PROG-3

SUBTOPIC 2

OBJECTS, CLASSES AND RELATIONSHIPS

Version 0.1

(Academic Year 2011-2012)

Pedro J. Ponce de León

Translated into English by Juan Antonio Pérez

lsi Departamento de Lenguajes y Sistemas Informáticos

Universitat d'Alacant
Universidad de Alicante

Contents

- Objects
- Classes
- Attributes
- Operations
- UML notation
- Relationships
  - Association
  - Whole/part
  - Using
- Metaclasses

Objects
Definition

- Anything that could be assigned some **properties** and **behavior** could be an object.

- From the perspective of a software analyst: an object represents an entity (real or abstract) with a well defined role in the problem domain.

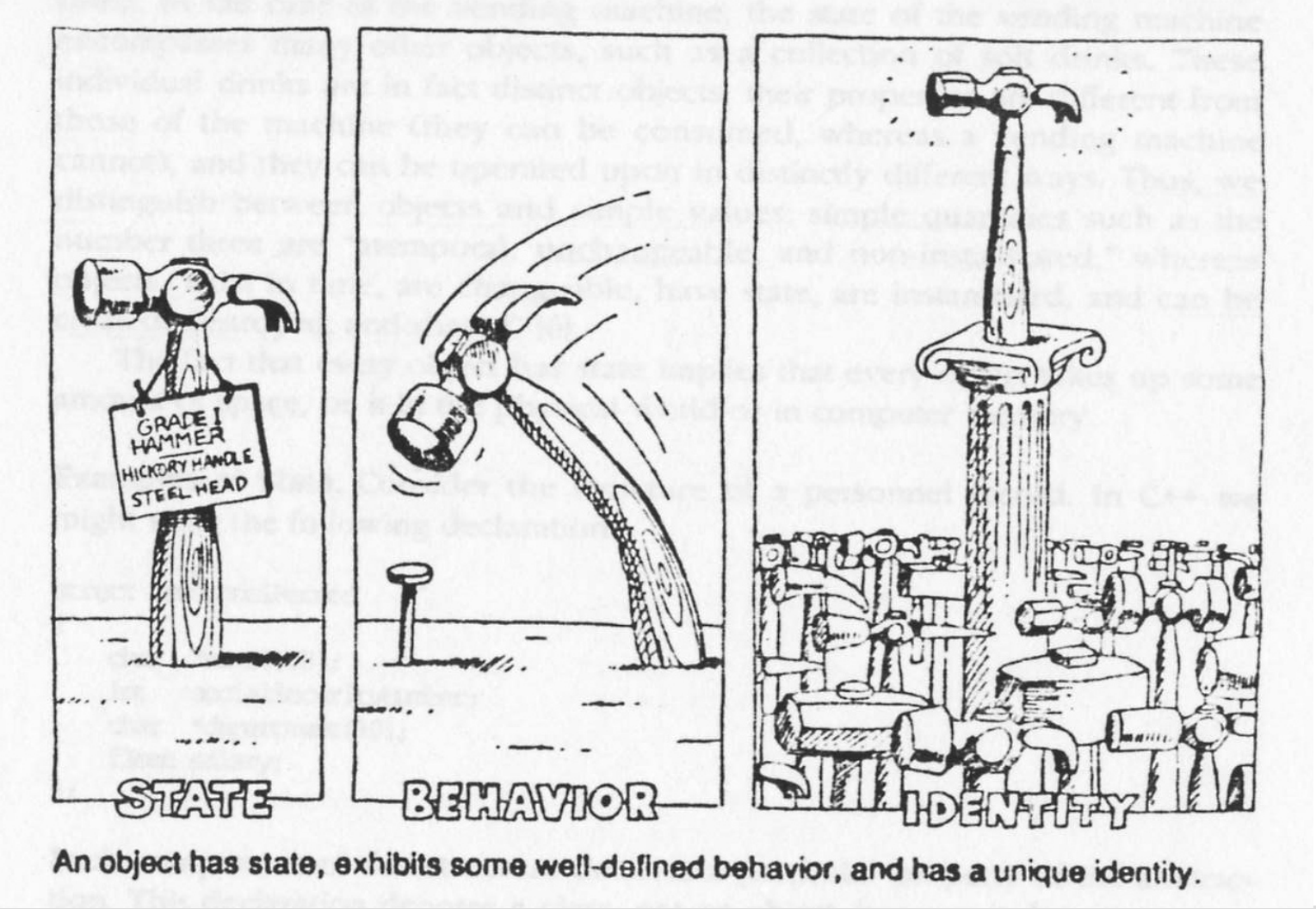
- From the perspective of a programmer: an object is a data structure upon which a set of operations may be executed.

