SUBTOPIC 5

INHERITANCE – SECOND PART

Cristina Cachero, Pedro J. Ponce de León

Translated into English by Juan Antonio Pérez

version 20111015

INTERFACE INHERITANCE

Interface inheritance

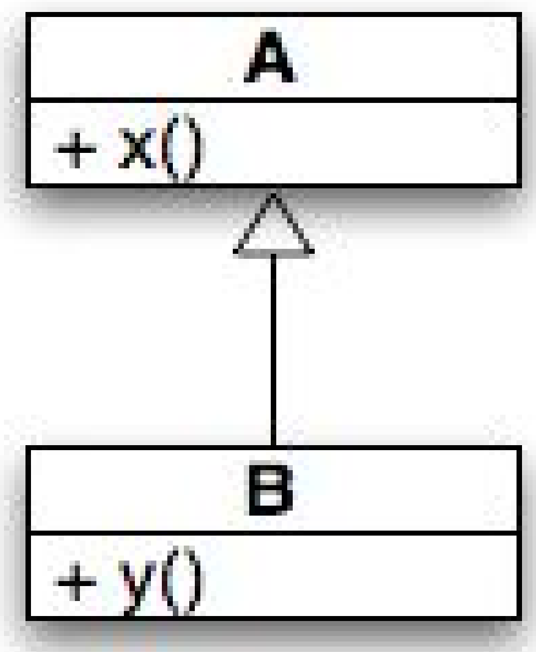
- This kind of inheritance does not allow to inherit code.
- Only the interface is inherited (sometimes with a partial or default implementation).
- Objectives
  - Decouple interface and implementation.
  - Ensure **substitution**.

Interface inheritance

- **Substitution**

```
A obj =  new B();

obj.x(); // OK
obj.y(); // ERROR
```



Principle of substitutability

One should be able to substitute an instance of a child class in a situation where an instance of the parent class is expected (Liskov, 1987).

The principle is valid if the two classes are subtypes of each other, but not necessarily in general.

Subtype*:* A type A is said to be a subtype of type B if an instance of type A can be substituted for an instance of type B with no observable effect.

Principle of substitutability

- All OOL support subtypes.
  - Strongly typed languages (usually, statically typed)
    - Objects are characterized by their class. The type of a expression is determined at compile time.
  - Weakly typed languages (dynamically typed)
    - Objects are characterized by their behavior.

Strongly typed language:

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

Weakly typed language:

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

Principle of substitutability

- **Java**: always

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

class Panadero
  extends Dependiente
{...}

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

- **C++**: subtypes must use pointers or references

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

class Panadero
  : public Dependiente
{...}

Panadero p;
Dependiente& d1=p; // substit.
Dependiente* d2=&p; // substit.
Dependiente d3=p;
// NO substit.: object slicing
```

INTERFACE INHERITANCE

- **Objectives:**
  - Concept reuse (interface)
  - Ensure that the principle of substitutability holds.

- Implemented by means of **interfaces** (Java/C#) or **abstract classes** (C++) and **dynamic binding**.

INTERFACE INHERITANCE
Binding time

- The time at which the piece of code (method) to execute for a particular message is identified, or the time at which the type of object associated to a variable is determined. The time at which binding takes place.

  - **Early or static binding:** at compile time

    Advantage: EFFICIENCY

    - **Late or dynamic binding:** at run time

    Advantage: FLEXIBILITY

INTERFACE INHERITANCE
Binding time

- Object binding time
  - **Static binding:** the type of object stored in a variable is determined at run time.

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

  - **Dynamic binding:** the type of object stored in a variable is not predefined; therefore, the system will manage the variable differently according to the real nature of the object at run time.
    - Some languages (e.g. Smalltalk) always use dynamic binding with variables.
    - Java uses dynamic binding with objects and static binding with scalar types.

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

    - C++ only supports dynamic binding for objects when declared as pointers or references and when inheritance hierarchies exist.

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

INTERFACE INHERITANCE
Binding time

- <u>Method binding time</u>
  - **Static binding:** choice of the method responsible of replying a message is done at compile time, depending on the type of the receiver object at compile time.

```
//C++ uses static binding by default
CuentaJoven tcj;
Cuenta tc;

tc=tcj; // object slicing
tc.abonarInteresMensual();
// Static binding: Cuenta::abonarInteresMensual()
```

  - **Dynamic binding:** choice of the method responsible of replying a message is done at run time, depending on the type of the receiver object at run time.

```
//Java uses dynamic binding by default
Cuenta tc = new CuentaJoven(); // substitution

tc.abonarInteresMensual();
// Dynamic binding: CuentaJoven.abonarInteresMensual()
```

INTERFACE INHERITANCE
Dynamic binding in Java

```
class Cuenta {

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


class CuentaJoven extends Cuenta {

  void abonarInteresMensual() { // dynamic binding by default

      if (getSaldo()>=10000) super.abonarInteresMensual();

  }

...}
```

- The derived class **overrides** the behavior of the base class.

- The intention is that overridden methods could be invoked from references to objects of the base class (exploiting the principle of substitutability).

```
Cuenta tc = new CuentaJoven(); // substitution

tc.abonarInteresMensual();
// Dynamic binding: CuentaJoven.abonarInteresMensual()
```

INTERFACE INHERITANCE
Dynamic binding in C++

C++ requires that:

- The method is declared as **virtual** by placing the keyword `virtual` in the parent class, thus indicating that overriding may take place, although not necessarily.
- The derived class overrides the method in the parent class.

