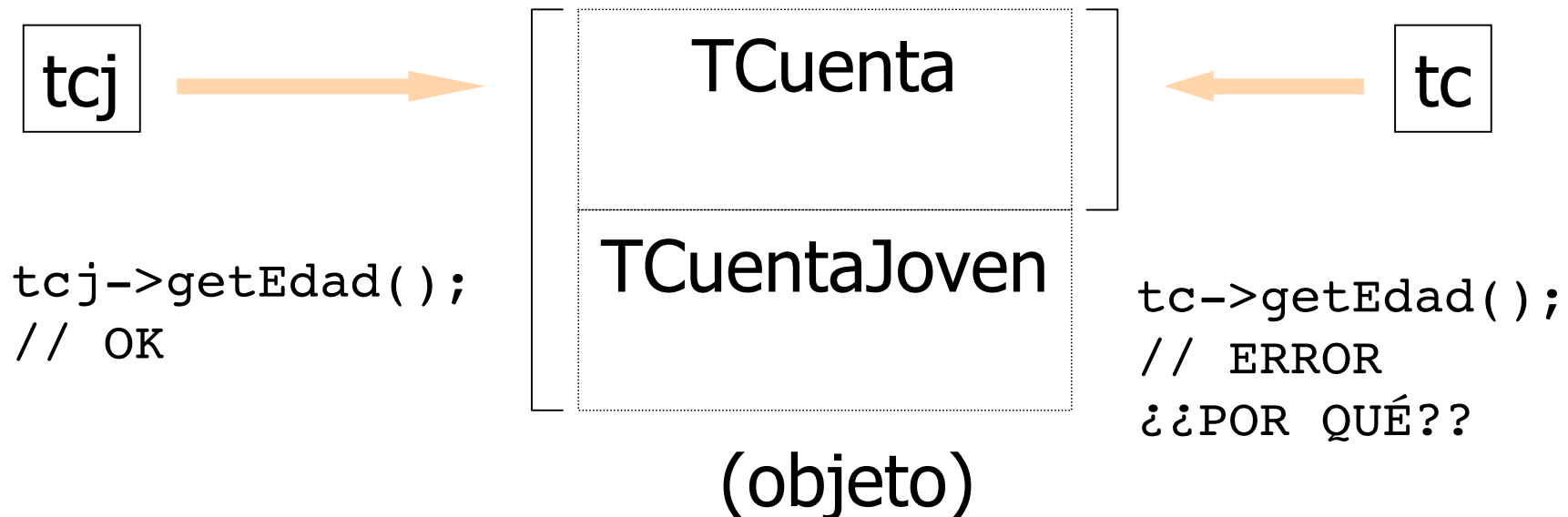


Herencia Simple (derivada): Upcasting



Con punteros o referencias en C++,
NO hay **object slicing**

```
TCuentaJoven* tcj = new TCuentaJoven;  
TCuenta* tc = tcj; // upcasting
```



Particularidades Herencia en C++



- En las jerarquías de herencia hay un refinamiento implícito de:
 - Constructor por defecto
 - Destructor
- El constructor de copia se refina explícitamente.
- El operador de asignación se reemplaza.
- De ahí que la forma canónica de la clase implique siempre definir estas cuatro funciones miembro.
- Las propiedades de clase definidas en la clase base también son compartidas (heredadas) por las clases derivadas.

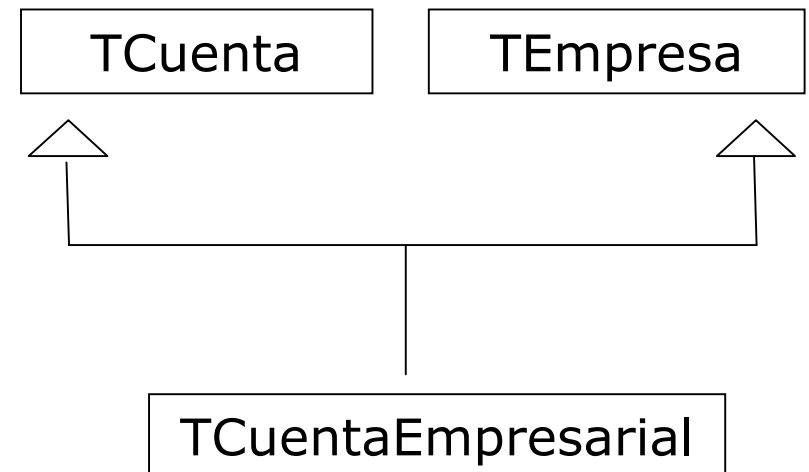
HERENCIA DE IMPLEMENTACIÓN

Herencia Múltiple

Ejemplo Herencia Múltiple



```
class TEmpresa{  
    protected:  
        string nomEmpresa;  
    public:  
        TEmpresa(string unaEmpresa) : nomEmpresa(unaEmpresa)  
        {};  
  
        void setNombre(string nuevoNombre) {  
            nomEpresa = numvoNombre; }  
  
        ~TEmpresa() {}  
};
```



¿Cómo implementar TCuentaEmpresarial?

Ejemplo Herencia Múltiple (II)