Objects
Definition

- According *to Grady Booch, an object has state, behavior, and identity*:
  - The **state** of an object encompasses all of the (usually static) properties of the object plus the current (usually dynamic) values of each of these properties.
  - **Behavior** is how an object acts and reacts, in terms of its state changes and message passing.
  - **Identity** is that property of an object which distinguishes it from all other objects.

Objects
Definition

- Definition by Booch:



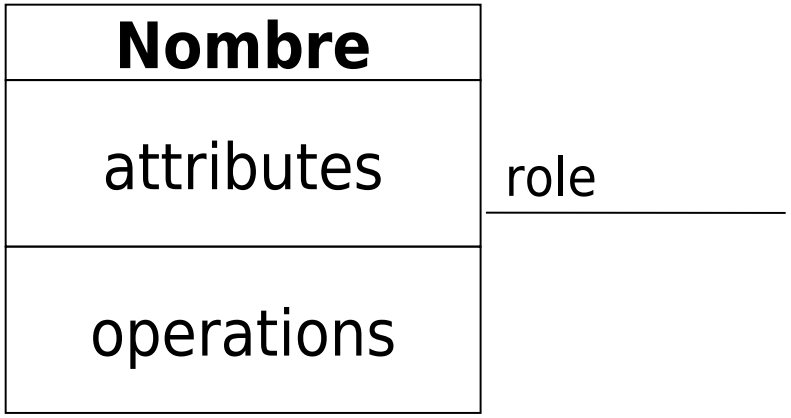
An object has state, exhibits some well-defined behavior, and has a unique identity.

Classes
Definition

- **Class:** abstraction of the set of attributes, operations, relationships and semantics of a set of objects.
- Therefore, a class is a set of objects that share a common structure and a common behavior. A single object is simply an **instance** of a class.

Classes
Elements in the definition

- **Class identifier**: name
- **Properties**
  - **Attributes or variables**: data needed to describe objects (instances).
    - The state of an object encompasses all of its attributes and their current values.
  - **Role**: the purpose wherein one class or object participates in a relationship with another; the role of an object denotes the selection of a set of behaviors that are well-defined at a single point in time.
  - **Operations, methods, services**: some work that one object performs upon another in order to elicit a reaction.




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

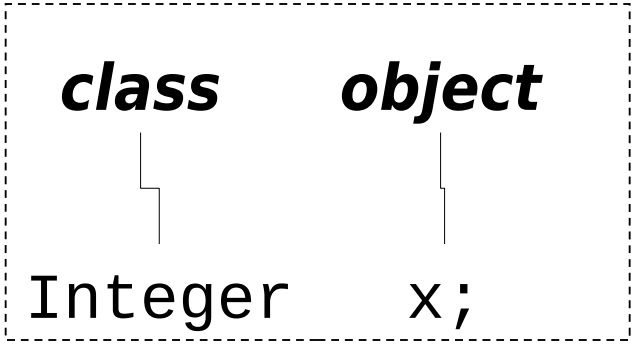
Objects or classes?