```
class Cuenta {
...
  virtual void abonarInteresesMensual();
  // When using inheritance in C++, it is recommended
  // that destructors in the base class are declared
  // as virtual.
  virtual ~Cuenta();
};
```

```
Cuenta* tc = new CuentaJoven();

tc->abonarInteresesMensual();
// Dynamic binding: CuentaJoven::abonarInteresesMensual()
delete tc; // CuentaJoven::~CuentaJoven();
```

INTERFACE INHERITANCE
Abstract classes

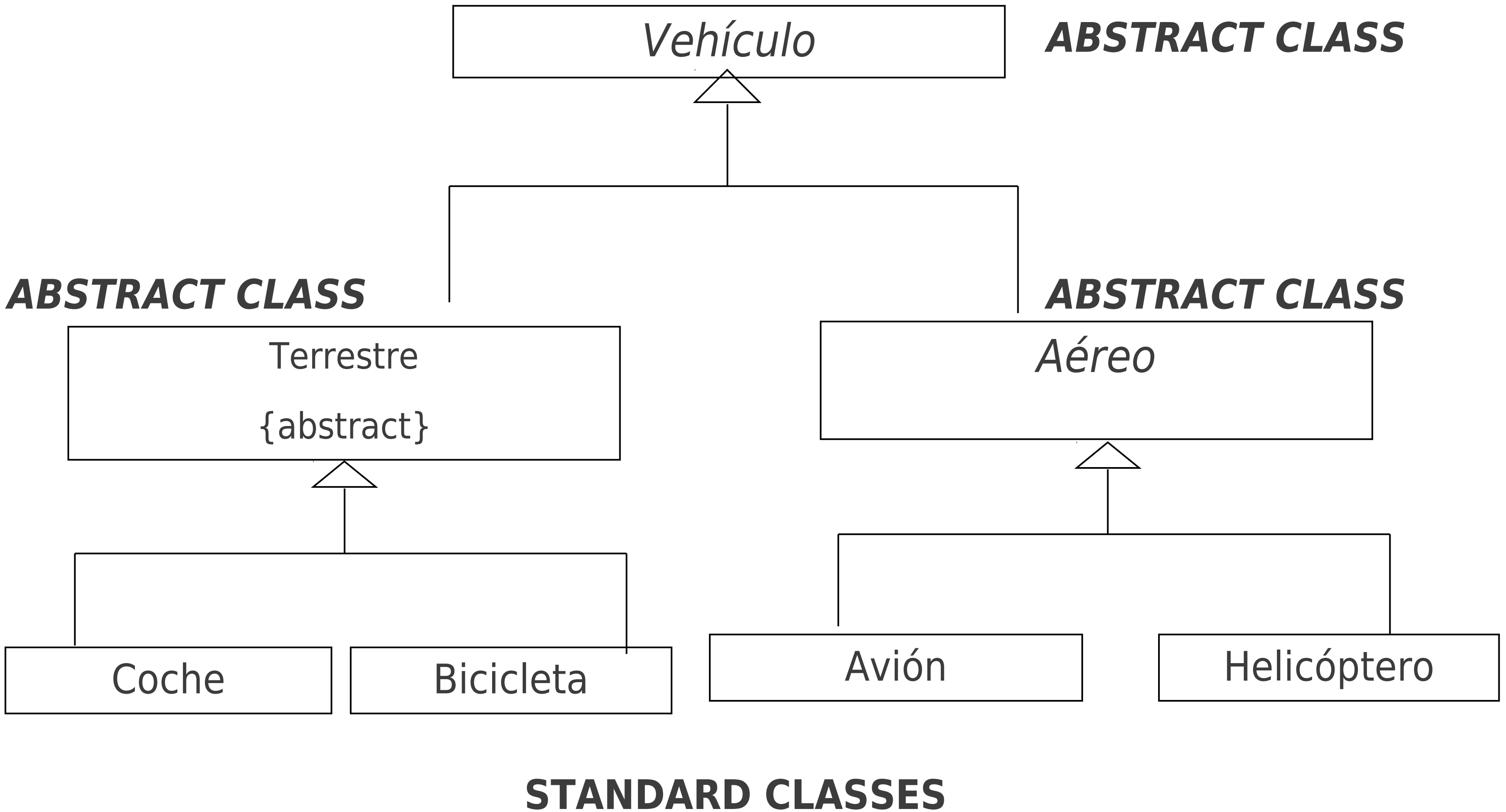
- A class that is not used to make direct instances but rather is used only as a base from which other classes inherit.
- In C++ the term is reserved for classes that contain at least one **abstract method** (pure virtual methods). In Java it refers to a class explicitly declared as `abstract`.
- Abstract methods obviously use dynamic binding.
- References to the abstract class will always point to objects of the derived classes.

INTERFACE INHERITANCE

- **Abstract classes**
  - Abstract methods (Java/C++) must be overridden by subclasses before creating instances (otherwise, they will be abstract as well).
  - Derived classes <u>implement the interface </u>of the abstract class.
    - The principle of substitutability is guaranteed.

INTERFACE INHERITANCE

UML notation for abstract classes (italicized text).



INTERFACE INHERITANCE

- Abstract classes in Java