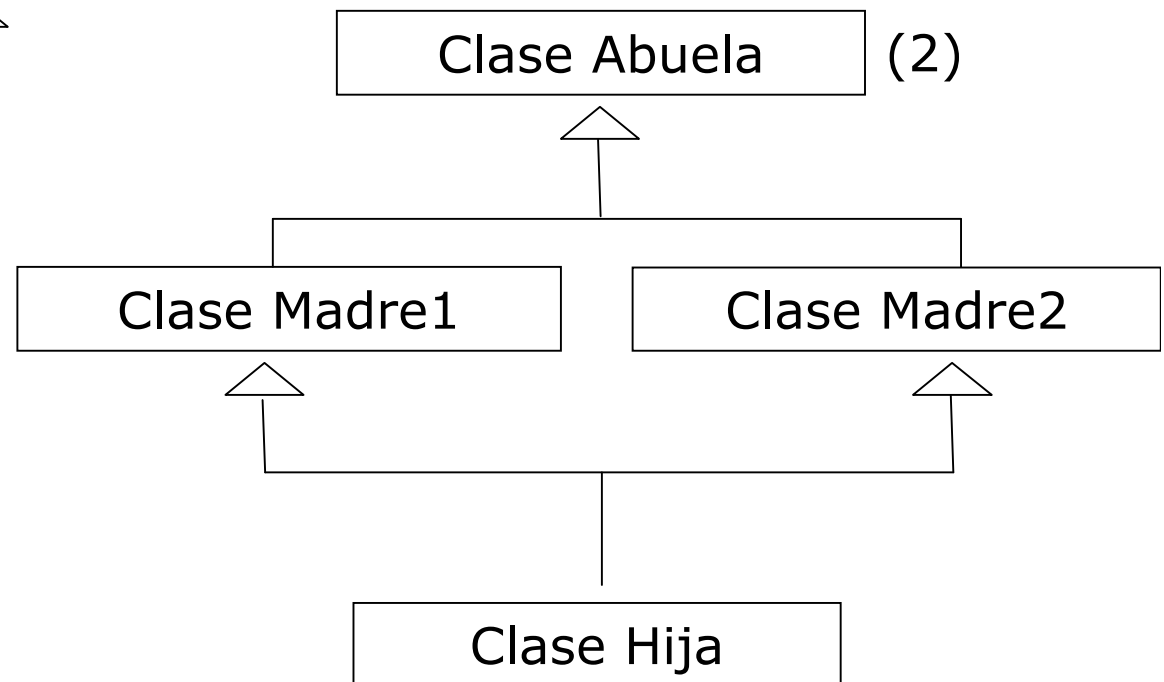
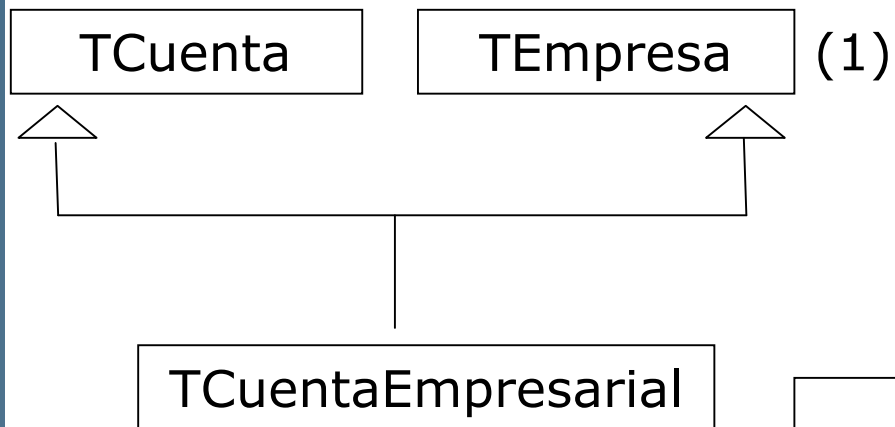
```
class TCuentaEmpresarial
: public TCuenta, public Tempresa {

    public:

        TCuentaEmpresarial(string unNombreCuenta,
            string unNombreEmpresa,
            double unSaldo=0, double unInteres=0)
            : TCuenta(unNombreCuenta, unSaldo, unInteres),
              Tempresa(unNombreEmpresa)
        {};

};
```

Problemas en Herencia Múltiple



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



Resolver los nombres mediante ámbitos:

```
class TCuentaEmpresarial: public TCuenta,  
    public Tempresa {  
    ...  
    { ... string n;  
        if ...  
            n= TCuenta::getNombre( );  
        else  
            n= Tempresa::getNombre( );  
        }  
    };
```

Duplicación de propiedades en herencia múltiple



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).
- Se utiliza exclusivamente con el propósito de garantizar la **sustituibilidad**.

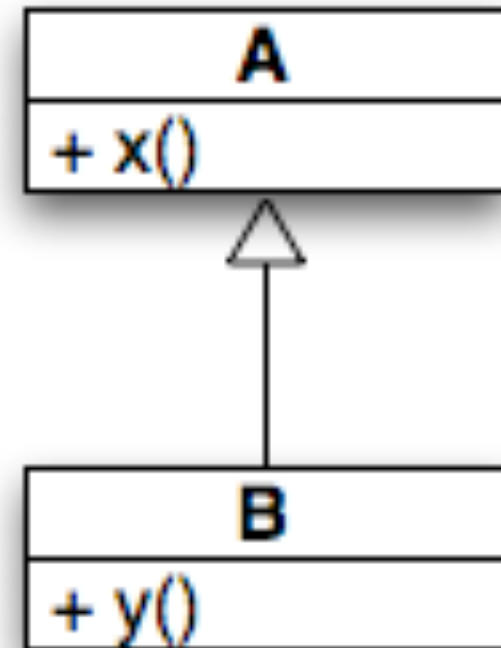
Herencia de interfaz



```
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

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

```
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
```

- **Java:** directamente

```
class Dependiente {
    public int cobrar();
    public void darRecibo();
...};

class Panadero
    extends Dependiente
{...}

Panadero p = new Panadero();
Dependiente d1=p; // sustit.
```

HERENCIA DE INTERFAZ

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

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.
 - 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.

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

```
TCuenta* tc = new TCuentaJoven; // sustitución  
  
tc->abonarInteresMensual();  
// Enlace dinámico: TCuentaJoven::abonarInteresMensual()
```

HERENCIA DE INTERFAZ

Métodos virtuales

```
TCuenta* tc = new TCuentaJoven;  
tc->abonarInteresMensual();  
// Enlace dinámico: TCuentaJoven::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).

En muchos lenguajes OO este rasgo es soportado de forma natural:
Por ejemplo, en Java, los métodos son virtuales por defecto.

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.

HERENCIA DE INTERFAZ

Métodos virtuales

```
class TCuenta {  
    ...  
    virtual void abonarInteresMensual() ;  
};
```

```
TCuenta* tc = new TCuentaJoven;
```

```
delete tc; // ¿Qué destructor se invoca? ~TCuenta()
```

Queremos destruir el objeto de tipo TcuentaJoven a través de un puntero a Tcuenta

- Necesitamos que el destructor de la clase base tenga enlace dinámico.

```
class TCuenta {  
    ...  
    virtual ~TCuenta() ;  
};
```

```
delete tc; // ¿Qué destructor se invoca? ~TCuentaJoven()
```

HERENCIA DE INTERFAZ

Métodos virtuales y el principio de sustitución

```
class TCuenta {  
    ...  
    virtual void abonarInteresMensual();  
    virtual ~TCuentaVirtual();  
}
```

- En resumen, si una clase tiene métodos virtuales
 - Es posible utilizar el principio de sustitución.
 - Heredar de ella y sobreescribir los métodos virtuales
 - Para destruir objetos derivados desde punteros a clase base, el destructor de esta clase base debe declararse como método virtual.