- Movie **CLASS**
  - Roll of film **CLASS**
- Roll of film with S/N 123456 **OBJECT**
- Tickets for ‘Super 8’ showing at Muchavista cinema at 19:30pm **OBJECT**

In general:
- One particular 'thing' will be a class if it may have instances.
- One particular 'thing' will be an object if it is something unique sharing characteristics with similar 'things'

Objects
Objects and classes in programming languages

- Class ⟺ Type
- Object ⟺ Variable or constant



Class: characterization of a set of objects that share a common structure and a common behavior.

Attributes Definition

- **Attribute** (or data member, or instance variable)
  - Data which is part of an object.
    - They are usually objects.
    - Declared as ‘fields’ of the class.
  - ***Visibility*** *of an attribute*
    - *Visibility affects from where an attribute may be accessed.*

+ Public (interface)	-> visible from everywhere
- Private (implementation)	-> from the same class only
# Protected (implementation)	-> from derivative classes
~ Package level (no modifier in Java)	-> from classes in the same package

Attributes are usually part of the implementation (hidden elements) of a class, since they constitute the state of an object.

Attributes
Classification of attributes

- **Constants / Variables**
  - Constant: e.g., a house is built with a specific number of rooms (stable feature):
    - `private final int numHab;`
  - Variable: e.g., salary of a worker may change in time:
    - `private int sueldo;`
- **Instance / class attributes**
  - Instance attributes**:** memory space is allocated for them with every object created:
    - `private String nombre; // name of an Employee`
  - Class attributes: class features common to every instance of the class:
    - `private static String formatoFecha; // in class Fecha`

Attributes
Class attributes

- A class variable is shared by all instances of the same class.
  - **They are used for:**
    - **Holding (constant) common features to all the objects of the class**
      - Number of wheels in a bicycle
      - Number of legs of a spider
    - **Holding features which depend on all the objects of the class**
      - Number of students in a university
  - A class attribute (static attribute in C++ or Java) can be referenced from any object of the class, as it is a data member of the class.

Operations Definition

- **Operation** (member function, method or class service)
  - Action performed by an object in response to a message. Methods define the behavior to be exhibited by instances of the associated class at program run time.
  - Each operation has a visibility (like attributes)
  - ***Instance / class methods*** (like attributes)
  - They can alter (**modifier** method) or not (**selector** method) the state of an object
  - **The signature of an operation** consists of its qualified name, and the number and types of its arguments and return values:

```
TipoRetorno
NombreClase.NombreFuncionMiembro(parametros)
```

Operations
Classification of operations

- **Instance /class operations**
  - ***Instance operations***:
    - Operations that can be performed by objects of a class.
    - They can access directly both instance and class attributes.
    - They usually act upon the object which receives the message.

```
Circulo c = new Circulo();

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

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

Operations
Classification of operations

- **Instance / class operations**
  - ***Class operations***:
    - Class methods are not permitted to access instance variables; they can access only class variables
    - They can be invoked independently of receivers (using explicit name qualification), unless it is explicitly passed as argument.

```
class Circulo {

    private static final double pi=3.141592;

    public static double getRazonRadioPerimetro()
    {
      return 2*pi;
    }
…
};
```

Operations
Classification of operations

- **Update, mutator or modifier methods**
  - They change the state of the receiver object.

```
c.setRadio(3); // modifies ‘c’ radius
```

- **Accessor or selector methods**
  - They provide the sole means for the state of an object to be accessed (retrieved without change) from other parts of a program.

```
c.getRadio(); // gets ‘c’ radius
```

Operation overloading

- Some programming languages (Java, C#, C++) support operation overloading.
  - This allows the creation of several methods (visible in the same scope) with the same name which differ from each other in the number and/or type of its arguments.