```
abstract class {
  ...
  abstract <return type> method (<arg list>);
}
```

Abstract class

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

Derived class

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

INTERFACE INHERITANCE

- Abstract classes in C++
  - They contain at least one **pure virtual method** (abstract method):

```
virtual <return type> method (<arg list>) = 0;
```

Abstract class

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

Derived class

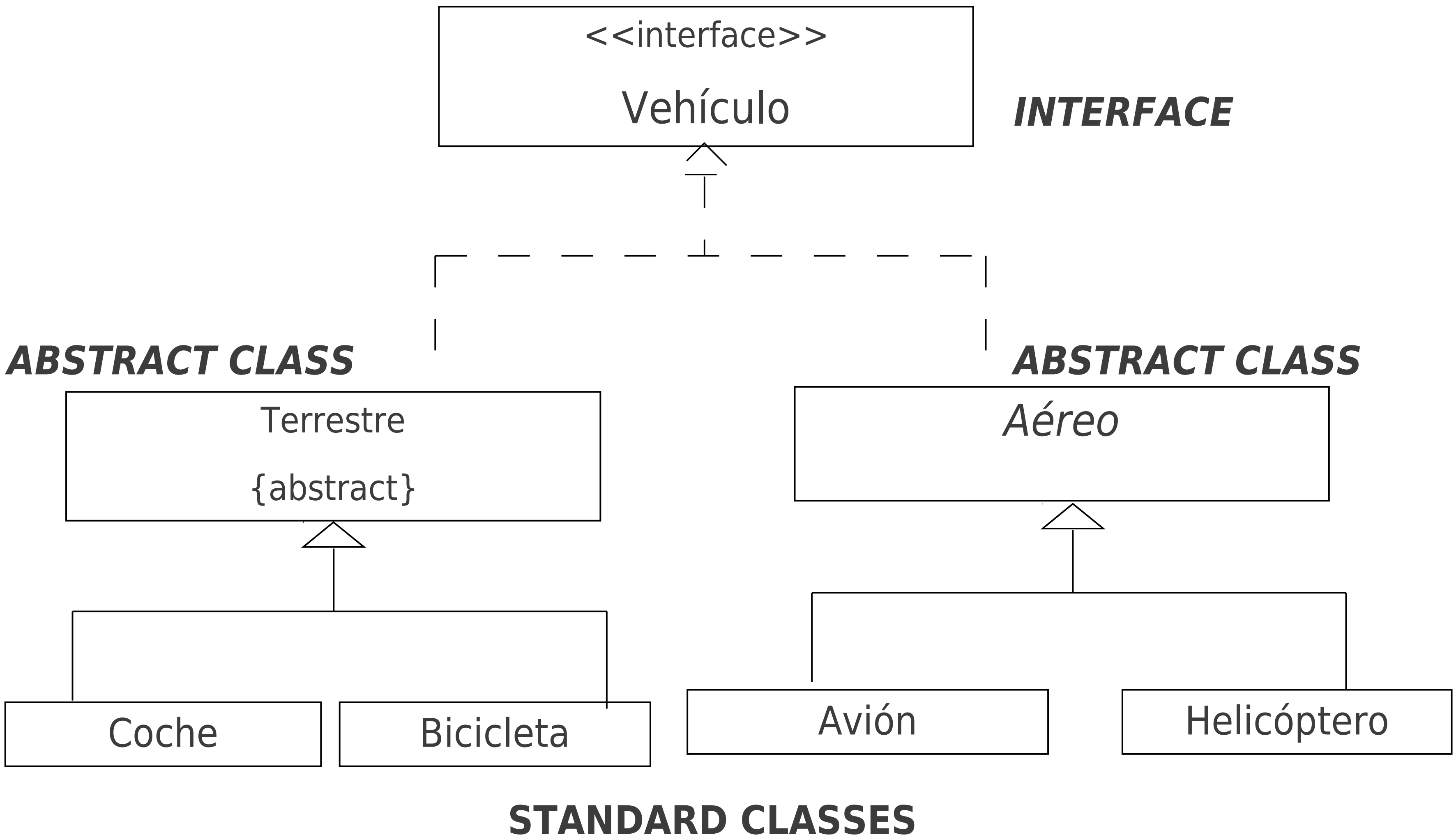
```
class Circulo : public Forma
{
  int radio;
  public:
    void dibujar() {...};
    ...
}
```

INTERFACE INHERITANCE

- **Interface**
  - Declaration of a set of abstract methods.
  - Interface and implementation are separated.
  - Java/C#: explicit declaration of interfaces
    - A class may implement more than one interface (multiple interface inheritance).

INTERFACE INHERITANCE

UML notation for interfaces.



INTERFACE INHERITANCE

- **Interfaces in Java**