HERENCIA DE INTERFAZ

Clases abstractas

- Alguno de sus métodos no está definido: son métodos abstractos
- No se pueden crear objetos de estas clases.
- Sí se pueden crear referencias (o punteros) a objeto de una clase abstracta (que apuntarán a objetos de clases derivadas)
 - Propósito: Garantizar que las clases derivadas proporcionan una implementación propia de ciertos métodos.
 - Se garantiza el principio de sustitución.

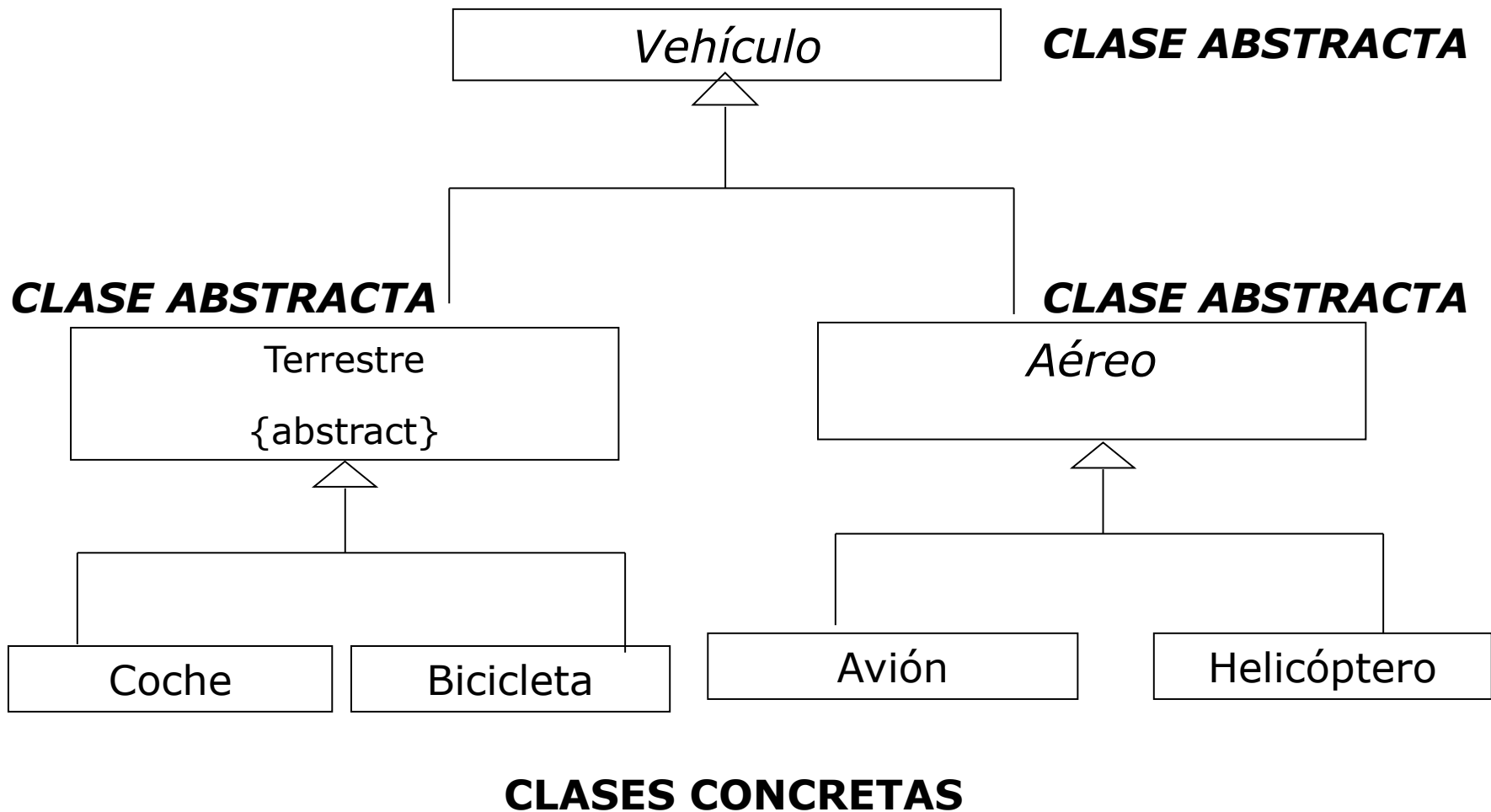
HERENCIA DE INTERFAZ

- **Clases abstractas**

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

HERENCIA DE INTERFAZ

Notación UML para clases abstractas



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() {...};
    ...
}
```

Los métodos abstractos tienen enlace dinámico (son métodos virtuales)

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()  
        {...};  
    ...  
}
```