```
class Circulo {
    // no fill
    public void pintar() {...}
    // filled with a color
    public void pintar(Color) {...}
}
```

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

Operations
Reference to the receiver object

- In most OOL, the receiver of an instance method is an implicit argument.
- A *pseudo-variable* is used to obtain a reference to it:
  - In C++ and Java, it is called **`this.`**
  - Other languages use **`self`**

- Example in Java:

receiver.selector(this ,<arguments>)

```
class Autoref {
  private int x;

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

Constructor

- An operation that creates an object and/or initializes its state
  - It is invoked every time an object is created by means of the **new** operator (in Java).
  - Because creation and initialization are invoked 'atomically', an object cannot be used before its initialization.
  - In Java and C++, constructors have the same name than the class and return no value (not even void).

```
class Circulo {

    public Circulo() {...};  // Default constructor
    public Circulo(double r) {...}; // Overloaded constructor
}
```

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

Constructor

- ***Default constructor:***
  - It is recommended to define one constructor which allows the initialization of an object with **no parameters** by setting object attributes to convenient default values.

```
public Circulo()
{
   super(); // (automatic) call to Object constructor
   radio = 1.0;
}
```

- Most languages provide a default public constructor if the programmer does not provide an explicit one.

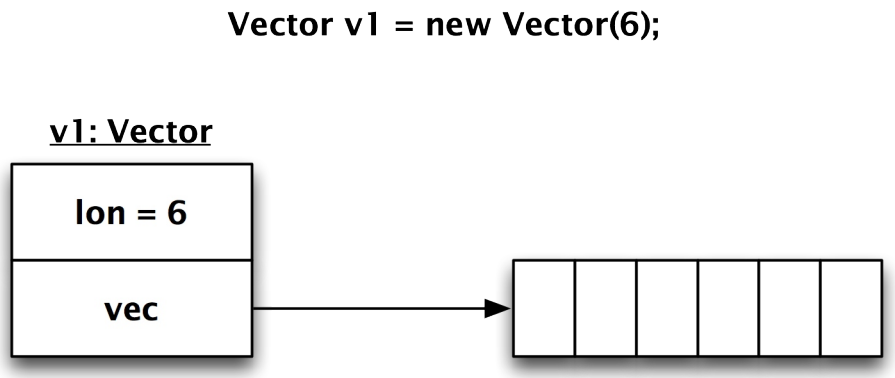
```
public Circulo()
{
  super();
  /* Java: attributes set to 0 or null */
}
```

Copying objects

- There exist two possible ways of 'copying' or 'cloning' objects:
  - Shallow copy
    - Bit-wise copy of the attributes of an object
  - Deep copy (full copy)

- Consider a Vector class:

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



Copying objects

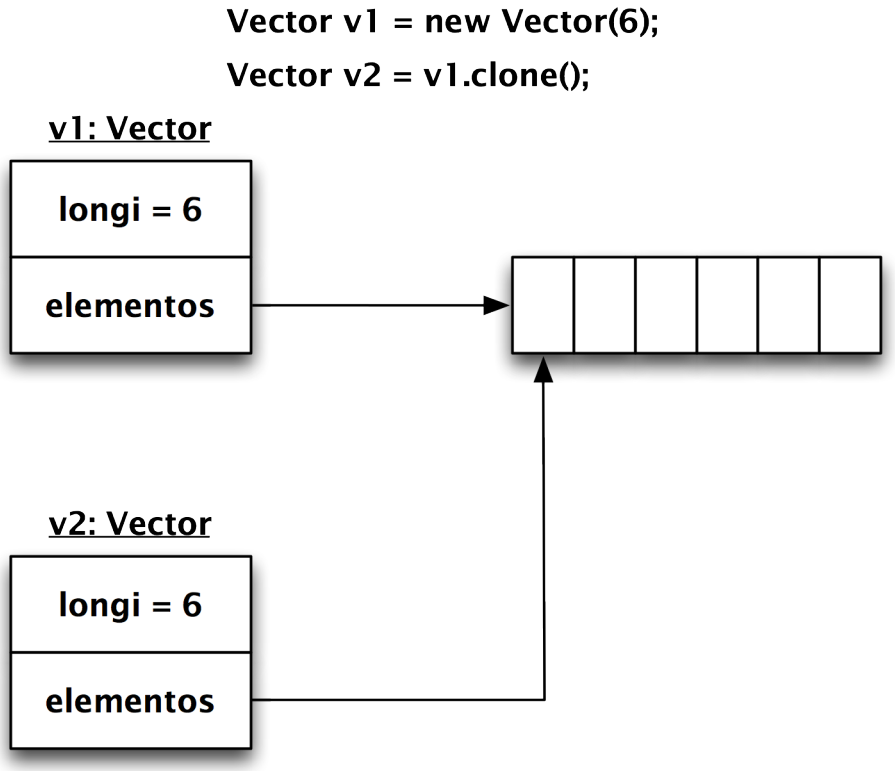
- Shallow copy
  - Default way of copying objects in C++ and Java