```
interface Forma
{
  // - All methods are abstract implicitly.
  // - Public visibility.
  // - No instance attributes, only static constants:
  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()
    {...}
}
```

INTERFACE INHERITANCE

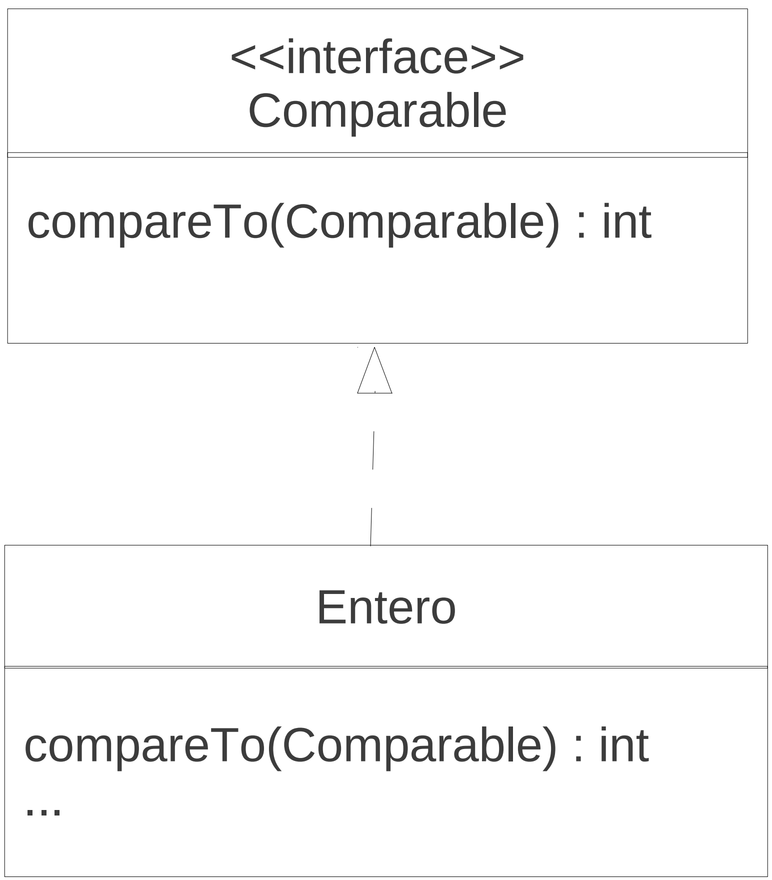
- **Interfaces in C**++: public inheritance of abstract classes

```
class Forma
{
  // - No instance attributes
  // - Only static constants allowed
  // - All methods declared abstract
 public:
  virtual void dibujar()=0;
  virtual int getPosicionX()=0;
  // rest of pure virtual methods...
}
```

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

INTERFACE INHERITANCE

- Interface example (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;
  }
}
```

INTERFACE INHERITANCE

- Interface example (Java, cont.):

```
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]; }
}
```

```
// Client code
// Upcasting
ParOrdenado po = new ParOrdenado(new Entero(7), new Entero(3));

po.getMenor(); // 3
po.getMayor(); // 7
```

INTERFACE INHERITANCE

- Multiple interface inheritance (Java)

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

interface CanSwim {
  void swim();
}

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

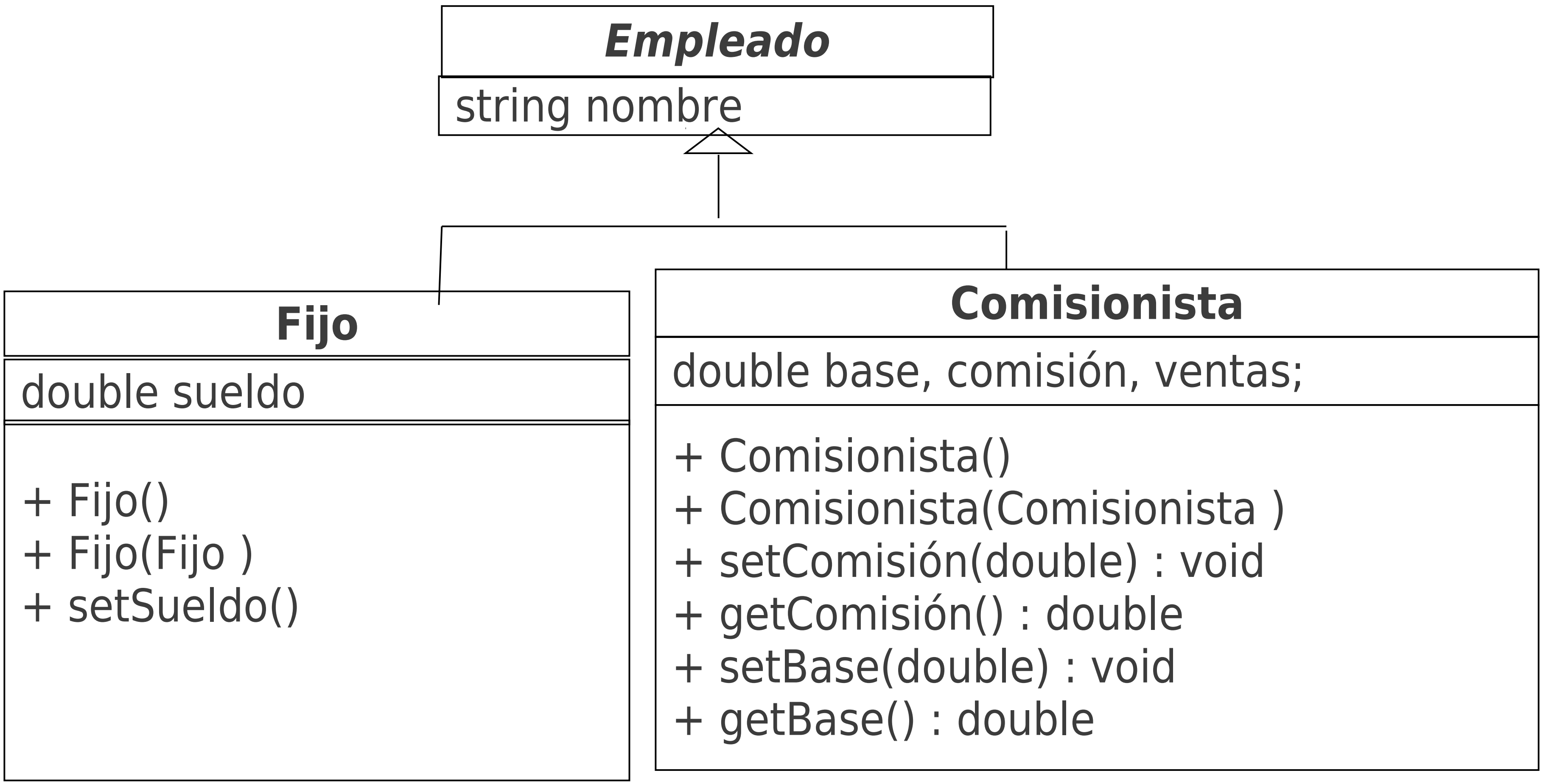
```
class ActionCharacter {
  public void fight() {}
}

class Hero extends ActionCharacter
    implements CanFight, CanSwim, CanFly {
  public void swim() {}
  public void fly() {}
  // definition of fight comes from inheritance
}

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 (upcasting)
    u(h); // Treat it as a CanSwim
  }
}
```

