SUBTOPIC 5

INHERITANCE – FIRST PART

Cristina Cachero, Pedro J. Ponce de León

Translated into English by Juan Antonio Pérez

version 20111015

INHERITANCE
Objectives

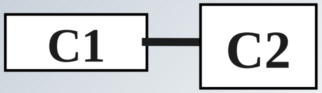
- Understanding inheritance as a powerful abstraction mechanism.
- Distinguishing among the different types of inheritance.
- Knowing how to implement inheritance hierarchies in Java.
- Distinguishing between safe (well defined) and unsafe inheritance hierarchies.
- Code reuse: deciding when to use inheritance and when to use composition.

Inheritance
Previously…

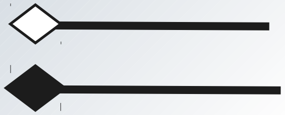
<table>
<tr><th></th><th>Persistent</th><th>Non-persistent</th></tr>
<tr><td><b>Object level relationships</b></td><td>

- **Association**

Ladybugs and flowers

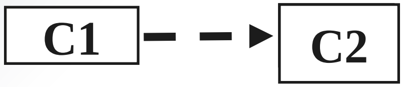


- **Whole/part** (“part of”/”has a”)
  - Aggregation
  - Composition



A petal is a part of a flower
</td><td>

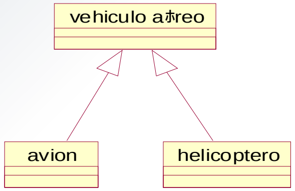
- **Using (dependency)**



</td></tr>
<tr><td><b>Class level relationships</b></td><td>

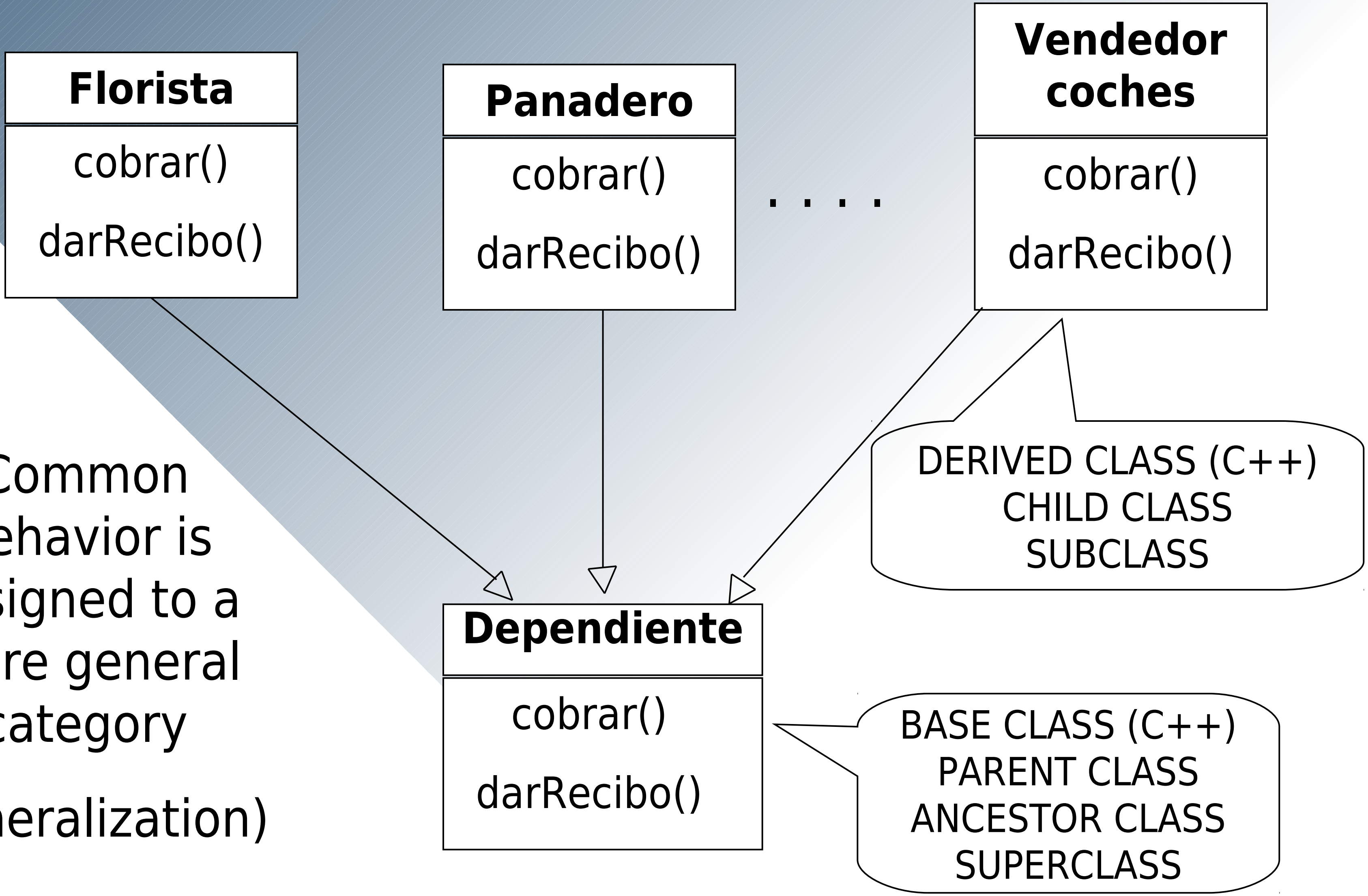
- **Generalization/Specialization** (“is a”) (inheritance)

A rose is a kind of flower



</td><td></td></tr>
</table>

INHERITANCE
Motivation



DERIVED CLASS (C++)
CHILD CLASS
SUBCLASS

BASE CLASS (C++)
PARENT CLASS
ANCESTOR CLASS
SUPERCLASS

Common behavior is assigned to a more general category

(generalization)

Classification and generalization

- The human mind groups concepts as:
  - Membership (HAS-A) -> *Whole/part relationships*
  - Variety (IS-A) -> *Inheritance*

- Using inheritance implies classifying abstractions (concepts):
  - A generalization of a concept is an extension of the concept to less-specific criteria. It is a foundational element of logic and human reasoning. Specialization is somehow the opposite concept.
  - When implemented in a programming language, generalization leads to inheritance hierarchies.

Inheritance as implementation of generalization

- Generalization is a semantic relationship between classes. Instances of a child class include all the properties of its parent class.
- The number of relationships (aggregations, compositions) in the model is reduced.
- Legibility and expressiveness of the model improves.
- The number of resulting classes usually increases.

INHERITANCE
Definition

- Inheritance is the mechanism by which more specific elements incorporate structure and behavior of more general elements (Rumbaugh 99)

- Inheritance makes possible **specializing** or **extending** the functionality of a class by deriving new classes from it.

- Inheritance is always **transitive**, so a class can inherit features from superclasses many levels away.
  - That is, if class *Dog* is a subclass of class *Mammal*, and class *Mammal* is a subclass of class *Animal*, then *Dog* will inherit attributes both from *Mammal* and from *Animal*.

INHERITANCE
Is-a test

- To tell if concept A should be linked by inheritance to concept B, try forming the English sentence “**An A is a B**”. If the sentence sounds right to your ear, the inheritance is most likely appropriate in this situation:
  - A bird is an animal
  - A cat is a mammal
  - An apple pie is a pie
  - An IntegerArray is an array

INHERITANCE
Is-a test

- On the other hand, the following assertions seem strange for one reason or another, and hence inheritance is likely not appropriate:
  - A bird is a mammal
  - An apple pie is an apple
  - An IntegerArray is an integer
  - An engine is a car
- Rarely, the is-a test fails.