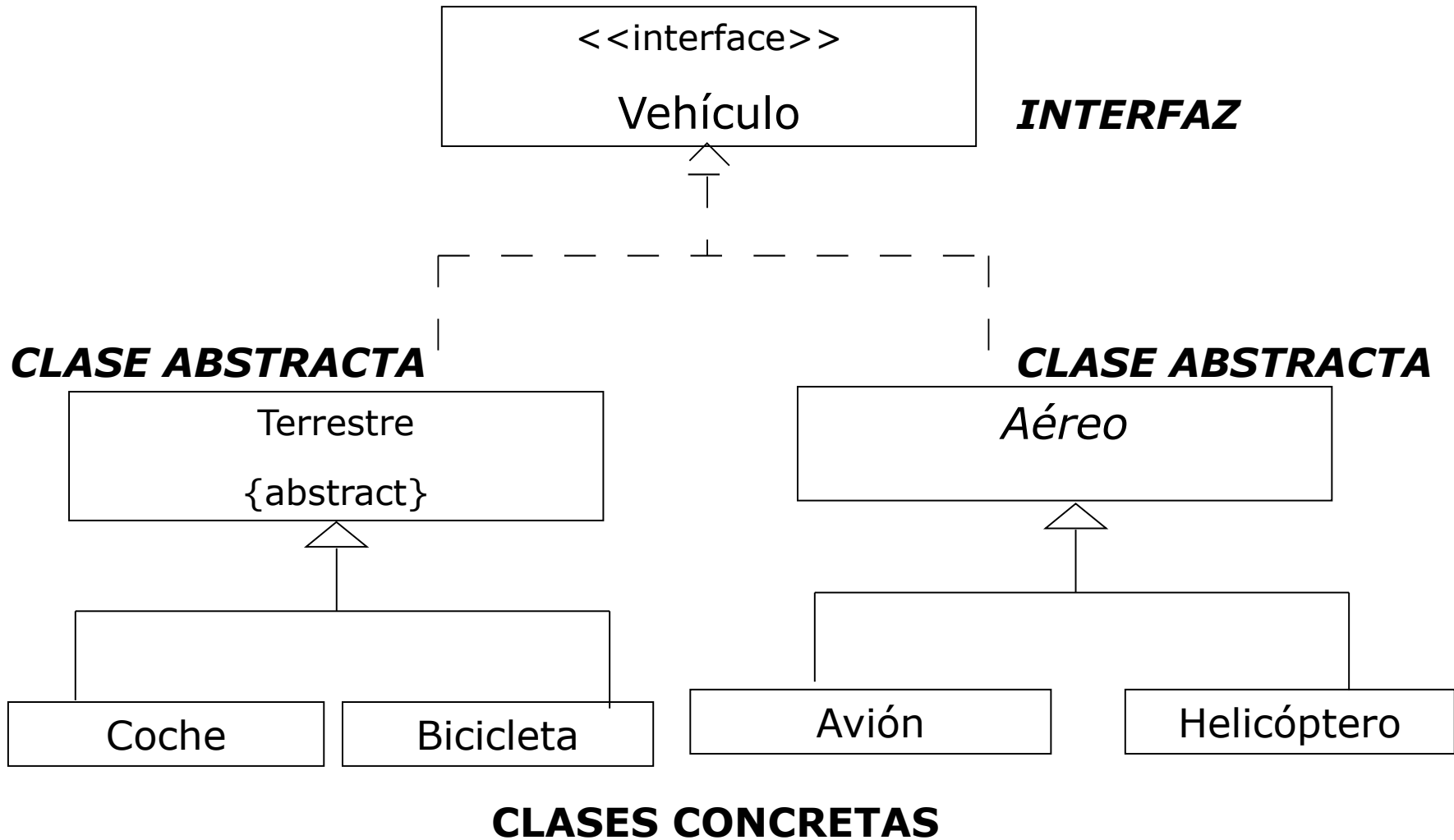
HERENCIA DE INTERFAZ

- **Interfaces**

- Declaración de un conjunto de métodos abstractos.
- En C++, son clases abstractas donde todos sus métodos son abstractos y no existen atributos de instancia.
 - Las clases derivadas heredan el interfaz mediante herencia pública
- Java/C#: declaración explícita de interfaces
 - Las clases pueden implementar más de un interfaz (herencia múltiple de interfaces)

HERENCIA DE INTERFAZ

Notación UML para interfaces



HERENCIA DE INTERFAZ

• Interfaces en Java

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

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

HERENCIA DE INTERFAZ

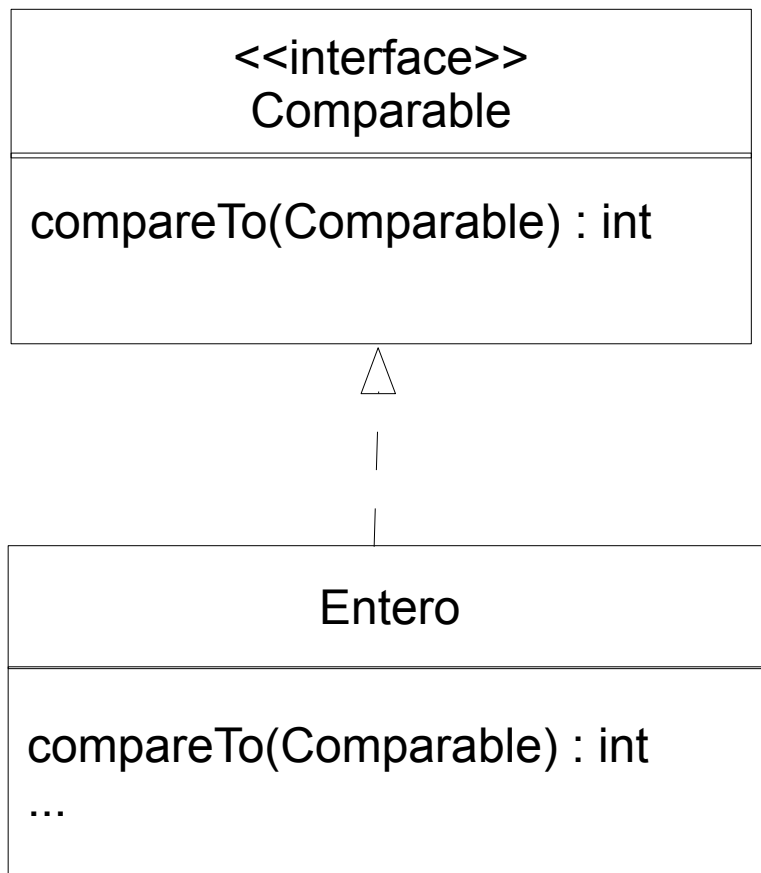
- Interfaces en C++

```
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);
}
```