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

Homework: paying wages



Homework: paying wages

Implement these classes by adding a method getSalario() which in the case of a permanent employee returns the salary and in the case of the commission agent (Comisionista) returns the base salary plus the commission; after that, next code will obtain the salary regardless of the type of the employee.

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

IMPLEMENTATION INHERITANCE

Safe use

Implementation inheritance

- Inheritance: property by which instances of a class inherit part or the whole implementation of another class.
- It must be used carefully.

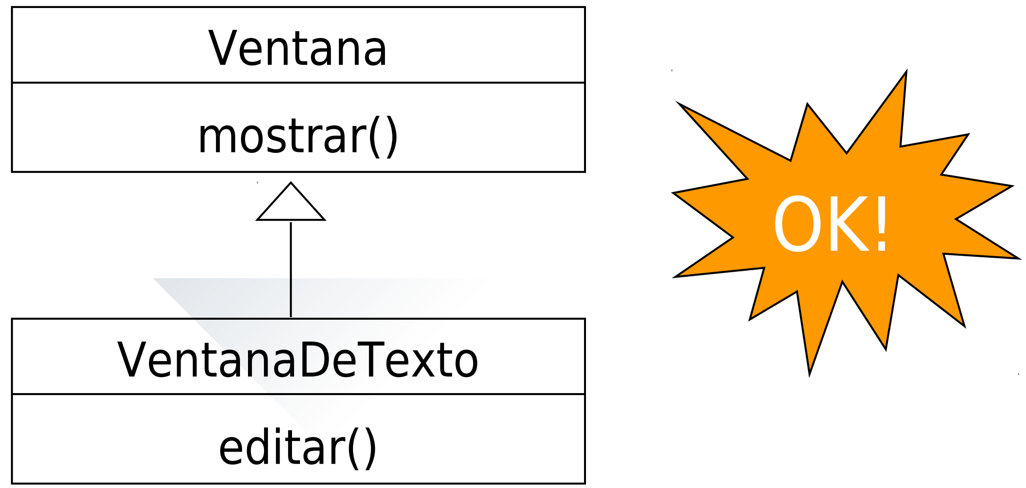
Safe use of implementation inheritance

- Behavior and data associated with child classes are an **extension** (a larger set) of the properties associated with parent classes.
- On the other hand, since a child class is a more specialized (or restricted) form of the parent class, it is also a kind of **contraction** of the parent type.
- This tension between inheritance as an expansion and inheritance as a contraction is a source for much of the power in the technique, but also for much confusion as to its proper employment.

- Method overriding should only be used, in principle, to make properties more specific:
  - Constraining restrictions
  - Extending functionality