INHERITANCE
Main uses

- *Inheritance as a means of code reuse.* Because a child class can inherit behavior from a parent class, the code does not need to be rewritten for the child. This can greatly reduce the amount of code needed to develop a new idea.
  - **Implementation inheritance**

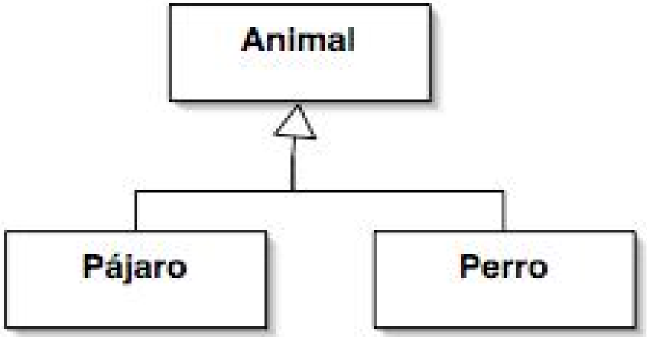
- *Inheritance as a means of concept reuse.* This occurs when a child class **overrides** behavior defined in the parent. Although no code is shared between parent and child, the child and parent share the definition of the method.
  - **Interface inheritance**

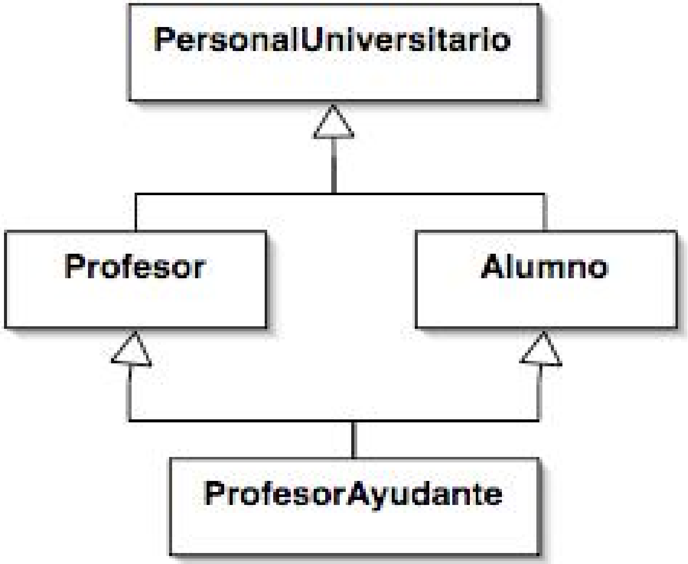
Types of inheritance

- Single/multiple inheritance
- Implementation/interface inheritance

Types of inheritance

- Single/Multiple
  - **Single**: one parent class
  - **Multiple**: two or more parent classes





Types of inheritance

- Implementation/interface inheritance
  - **Implementation inheritance**: the subclass inherits the implementation of the methods, although they can be overriden in the subclass.
  - **Interface implementation**: only the interface is inherited; no implementation is provided by the base class (*interfaces* in Java, *abstract classes* in C++)

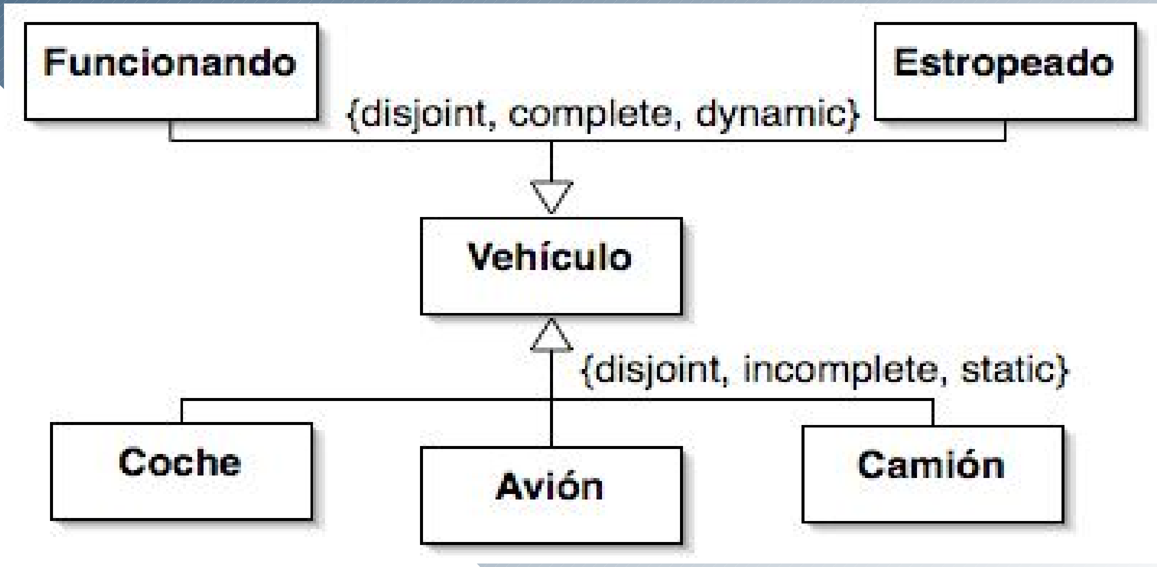
Inheritance

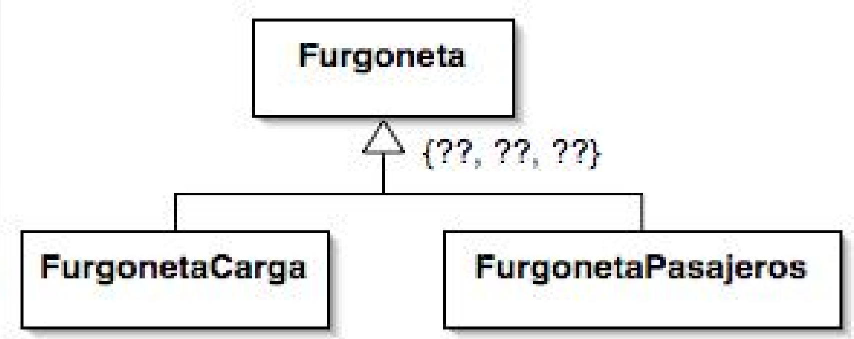
Semantic description

- Generalization constraints:
  - **Overlapping/Disjoint**
    - Determines whether an instance of the superclass can be an instance of multiple subclasses (overlapping) or not.
    - Overlapping inheritance is not supported in Java/C++ (strong typing)
  - **Complete/Incomplete**
    - Determines whether every instance of the superclass is an instance of at least one subclass (complete) or not.
  - **Static/Dynamic**
    - Determines whether an object which is an instance of a subclass may become an instance of a sibling class.
    - Dynamic inheritance is not supported in Java/C++ (strong typing)