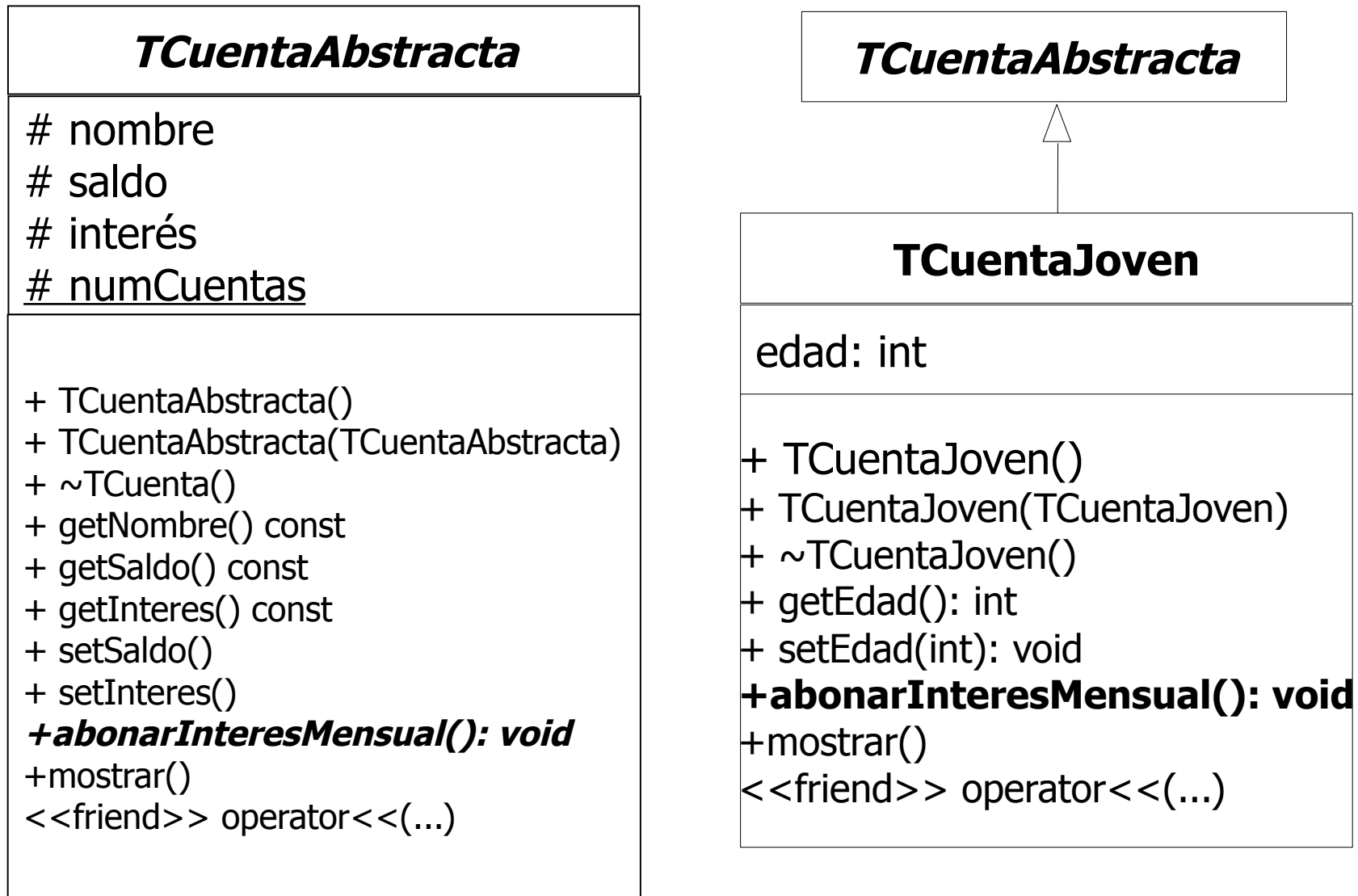
```
class Entero implements Comparable {
    private int n;

    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 clase abstracta (C++)



HERENCIA DE INTERFAZ

```
class TCuentaAbstracta {  
    protected:  
        ...  
    public:  
        ...  
    virtual void abonarInteresMensual()=0;  
};
```

```
class TCuentaJoven: public TCuentaAbstracta {  
    private:  
        int edad;  
    public:  
        ...  
    // IMPLEMENTACION en clase derivada  
    void abonarInteresMensual() {  
        //no interés si el saldo es inferior al límite  
        if (getSaldo() >= 10000) {  
            setSaldo(getSaldo() * (1 + getInteres() / 12 / 100));  
        }  
    };  
    // sigue...
```


HERENCIA DE INTERFAZ

¿Upcasting a clase abstracta?