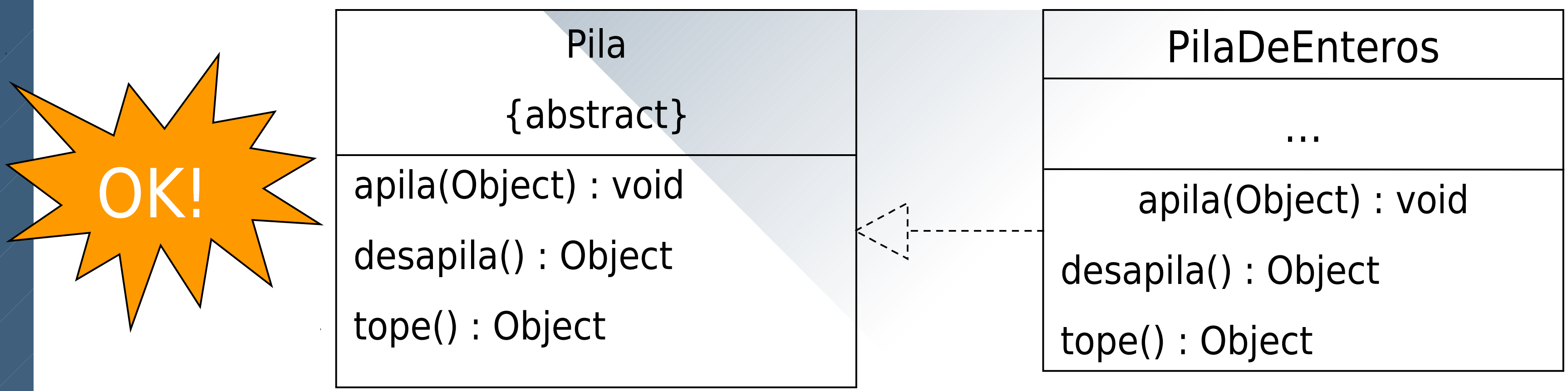
Safe use of implementation inheritance

- **Specialization (subtyping)**
  - The new class is a specialized form of the parent class.
  - It may add a new behavior, but satisfies the specifications of the parent in all relevant respects.
    - The principle of substitutability holds (subtyping)



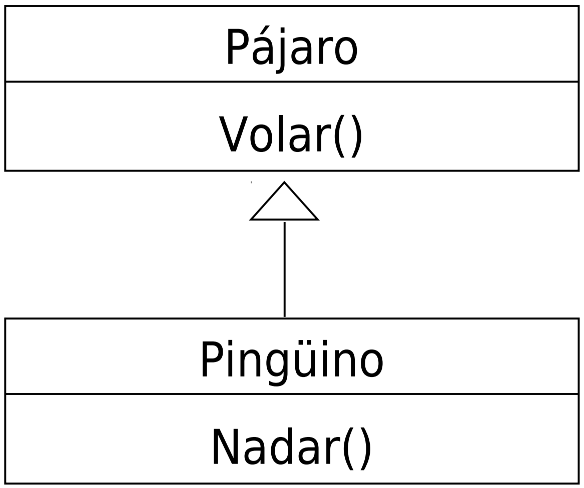
Safe use of implementation inheritance

- **Specification**
  - The parent class (an abstract class or an interface) can be a combination of implemented operations and operations that are deferred to the child classes.
    - There is no interface change.
    - The child merely implements behavior described, but not implemented, in the parent.



Unsafe use of implementation inheritance

- **Restriction (limitation)**



The behavior of the subclass is smaller or more restrictive than the behavior in the parent class.

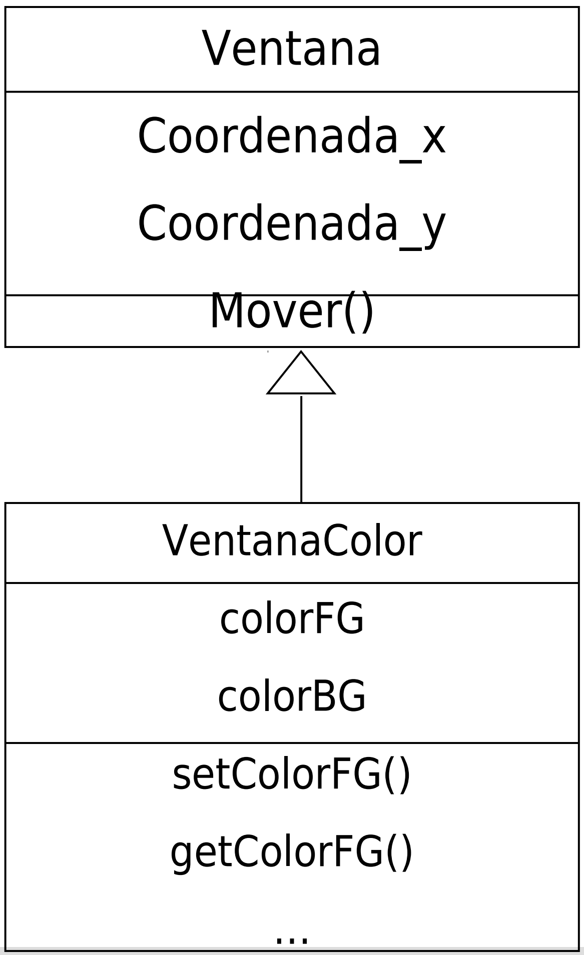
Part of the base class is not useful.

Usually, the base class cannot be modified.

It is an explicit contravention of the principle of substitutability (no subtyping) and *it should be avoided* (a penguin does not fly).