Inheritance

Semantic description





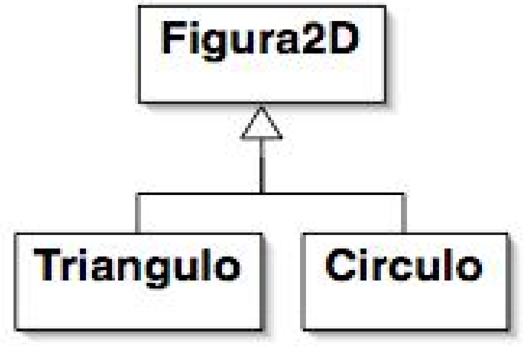
IMPLEMENTATION INHERITANCE

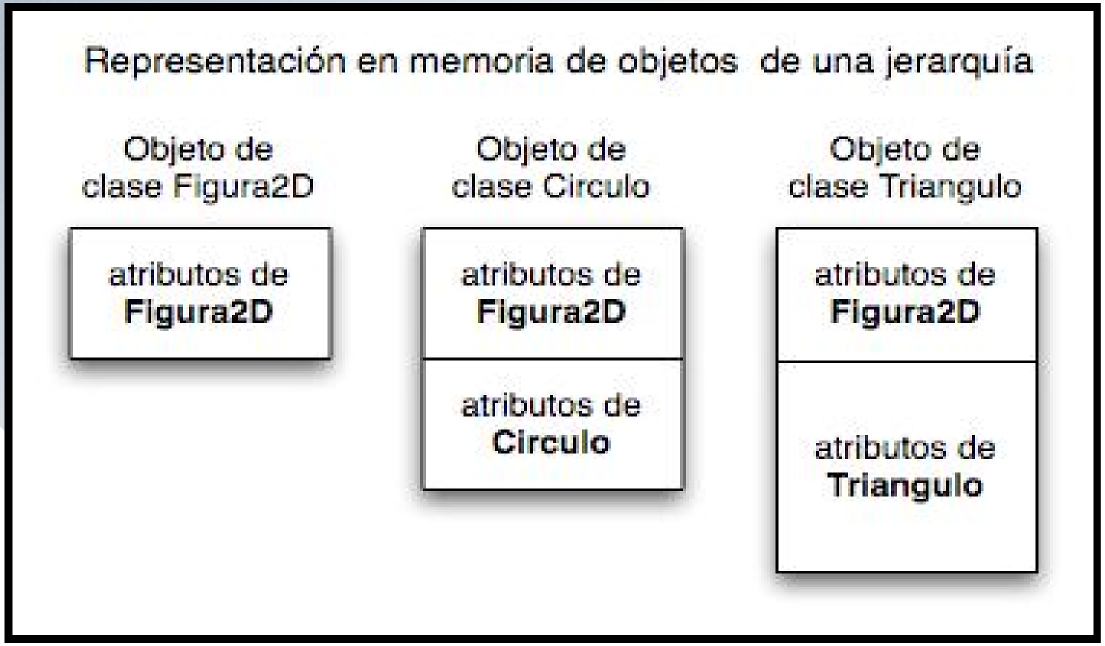
Single Inheritance

Single Inheritance

- By inheritance, we mean the property that instances of a child class can access both data and behavior (methods) associated with a parent class.

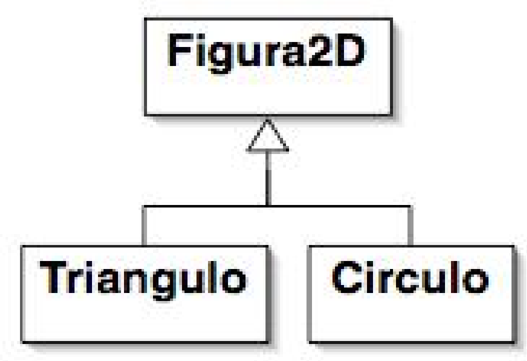
- Derived classes may add new attributes, methods or roles.





Single Inheritance

```
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 is private
      setColor(Color.NINGUNO); // OK
    }
}
```



Private members of the superclass are not directly accessible from the subclasses.

```
// client code
  Circulo c = new Circulo();
  c.setColor(AZUL);
  c.getColor();
  c.vaciarCirculo();
```

Single inheritance

Attribute/method visibility

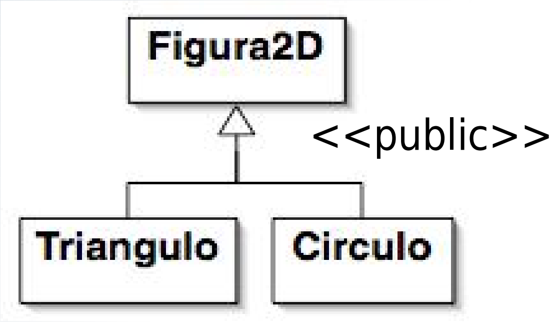
- **Protected** visibility scope
  - A protected feature is accessible only within a class definition or within the definition of any child classes. They are identified in UML by the symbol ‘#’.

```
    class Figura2D {
        protected Color colorRelleno;
    ...
    }

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