```
TCuentaJoven tcj;
```

```
¿ TCuenta tc = (TCuenta)tcj; ?
```

```
¿ TCuenta* ptc = (TCuenta *) &tcj; ?
```

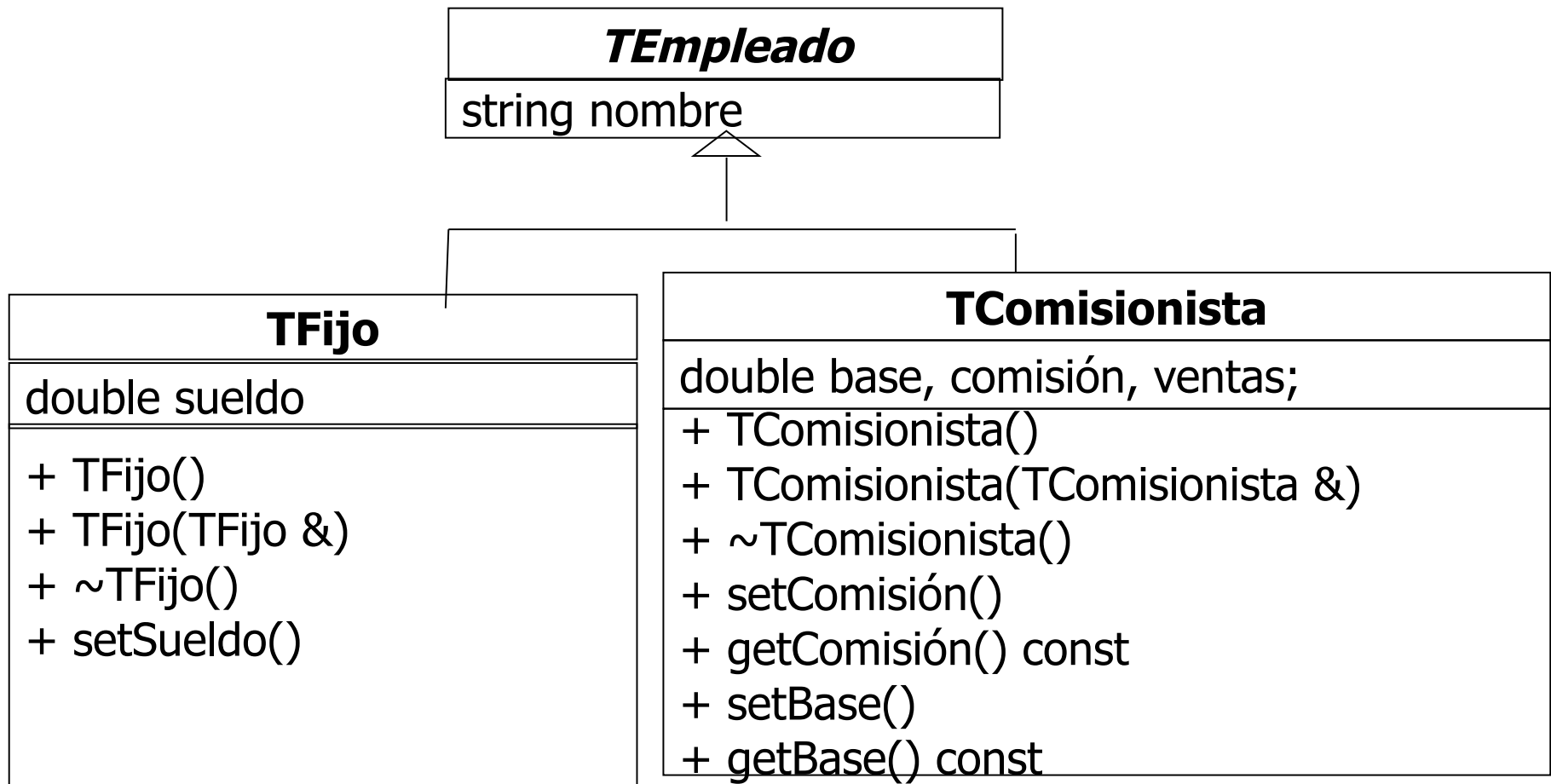
```
¿ TCuenta& rtc = (Tcuenta &) tcj; ?
```

```
TCuentaJoven tcj;
```

```
TCuenta* ptc = &tcj;
```

```
TCuenta& rtc = tcj;
```

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.