Unsafe use of implementation inheritance

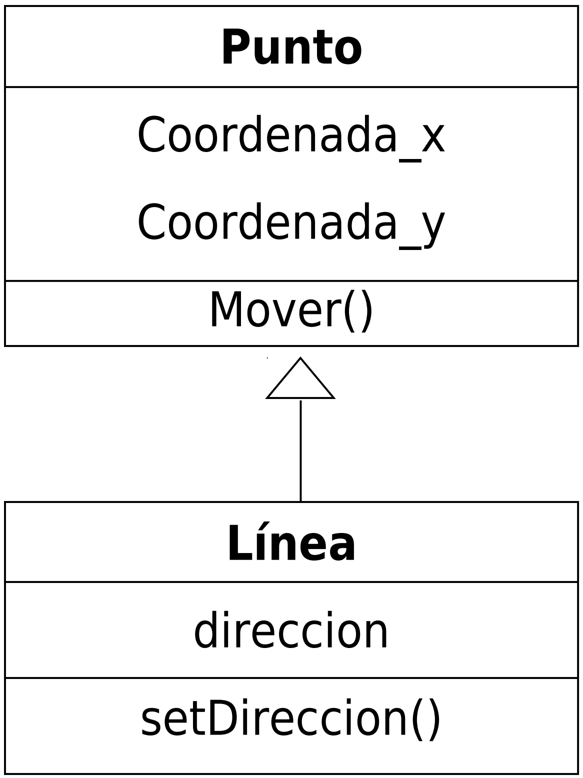
Generalization



- A subclass extends the behavior of a parent class to create a more general kind of object.
- Often applicable when the base class cannot be modified.
- It is recommended to invert the type hierarchy and use subclassing for specialization.

Unsafe use of implementation inheritance

- **Variance (convenience inheritance)**



Two or more classes have similar implementations but there is no hierarchical (“is-a”) relationship between the abstract concepts represented by the classes.

WRONG!

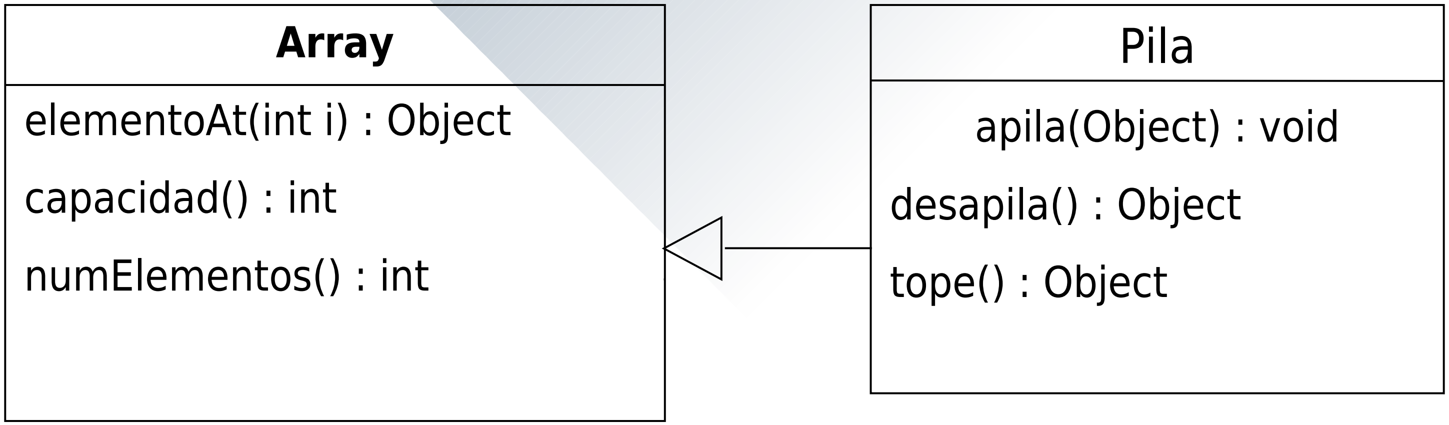
Solution: factor out the common code into an abstract class (e.g., Tablet and Mouse → PointingDevice)

Construction inheritance

- ***A.k.a. Pure implementation inheritance***

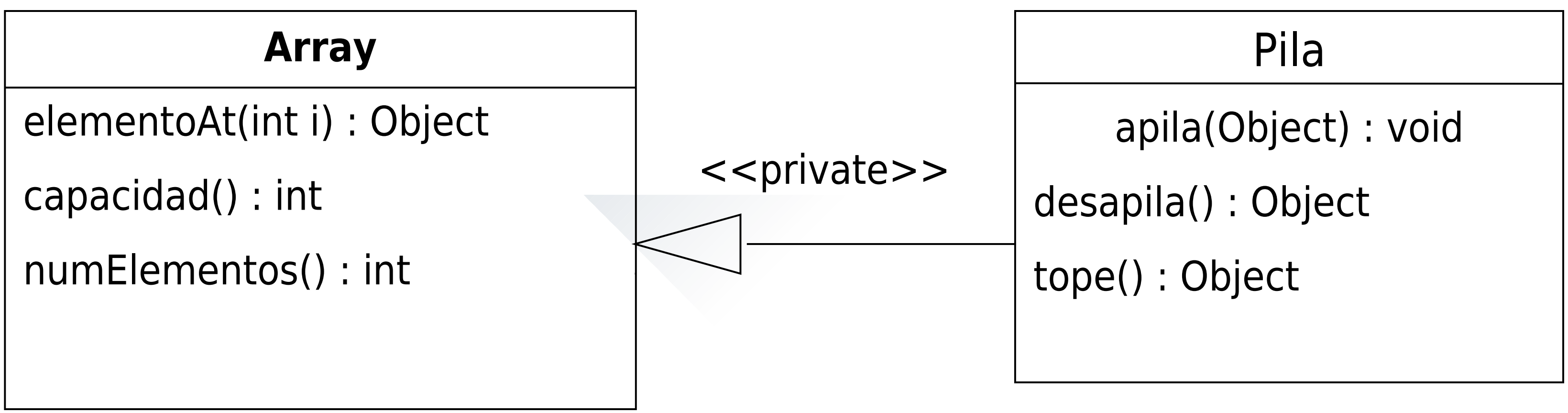
- A class inherits almost all of its desired functionality from a parent class, perhaps changing only the names of methods or modifying the arguments in a certain fashion.