```
// client code
  Circulo c;
  c.colorRelleno=NINGUNO;
 // ERROR! colorRelleno
 // is private here
```

Types of single inheritance

- Public inheritance
  - Both implementation and interface are inherited.



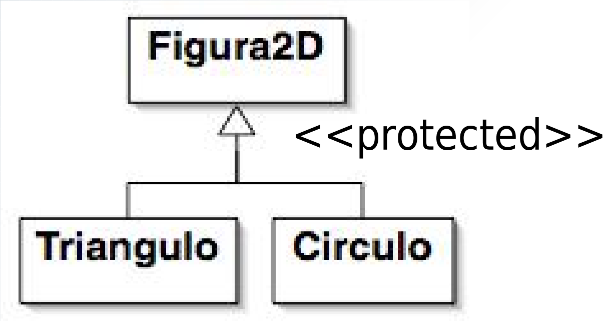
```
// JAVA only supports public inheritance
class Circulo extends Figura2D
{
...
}
```

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

Types of single inheritance

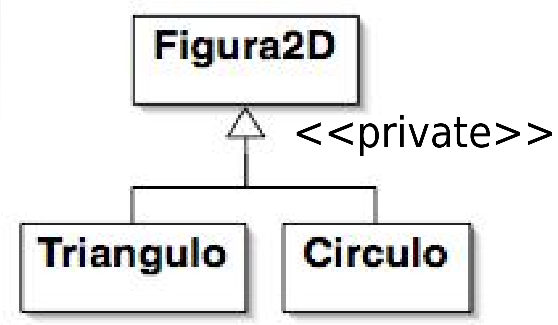
- Protected inheritance (C++)

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



- Private inheritance (C++, default)

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



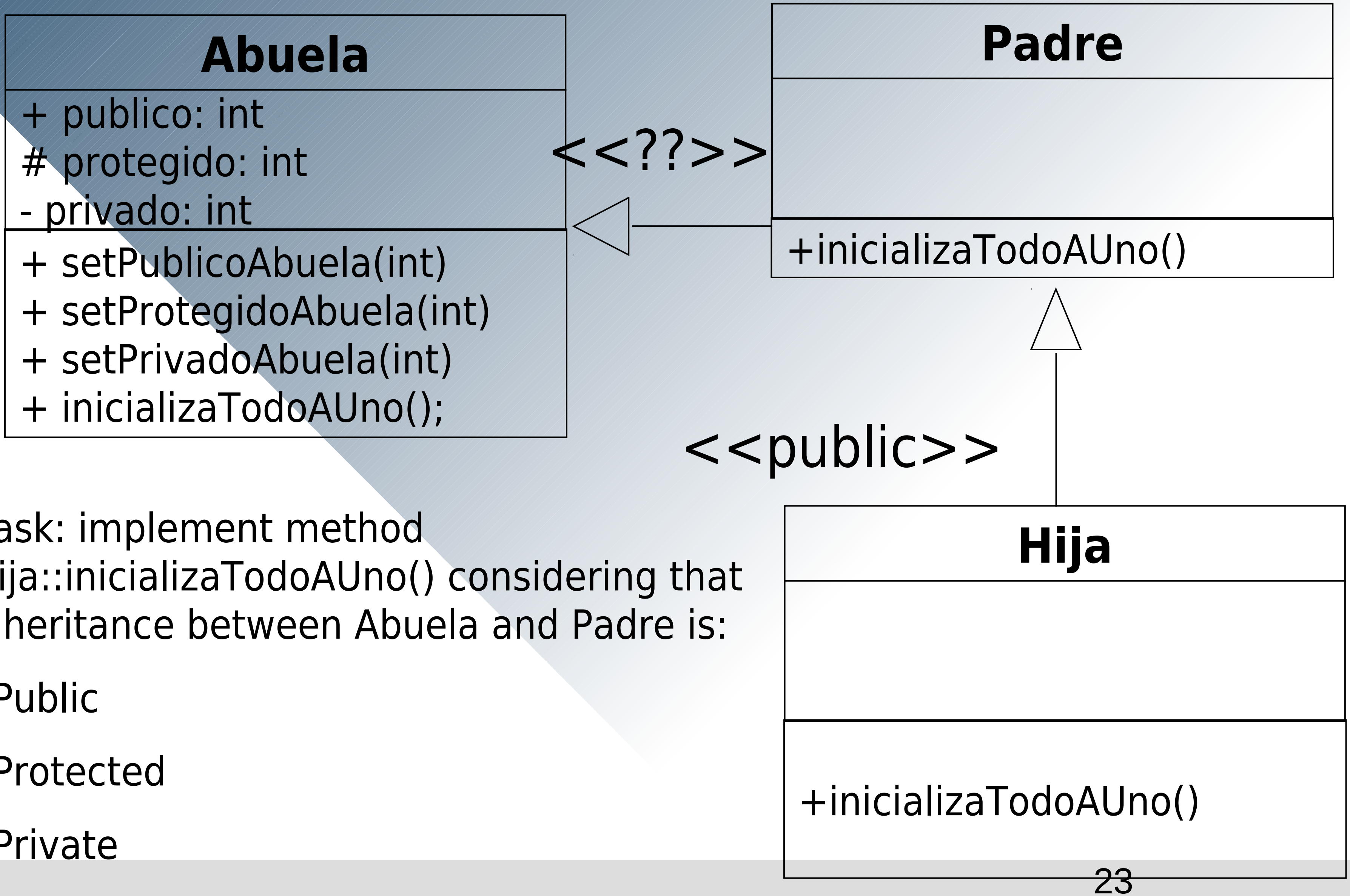
These types of inheritance in C++ only allow to inherit implementation. The interface of the base class is not shared with the derived class. More information.

Types of single inheritance

Inheritance Scope Visibility In base class	DC (*) Public inheritance	DC Protected inheritance	DC Private inheritance
Private	Not directly accessible	Not directly accessible	Not directly accessible
Protected	Protected	Protected	Private
Public	Public	Protected	Private

(*) DC: Derived class

Types of single inheritance
Task



Abuela

+ publico: int
# protegido: int
- privado: int

+ setPublicoAbuela(int)
+ setProtegidoAbuela(int)
+ setPrivadoAbuela(int)
+ inicializaTodoAUno();

<<??>>

Padre

+inicializaTodoAUno()

<<public>>

Hija

+inicializaTodoAUno()

Task: implement method Hija::inicializaTodoAUno() considering that inheritance between Abuela and Padre is:

- Public
- Protected
- Private

Single Inheritance

Methods in derived classes

- Derived classes may…
  - Add new methods/attributes
  - Modify methods inherited from the base class
    - **REFINEMENT**: a style of overriding in which the inherited code is merged (before and/or after) with the code defined in the child class (it can be simulated in C++, Java)
      - *C++*, *Java*: constructors and destructors are usually refined
    - **REPLACEMENT**: a style of overriding in which the inherited code is completely replaced by the code defined in the child class.

Single Inheritance

Methods in derived classes

- **Replacement** in 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”);
  }
}
```

Single Inheritance
Methods in derived classes

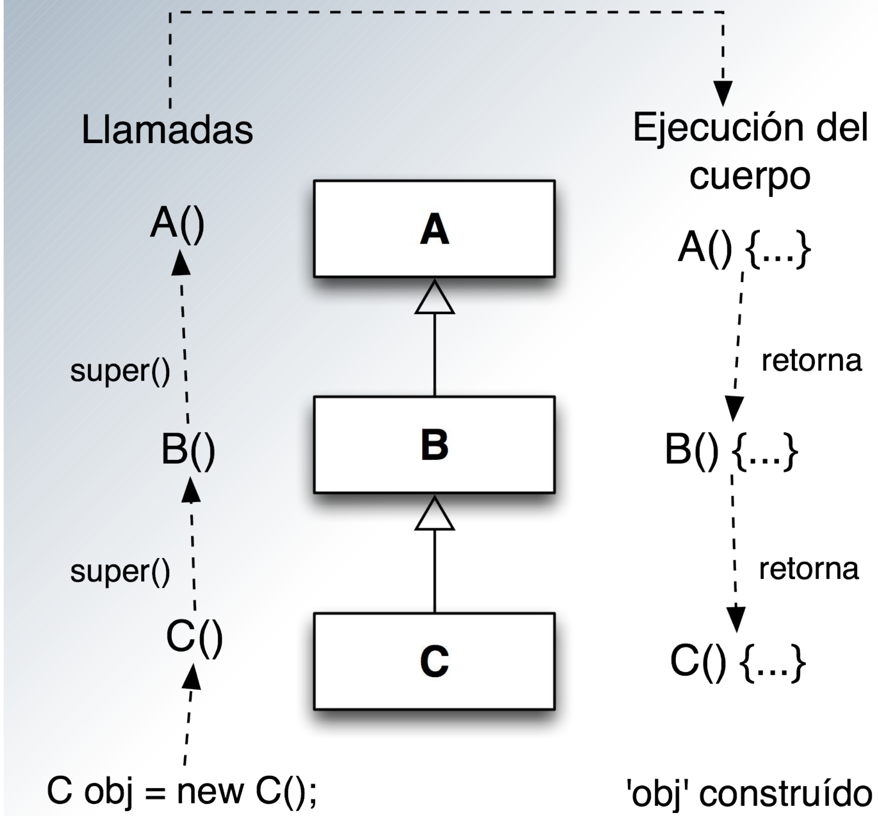
- **Refinement** in Java

```
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(); // base implementation after child implementation
  }
  public void doItAgain() {
    super.doItAgain(); // before
    System.out.println(“HECHO otra vez en B.”);
  }
}
```