C++:

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

Java:

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

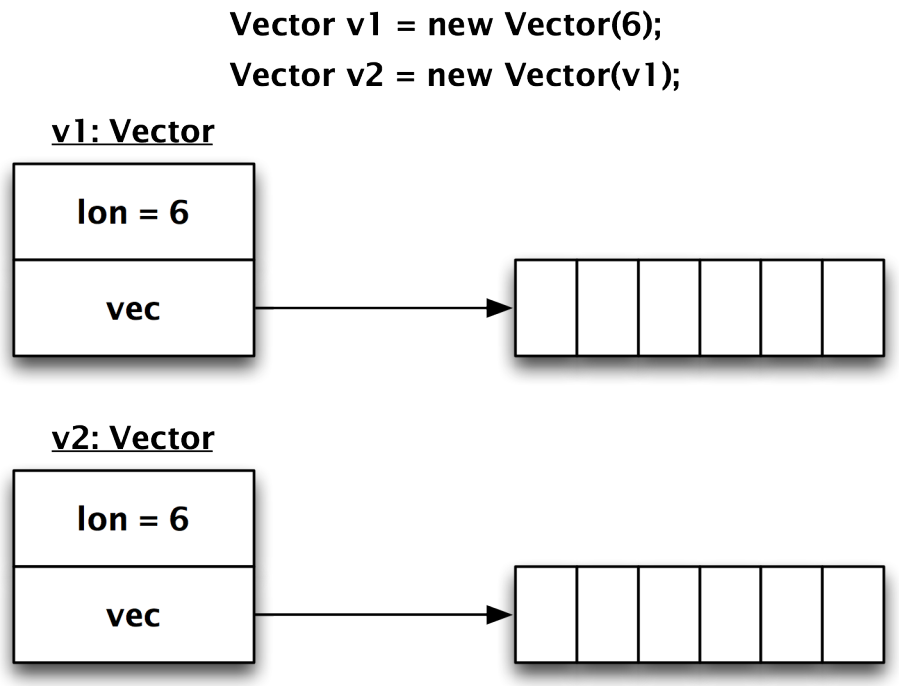


Copying objects

- Deep copy
  - It has to be implemented explicitly
    - C++: with a **copy constructor**
    - Java: with a copy ctor or with method **clone()**