- The derived class is not a specialization of the base class.
  - No intention to follow the principle of substitutability.
  - E.g., a stack can be built from an array.



Construction inheritance in C++

- Private inheritance in C++ permits subclassing for construction without breaking the principle of subclassing for specialization.:
    - The interface of Array will not be public in Pila.
        - Use composition if possible.



INHERITANCE

The benefits and costs of inheritance

The benefits of inheritance

- Software reusability
- Code sharing
- Consistency of interface
- Software components
- Rapid prototyping
- Polymorphism
- Information hiding

[BUDD] 8.8

The costs of inheritance

- Execution speed
- Program size
- Message-passing overhead
- Program complexity

[BUDD] 8.9

INHERITANCE

Mechanisms for software reuse

Choice of the reuse mechanism
Introduction

- Inheritance (IS-A) and Composition (HAS-A) are the two most common mechanisms for software reuse
  - COMPOSITION (a.k.a. Layering): Has-a relationship: BETWEEN OBJECTS.
    - Composition means that the new component will <u>contain</u> an instance of the already implemented object
    - Example: a car has an engine.

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

  - INHERITANCE: Is-a relationship: BETWEEN CLASSES
    - Inheritance means “<u>containing</u>” a class
    - Example: a car is a vehicle

```
class Coche extends Vehiculo{
        …
}
```

Inheritance vs. Whole/Part

- **Rule of change:** do not use inheritance to describe a perceived “is-a” relationship if the corresponding object components may have to be changed at run time.
  - Because the inheritance relationship holds between classes, it is not possible to modify the object relationship dynamically; composition holds between objects, thus this change is easier.

- **Polymorphism rule:** inheritance is appropriate to describe a perceived “is-a” relation if entities or data structure components of the more general type may need to become attached to objects of the more specialized type (polymorphic effects).

“Object-Oriented Software Construction”, B. Meyer

- Example: construction of a set abstraction by using an existing class, Lista, which maintains a list of integer values.

- We want the set to perform operations such as adding a value to the set, determining the number of elements, and determining whether a specific value occurs in the set.

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

Choice of the reuse mechanism
Composition (Layering)

- When composition is <u>used</u>, a portion of the state of the new data structure is simply an instance of the existing structure.

```
class Conjunto {

        public Conjunto() { losDatos = new Lista(); }
        public int size(){ return losDatos.size(); }
        public int includes (int el){return losDatos.includes(el);};
        // a value cannot be in the set twice
        public void add (int el){
          if (!includes(el)) losDatos.add(el);
        }

    private Lista losDatos;
}
```

Choice of the reuse mechanism
Composition (Layering)

- Composition makes no explicit or implicit claims about substitutability. When formed in this fashion, the data types Conjunto and Lista are entirely distinct, and neither can be substituted in situations where the other is required.

Choice of the reuse mechanism
Inheritance

- With inheritance all data areas and functions associated with the original class Lista are automatically associated with the new data abstraction Conjunto.

```
class Conjunto extends Lista {
    public Conjunto() { super(); }
    // a value cannot be in the set twice
    void add (int el){ // refinement
          if (!includes(el)) super.add(el);
        }
}
```

- All operations associated with lists are immediately applicable to sets as well; some of them may be used out of the box, whereas with composition all of them may be redefined.
  - The new class does not define any new data fields.

Choice of the reuse mechanism
Inheritance

- Inheritance carries an implicit assumption that subclasses are, in fact, subtypes.

  - In this case, a set IS NOT a list.

  - It appears that, *in this case*, composition is the better approach.

Choice of the reuse mechanism Composition vs. Inheritance

- Composition is the simpler of the two techniques.
  - It indicates more clearly what operations can be performed on a particular data structure, regardless of the interface of the part object.

- With composition it is not necessary to use methods in the existing class which are not relevant to the new class.
- It would be easy to reimplement the class Conjunto to use a different technique (such as a hash table) with minimal impact on the users of the Conjunto abstraction.

Choice of the reuse mechanism Composition vs. Inheritance

- (Public) inheritance carries an implicit assumption that subclasses are, in fact, subtypes.
  - Implementations are shorter in code.
  - If a new method is added to the base class, it will be immediately available to the derived classes.

- Disadvantages
  - Inheritance does not prevent users from manipulating the new structure using methods from the parent class, even if these are not appropriate.
  - Yo-yo problem when a programmer tries to understand a child class.
  - A change in the base class may cause a lot of problems to users of the derived classes.

Bibliography

- Bruce Eckel. ***Piensa en Java, 4th edition***
  - Ch. 7 y 9

- Timothy Budd. ***An Introduction to object-oriented programming, 3rd ed.***
  - Ch. 8-13.

- C. Cachero et al. ***Introducción a la programación orientada a objetos***
  - Ch. 3 (examples in C++)