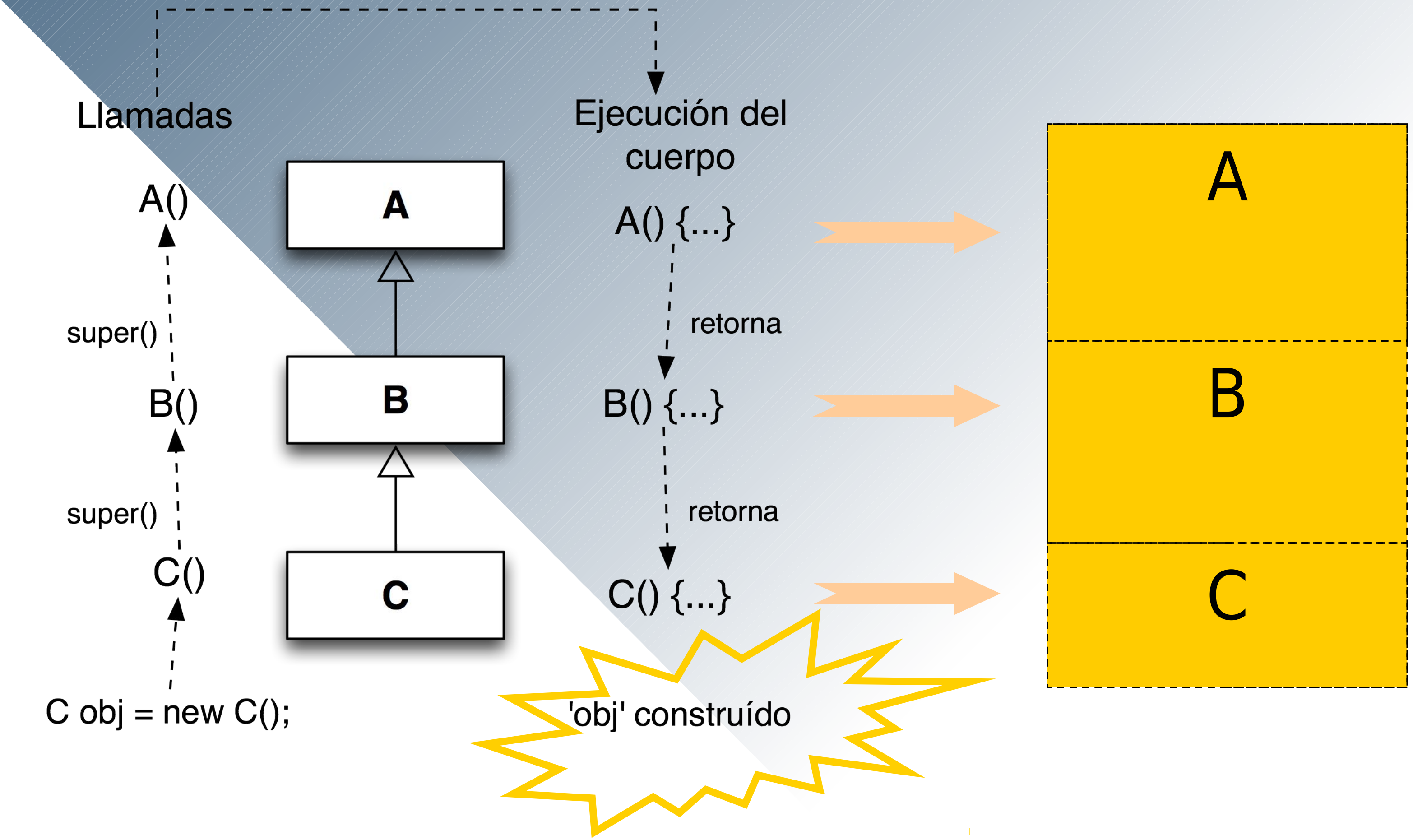
this : reference to the receiver of the message (current class implementation)
super : reference to the receiver of the message (using parent implementation)

Constructors under single inheritance

- Constructors are not inherited
  - Code in both classes must be executed.
  - Constructors must be defined in subclasses.
  - Creation of an object of the subclass: all the constructors (starting with the “highest” parent) are invoked.



Constructors under single inheritance



Constructors under single inheritance

- Derived classes **refine** the constructor in the base class.
- Implicit execution of the default parent constructor when invoking a constructor in the child class.
- Explicit execution of any other constructor (usually, requiring parameters) in the initialization part (explicit refinement); this happens, for instance, with copy constructors.

(WARNING: base class attributes should be initialized in the base class, not in the child class)

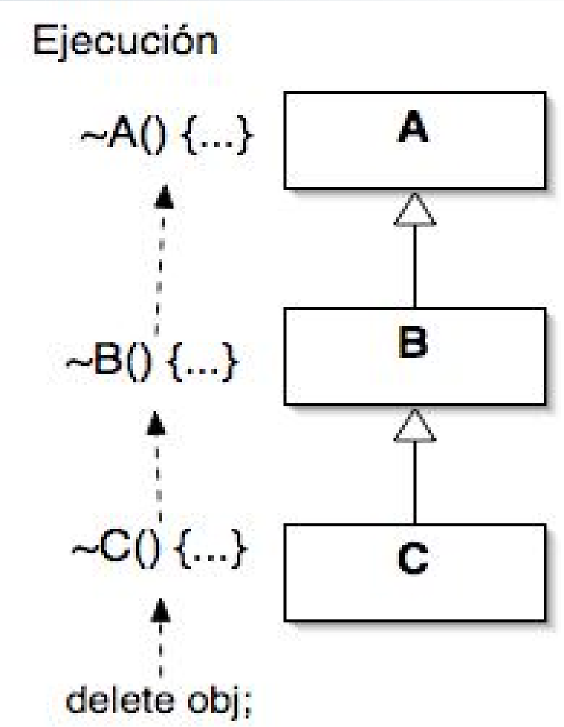
Constructors under single inheritance

- Example

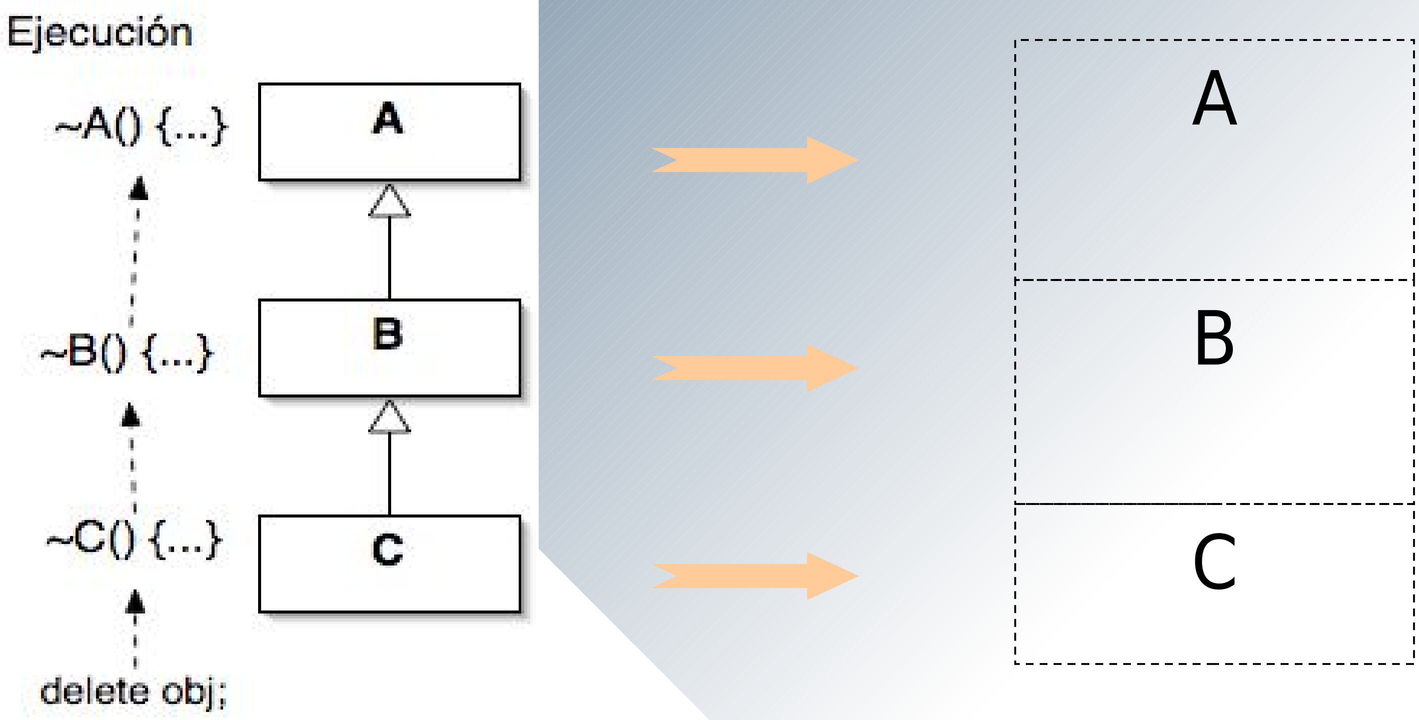
```
class Figura2D {
    private Color colorRelleno;
    public Figura2D() { colorRelleno= Color.NINGUNO; }
    public Figura2D(Color c) { colorRelleno=c; }
    public Figura2D(Figura2D f) { colorRelleno=f.colorRelleno; }

...}

class Circulo extends Figura2D {
  private  double radio;
  public Circulo() { radio=1.0; } // implicit call to Figura2D()
  public Circulo() { super(); radio=1.0; } // explicit call
  public Circulo(Color col, double r) { super(col); radio=r; }
  public Circulo(Circulo cir) { super(cir); radio=cir.radio; }
...}
```

Constructors under single inheritance (C++)

- C++: destructors are not inherited.
  - They must be defined for the derived class.
  - Destruction of an object of the subclass: all the destructors (starting with the subclass) in the hierarchy are invoked.
  - Base class destructors are called implicitly.