Java: copy constructor

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



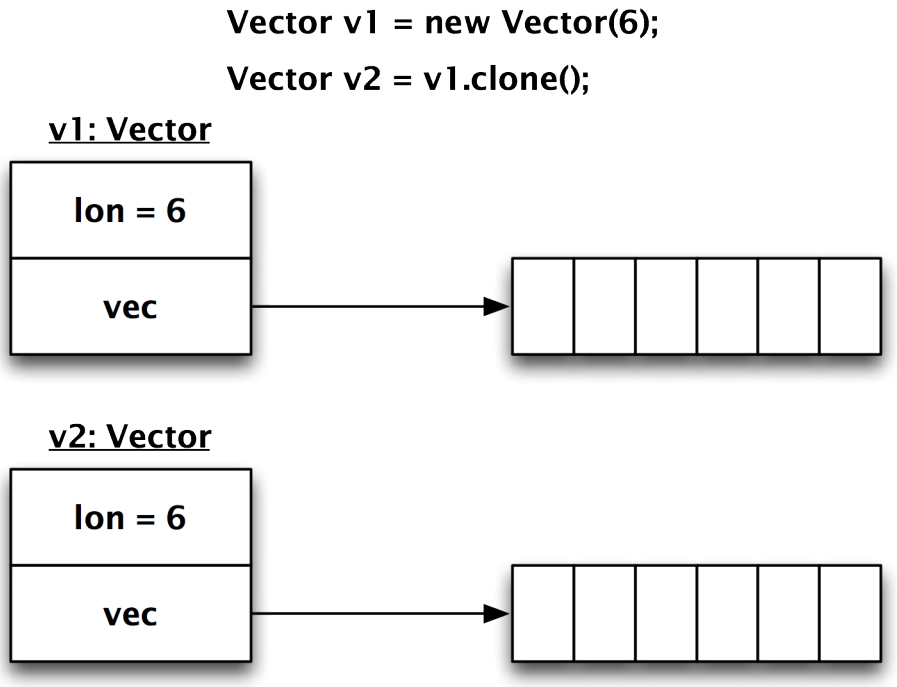
Copying objects

- Deep copy with **clone()**

  Unless the copy constructor, typically the clone() method of the superclass is also called to obtain the copy, etc. until it eventually reaches Object's clone() method. The special clone() method in the base class Object provides a standard mechanism for duplicating objects.

Java:

```
class Vector implements Cloneable
...
public Vector clone() throws
CloneNotSupportedException {

  Vector clon = (Vector)super.clone();
  clon.vec = new int[vec.length];

  for (int i=0; i<vec.length; i++)
    clon.vec[i] = vec[i];

  return clon;
}
```



Object destruction
Java: garbage collector and finalize()

- Most OOL allow the programmer to indicate when an object value is no longer being used by a program and hence can be recovered and recycled.
  - C++: destructor
  - Java: method finalize()

- In Java, the garbage collector monitors the manipulation of object values and will automatically recover memory from objects which are no longer used.

  Programmers do not control when memory is recovered, and garbage collection uses a certain amount of execution time, but it also prevents a number of common programming errors.

  Java and Eiffel allow us to define **`finalize()`** methods, which will be executed just before object destruction.

  - `protected void finalize() {} // defined in class Object`

Finalization is the opposite of initialization. It is used to to deallocate resources assigned to the object (opened files, opened database connections...).

→ finalize() methods are usually not defined.

Canonical form of a class

- Set of methods that every class *should* define. The compiler or the virtual machine usually provide them [“de oficio”], in case they are not defined in the class.

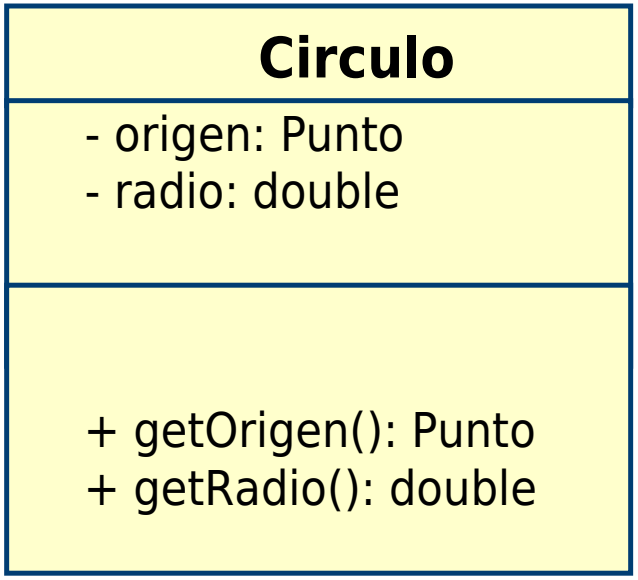
- In C++, canonical form includes:
  - Default constructor
  - Copy constructor
  - Assignment operator
  - Destructor
- In Java:
  - Default constructor
  - `public String toString()` - Representation of object state as a string
  - `public boolean equals(Object o)` – object *deep* comparison
  - `public int hashCode()` - integer hash code value for the object; if two objects are equal according to equals() they must return the same integer value.