```
int main() {
    int tipo;
    Empleado *eptr;
    cout<<"Introduce tipo"<<endl;
    cin>>tipo; //1:fijo, 2 comisionista
    switch (tipo){
        case 1: eptr=new Fijo();
        break;
        case 2: eptr=new Comisionista();
        break;
    }
    eptr->getSalario();
    delete eptr;
}
```

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.

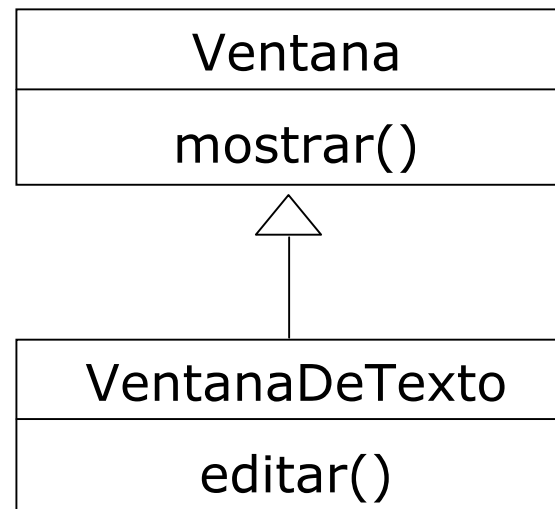


- En la herencia existe una tensión entre expansión (adición de métodos más específicos) y contracción (especialización o restricción de la clase padre)
- Esta tensión está en la base de su poder, y también de los problemas asociados con su uso.
- En general, la redefinición de métodos sólo debería usarse para hacer las propiedades más específicas
 - Constreñir restricciones
 - Extender funcionalidad



■ **Especialización**

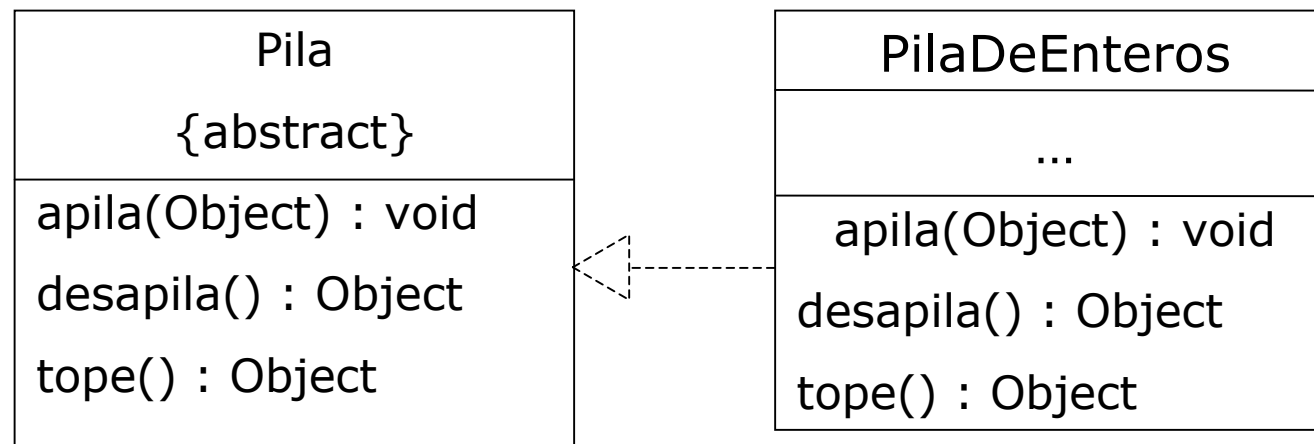
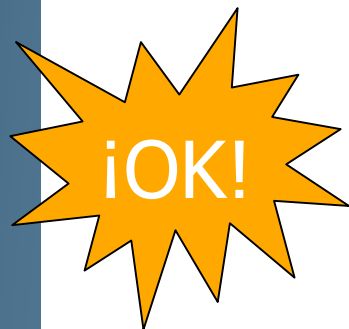
- La clase derivada es una **especialización** de la clase base: añade comportamiento pero no modifica nada
 - Satisface las especificaciones de la clase base
 - Se cumple el principio de sustitución (subtipo)





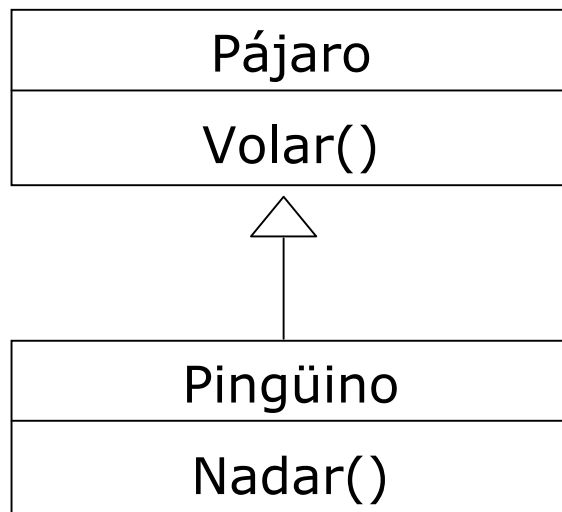
■ **Especificación**

- 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)**



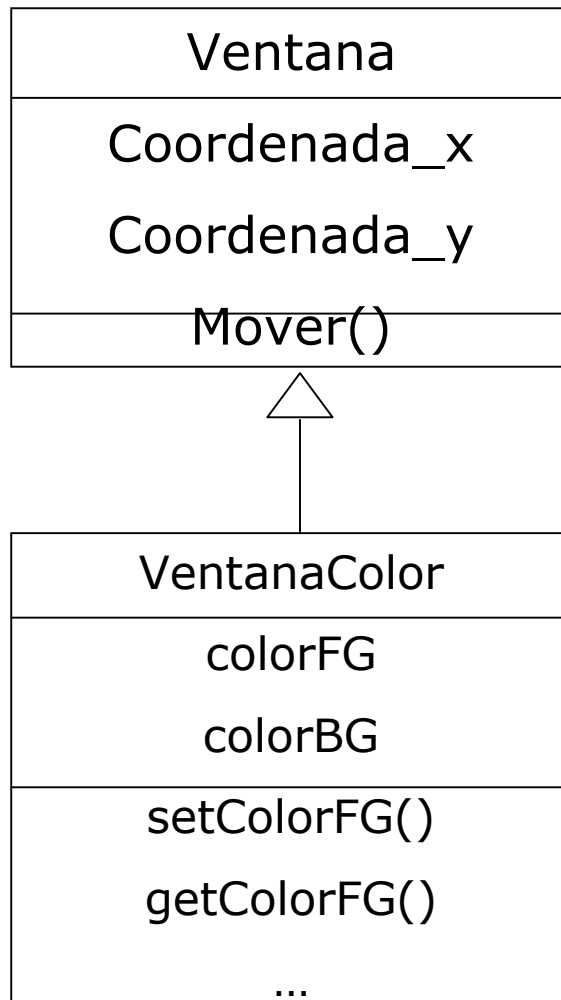
No todo lo de la clase base sirve a la derivada.

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)



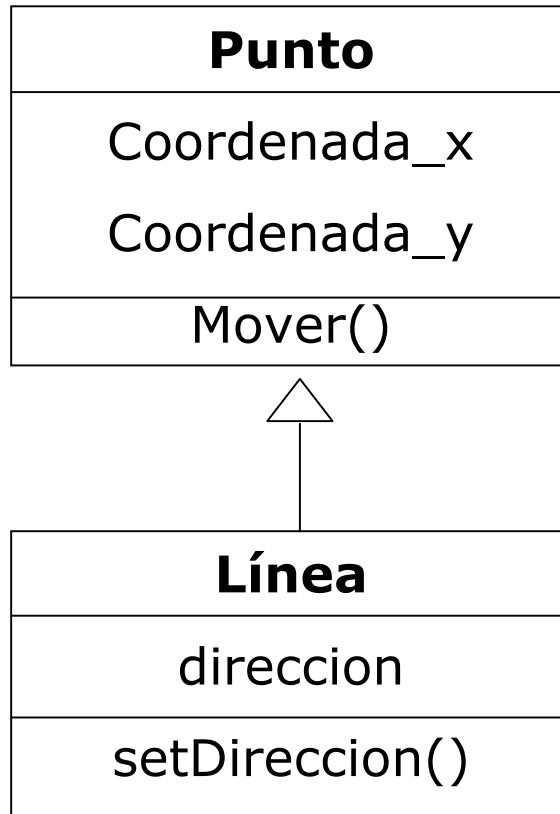
■ Generalización



- 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)**



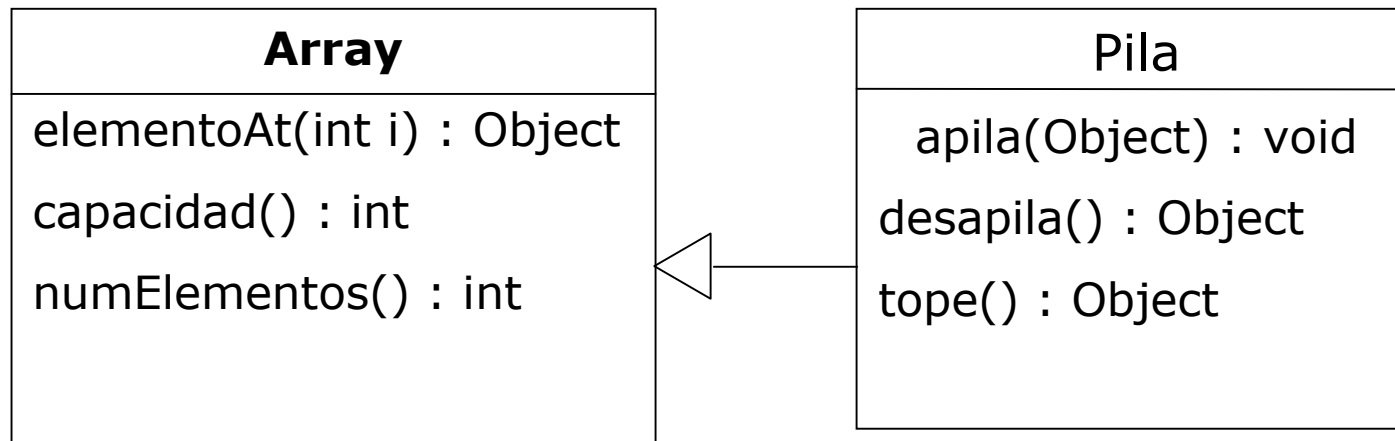
La implementación se parece pero semánticamente los conceptos no están relacionados jerárquicamente (test "es-un").

INCORRECTA!!!!

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

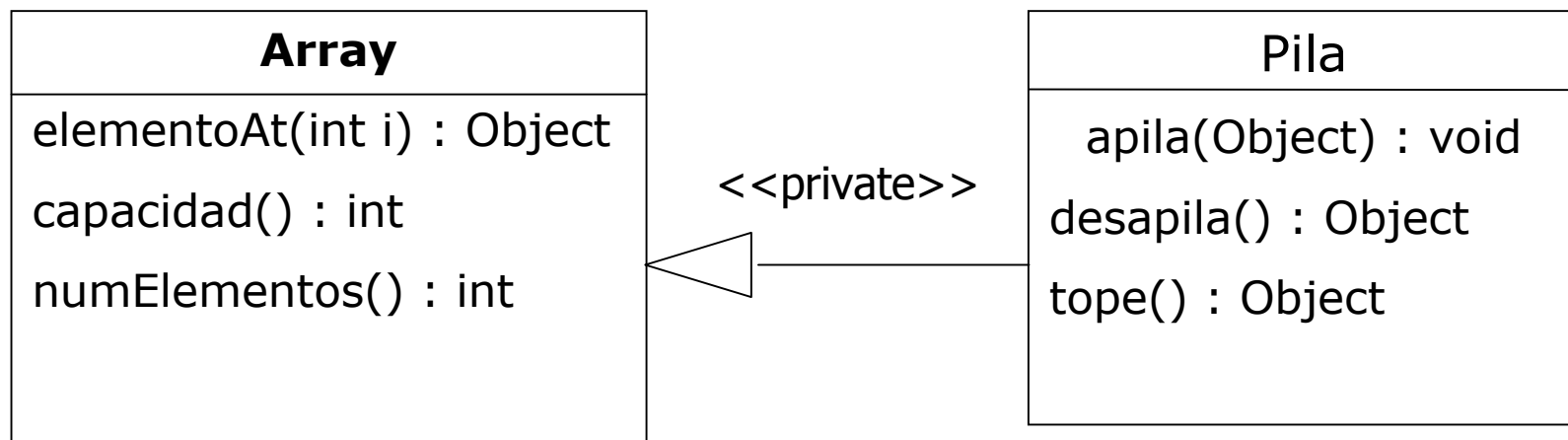


- También llamada ***Herencia de Implementación Pura***
- Una clase hereda parte de su funcionalidad de otra, modificando el interfaz heredado
- La clase derivada no es una especialización de la clase base (puede que incluso no haya relación "es-un")
 - No se cumple el principio de sustitución (ni se 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 es una relación entre clases, mientras que Agregación/Composición es una relación entre objetos
 - Herencia es menos flexible
 - Donde se detecta una relación HAS-A no siempre es posible cambiarla por una relación de herencia. Sin embargo, donde se detecta una relación de herencia, **siempre** es posible reformularla para que se convierta en una relación de composición.
 - Un programador de C++ es un programador
 - Todo programador de C++ tiene un programador en su interior
 - Todo programador de C++ tiene una vocación de programar en su interior



- **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).



- 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: public vehiculo{
    ...
}
```