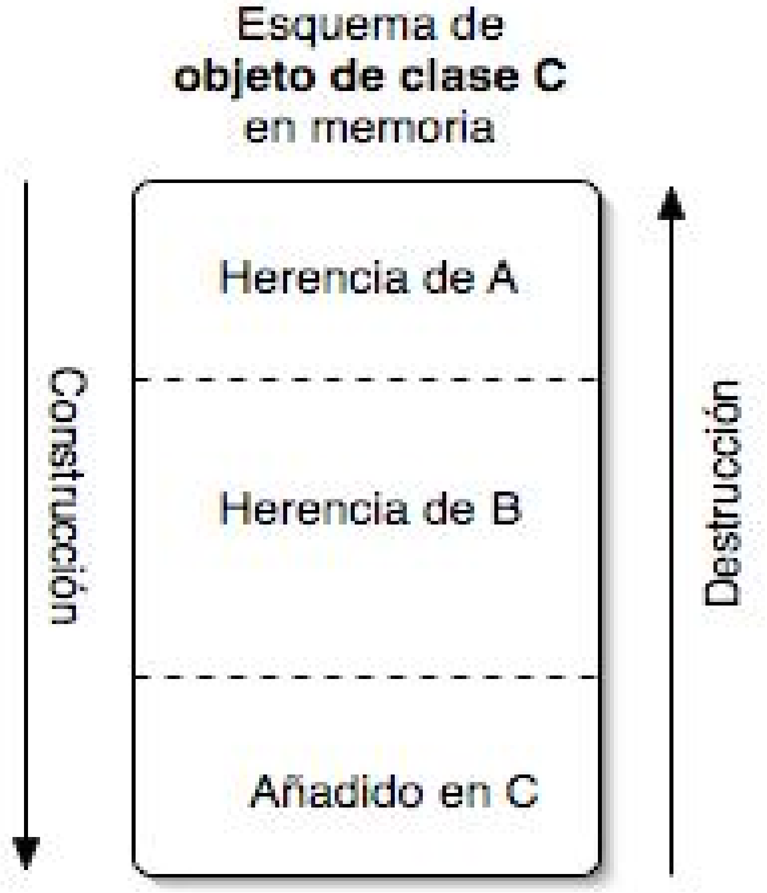


Destructors under single inheritance (C++)



Destruction/construction order in C++

- The destruction order in derived objects goes in exactly the reverse order of construction.



Destruction/construction order in Java

- In Java the construction order is the same than in C++: from the parent class to the child class.

- The destruction order is responsibility of the programmer. It must be implemented whenever resources (memory not included) to be released exist.

  - Two strategies:
    - Use **finalize**() methods
      - Disadvantage: it cannot be determined when they will be executed.

    - Create specific methods responsible of releasing the resources.
      - Disadvantage: client code must invoke explicitly those methods.

Destruction/construction order in Java

Cleanup/destruction using 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);
    // Warning: this method is deprecated
  }
} ///:~
```

(taken from 'Piensa en Java', 4th ed., Bruce Eckl)

Destruction/construction order in Java

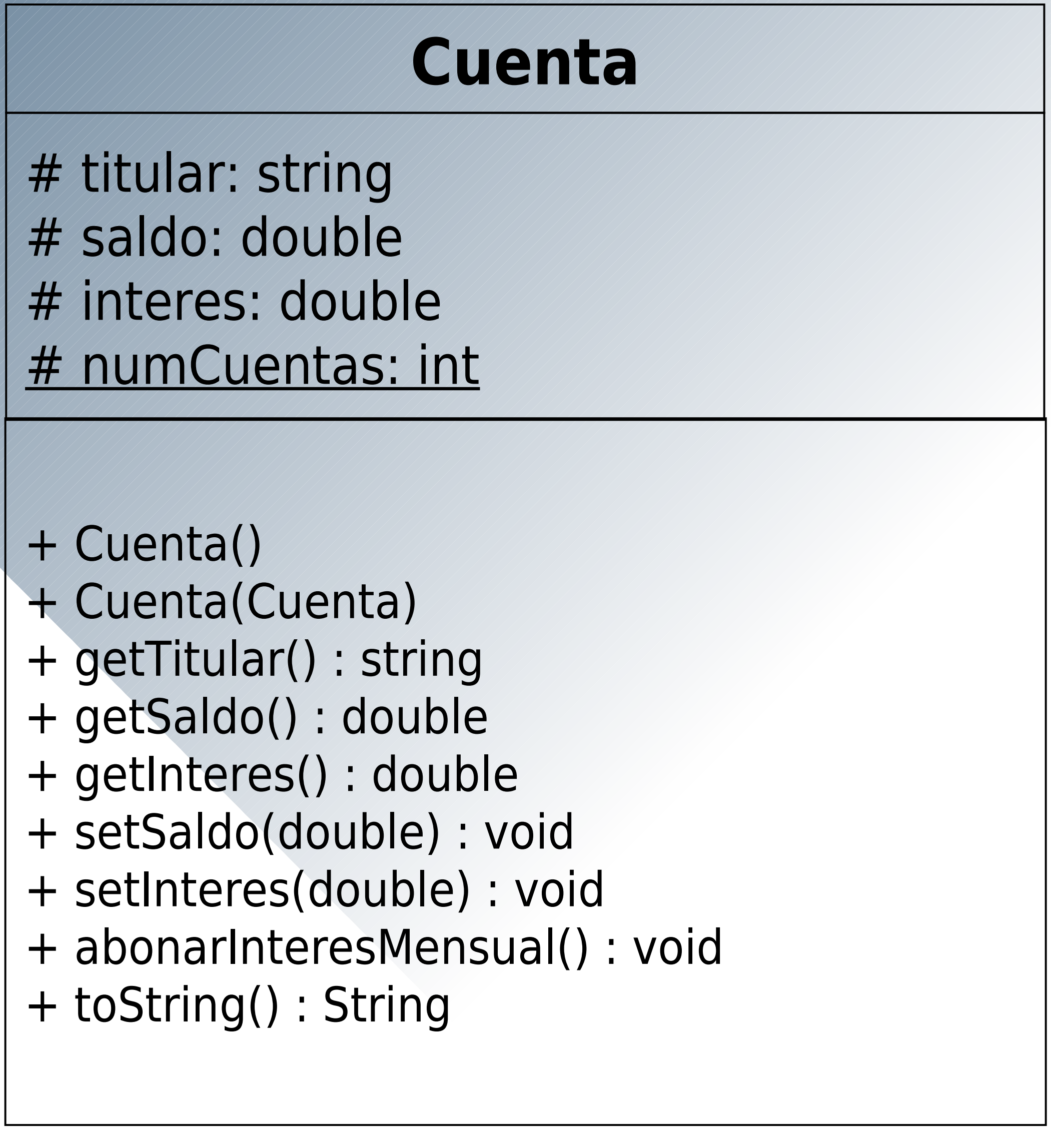
Cleanup/destruction using specific methods:

```
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();
    }
  }
}
```

(taken from 'Piensa en Java', 4th ed., Bruce Eckl)

Example: base class



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
```

Single inheritance (base class): Cuenta

```
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;
//...
```

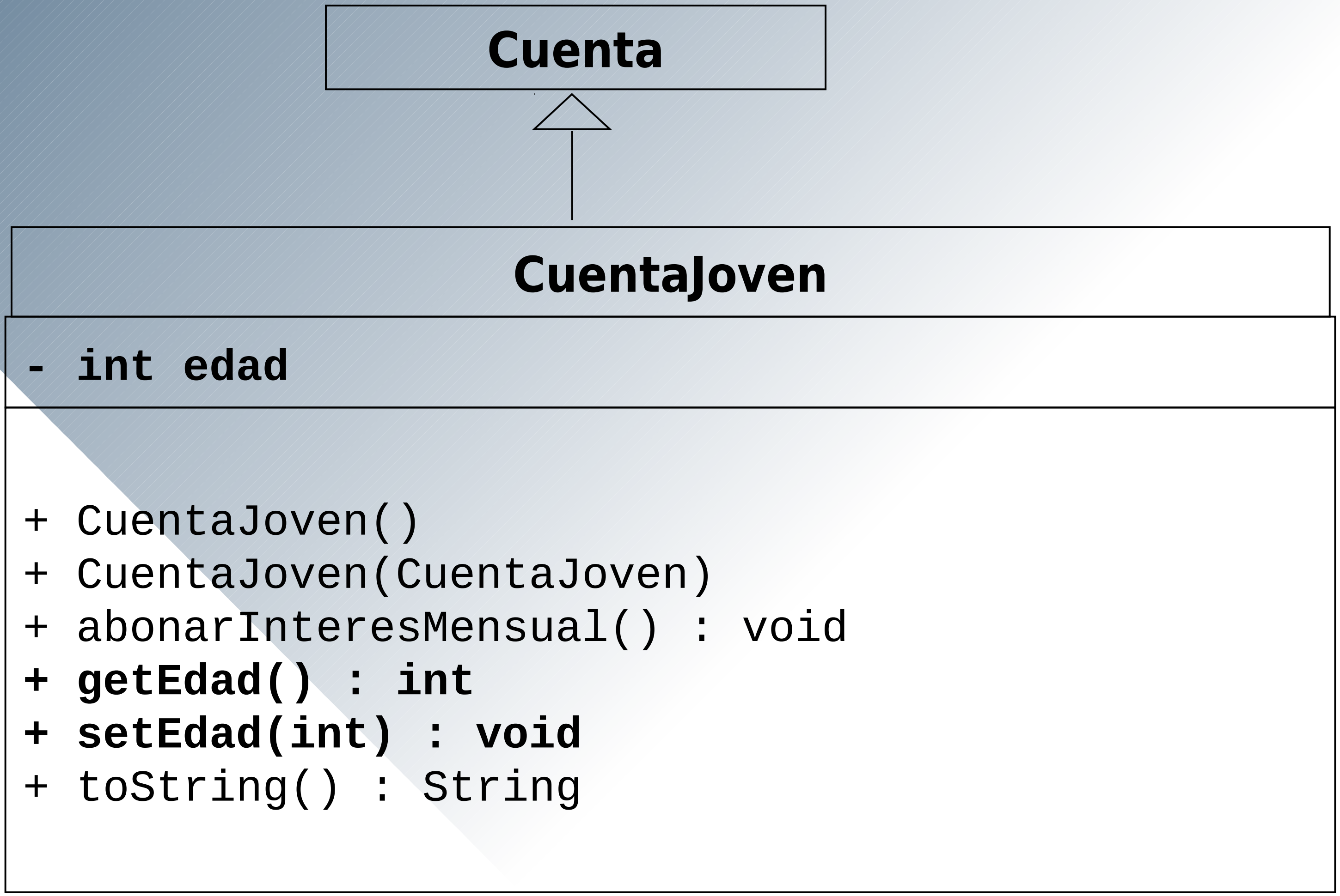
Single inheritance (base class): Cuenta (II)

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

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

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

Single inheritance (base class): Cuenta (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”;
  }
}
```

Example: derived class



Cuenta

CuentaJoven

- int edad

+ CuentaJoven()
+ CuentaJoven(CuentaJoven)
+ abonarInteresMensual() : void
+ getEdad() : int
+ setEdad(int) : void
+ toString() : String

(Methods whose implementation is inherited from the base class are not specified in UML)

Single inheritance (child class): 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(CuentaJove tcj)
  // explicit call to the copy constructor in Cuenta
  {
    super(tcj);
    edad=tcj.edad;
  }

...
```

Should numCuentas be incremented?

Refinement

Single inheritance (child class): CuentaJoven (II)

```
  ...
  void abonarInteresMensual() {
      // no interest if balance is below limit
      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;
  }
}// end of class CuentaJoven
```

Replacement

New methods added

Refined method

Single Inheritance
Upcasting

Upcasting is converting a derived-class reference to a base-class.

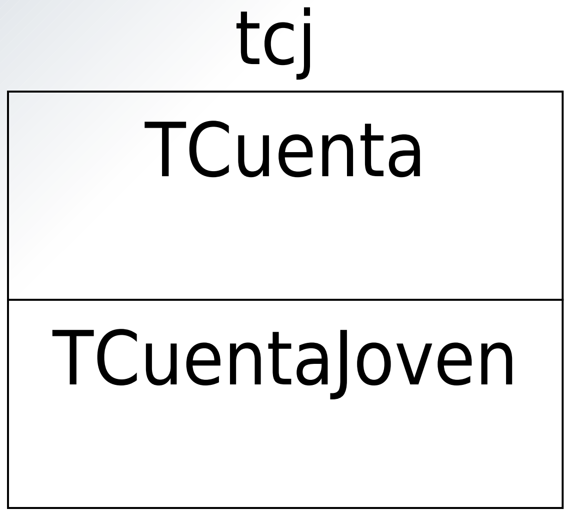
```
CuentaJoven tcj = new CuentaJoven();
Cuenta c;

c = (Cuenta)tcj; // explicit
c = tcj; // implicit

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

An object of the child class accessed through a reference to the base class can only be handled by using the interface of the base class.

Single inheritance: upcasting

Object slicing is used when converting objects (not pointers) in C++: fields in CuentaJoven which are not found in TCuenta are sliced off during the process.