Canonical form of a Java class

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

  public String toString() {  return ... }

  public boolean equals(Object o)
  {  ... It defines an equivalence relationship:
    Reflexive: a.equals(a) must be true.
    Symmetric: if a.equals(o) then o.equals(a).
    Transitive: if a.equals(b) and b.equals(c), then a.equals(c).
    (By default, true must be returned if the objects correspond to
a single object in memory.)
  }
  public int hashCode()
  {  ... It must follow 'equals': Two "equal" objects must
     Have the same hash code.
  }
}
```

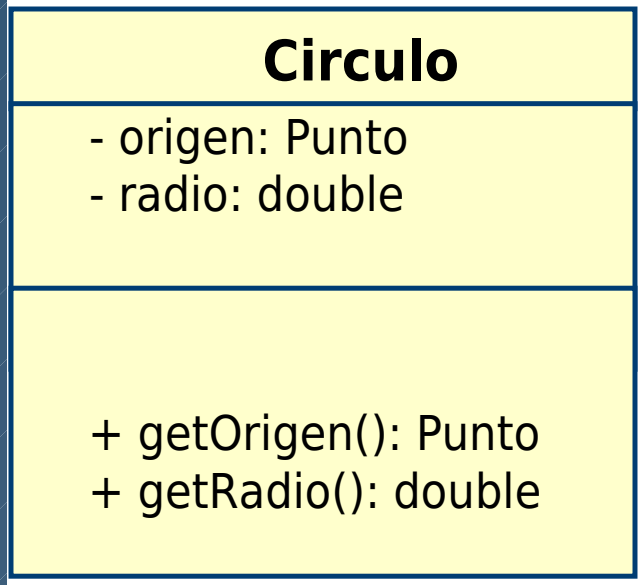
UML for classes, methods and attributes Examples in C++

- Different OOL have different syntax for the same concepts.
  - Code does not seem a good way for representing abstractions.
- UML notation integrates in a homogenized way how OO concepts are communicated and represented.




```
// C++

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

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

UML for classes, methods and attributes Examples in Java and C#




```
// Java and C#
class Circulo{
  public Punto getOrigen()
    {return origen;}
  public double getRadio()
    {return radio;}
  private Punto origen;
  private double radio;
}
```

Relationships among classes and objects

- Classes, like objects, do not exist in isolation.
- Rather, for a particular problem domain, the key abstractions are usually related in a variety of interesting ways.

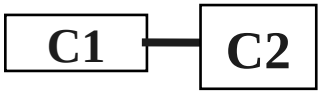
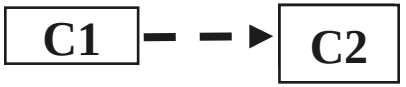
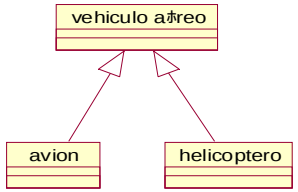
- An object oriented program is structured as a community of interacting agents, called objects. For interacting, objects must know each other.

- This knowledge is articulated by means of establishing **relationships**.

- Relationships may hold among classes or among objects.