Elección de técnica de reuso

Introducción



- Ejemplo: construcción del tipo de dato 'Conjunto' a partir de una clase preexistente 'Lista'

```
class Lista{  
    public:  
        Lista(); //constructor  
        void add (int el);  
        int firstElement();  
        int size();  
        int includes(int el);  
        void remove (int pos);  
        ...  
};
```

- 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.

Elección de técnica de reuso

Uso de **Composición** (Layering)



- Un objeto es una encapsulación de datos y comportamiento. Por tanto, si utilizamos la Composición estamos diciendo que parte del estado de la nueva estructura de datos es una instancia de una clase ya existente.

```
class Conjunto{
    public:
        //constructor debe inicializar el objeto Lista
        Conjunto():losDatos(){};
        int size(){return losDatos.size();};
        int includes (int el){return losDatos.includes(el);};
        //un conjunto no puede contener valor más de una vez
        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 tipos de datos totalmente distintos, y se supone que ninguno de ellos puede sustituir al otro en ninguna situación.
- La composición se puede aplicar del mismo modo en cualquier lenguaje OO e incluso en lenguajes no OO.

Elección de técnica de reuso

Uso de **Herencia**



- Con herencia una clase nueva puede ser declarada como una subclase de una clase existente, lo que provoca que todas las áreas de datos y funciones asociadas con la clase original se asocien automáticamente con la nueva abstracción de datos.

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

- Implementamos en términos de clase base (no objeto)
 - 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.
 - Así, las instancias de la nueva abstracción deberían comportarse de manera similar a las instancias de la clase padre.
 - En nuestro ejemplo, un Conjunto NO ES una Lista.
- En realidad deberíamos:
 - Usar herencia privada
 - Reescribir todos los métodos de Conjunto basando su implementación en los de Lista.

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.
 - La herencia es ligeramente más eficiente que la composición (evita una llamada).
- 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.

Elección de técnica de reuso

Ejemplos Composición vs. Herencia



- Clase Persona y clase Empleado
 - Herencia: un empleado es una persona.
- Clase Persona y clase Domicilio
 - Composición: una persona tiene un domicilio
- Clase Lista y clase Nodo de la lista
 - Composición: una lista tiene un puntero de tipo nodo al nodo que está en cabeza de la lista (tener-un).
- Clase Empresa, clase Empleado y clase Jefe de grupo de empleados
 - Herencia entre empleado y jefe: Un jefe es un empleado
 - Composición entre empresa y empleado (o empleado y jefe):
 - Un empresa puede tener una lista de empleados y otra de jefes
 - Por el principio de los subtipos, una empresa puede tener una única lista donde aparezcan tanto los jefes como empleados.