```
CuentaJoven tcj;

(TCuenta)tcj
```



tcj

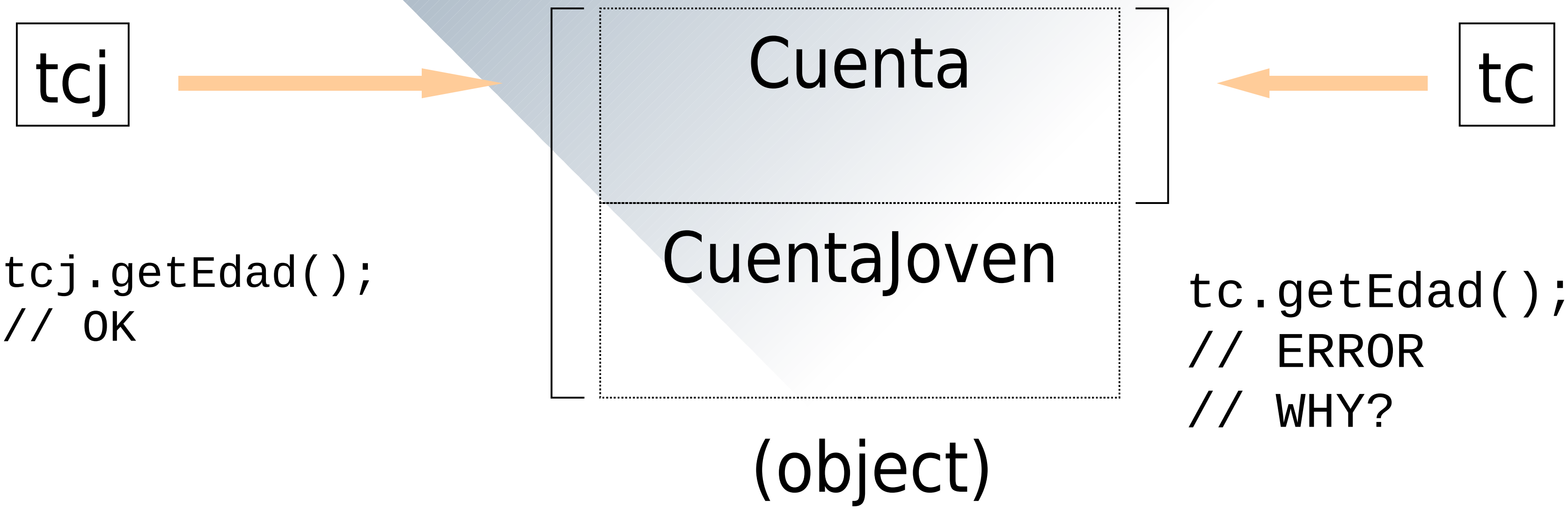
TCuenta

TCuentaJoven

Single inheritance: upcasting

When using <u>references</u> (in Java or C++), no **object slicing** is performed

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



```
tcj.getEdad();
// OK
```

```
tc.getEdad();
// ERROR
// WHY?
```

Some facts about inheritance

- In inheritance hierarchies an implicit refinement exist for:
  - Default constructors

- Overloaded constructors are refined explicitly.

- Class (static) properties defined in the base class are shared (inherited) with the child classes.

IMPLEMENTATION INHERITANCE

Multiple Inheritance

Multiple inheritance

- An object can have two or more different parent classes and inherit both data and behavior from each.

- **C++** supports multiple implementation inheritance.
  - Both interfaces and implementation are inherited from the base class.

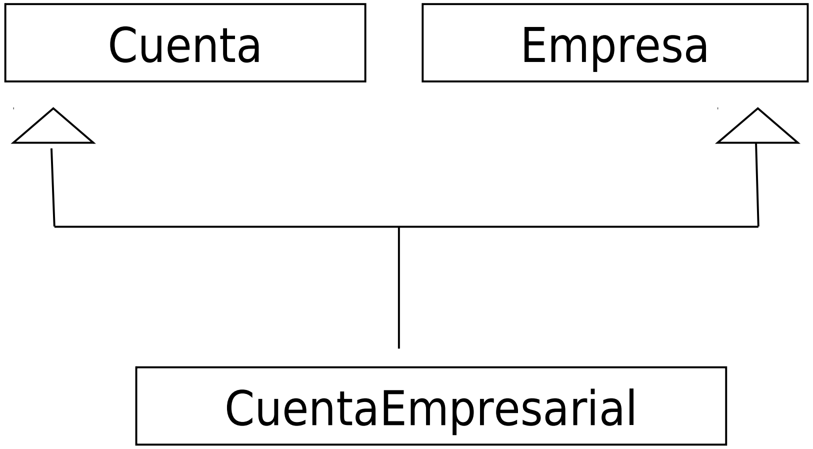
- **Java** only supports multiple interface inheritance.
  - Interface is inherited but not the implementation.

In these slides we will focus on multiple implementation inheritance.

Multiple implementation inheritance

(Examples in C++)

```
class Empresa {
 protected:
    string nomEmpresa;

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

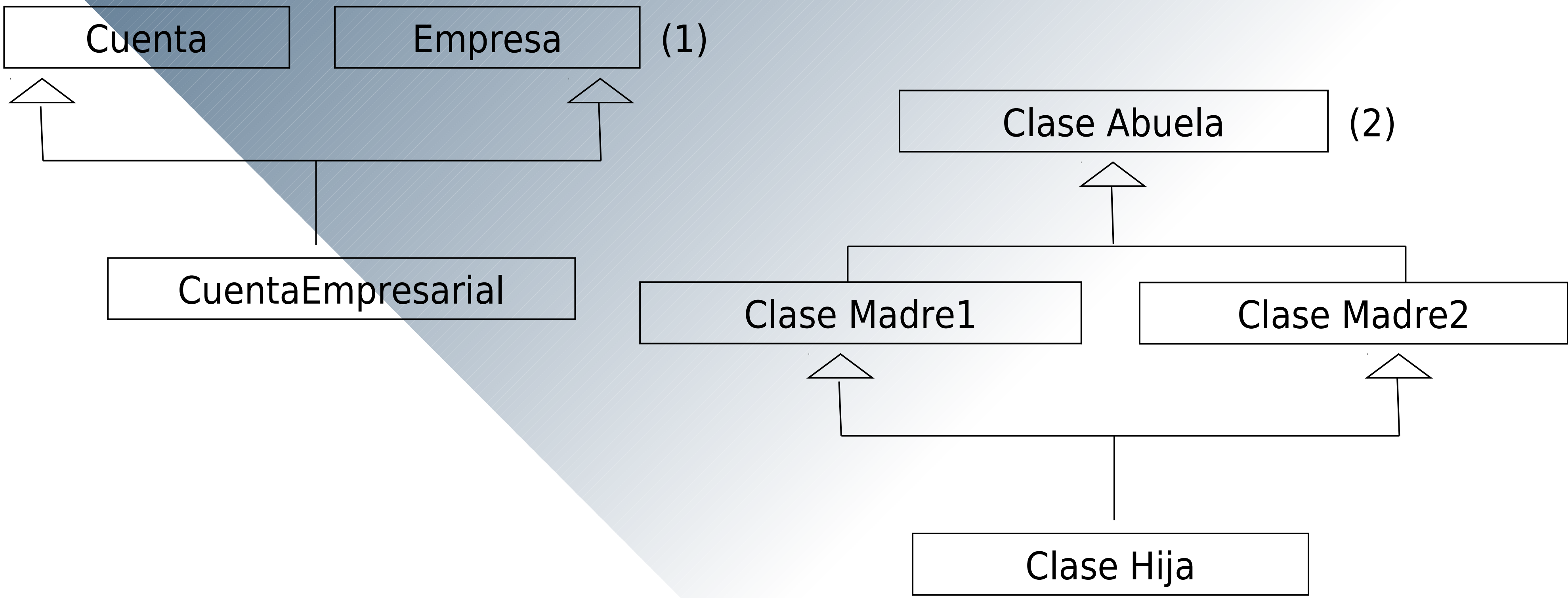


What is the implementation of CuentaEmpresarial?

Multiple implementation inheritance in 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)
     {};
};
```

Problems with multiple implementation inheritance



Which problems may arise in (1)? And in (2)?

Name clash under multiple inheritance (C++)

The problem is with the child and not with the parents.

Possible solution: use fully qualified names:

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

Property duplication under multiple inheritance

In C++ this “diamon problem” is overcome by using **virtual inheritance**:

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