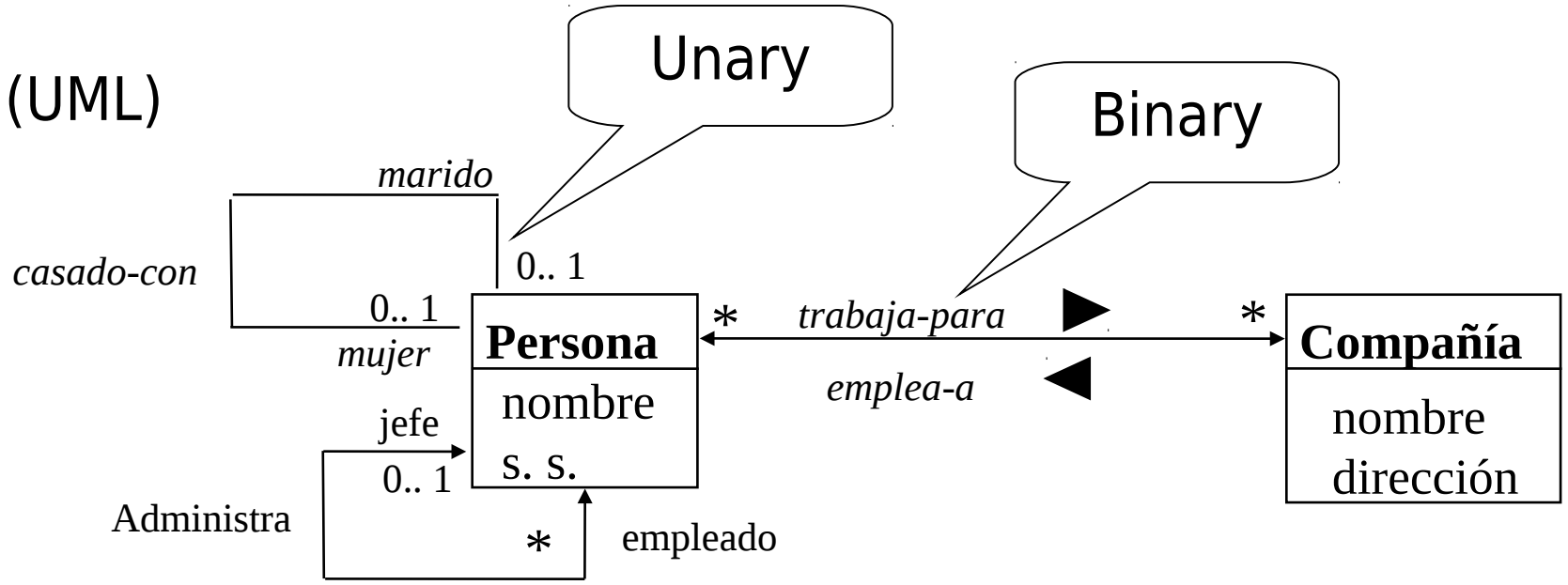
- Besides that, there exist two different type of relationships among objects:
  - **Persistent*:*** they may be stored and reused at a later time.
  - **Non-persistent*:*** these relationships vanish after being established and used.

Relationships among classes and objects

	Persistent	Non-persistent
Object level relationships	▪ **Association** Ladybugs and flowers  ▪ **Whole/part** (“part of”/”has a”) ▪ Aggregation  ▪ Composition  A petal is a part of a flower	▪ **Using (dependency)** 
Class level relationships	▪ **Generalization/Specialization** (“is a”) (inheritance)  A rose is a kind of flower	

Relationships among objects
Association

- **Association** defines a relationship between classes of objects that allows one object instance to cause another to perform an action on its behalf.
- Unless specified, **navigation** across an association is bidirectional, although it may be limited to just one direction by using an arrowhead pointing to the direction of traversal:



Relationships among objects
Association

- The ends of an association can be adorned with role names and multiplicity:
  - **Role**: exact way in which one class relates to another (semantic dependency)
    - Implementation: reference name

- **Multiplicity**: cardinality of the association.
  - Default: 1
  - Format: *(minimum..maximum)*
    - Examples (UML notation)

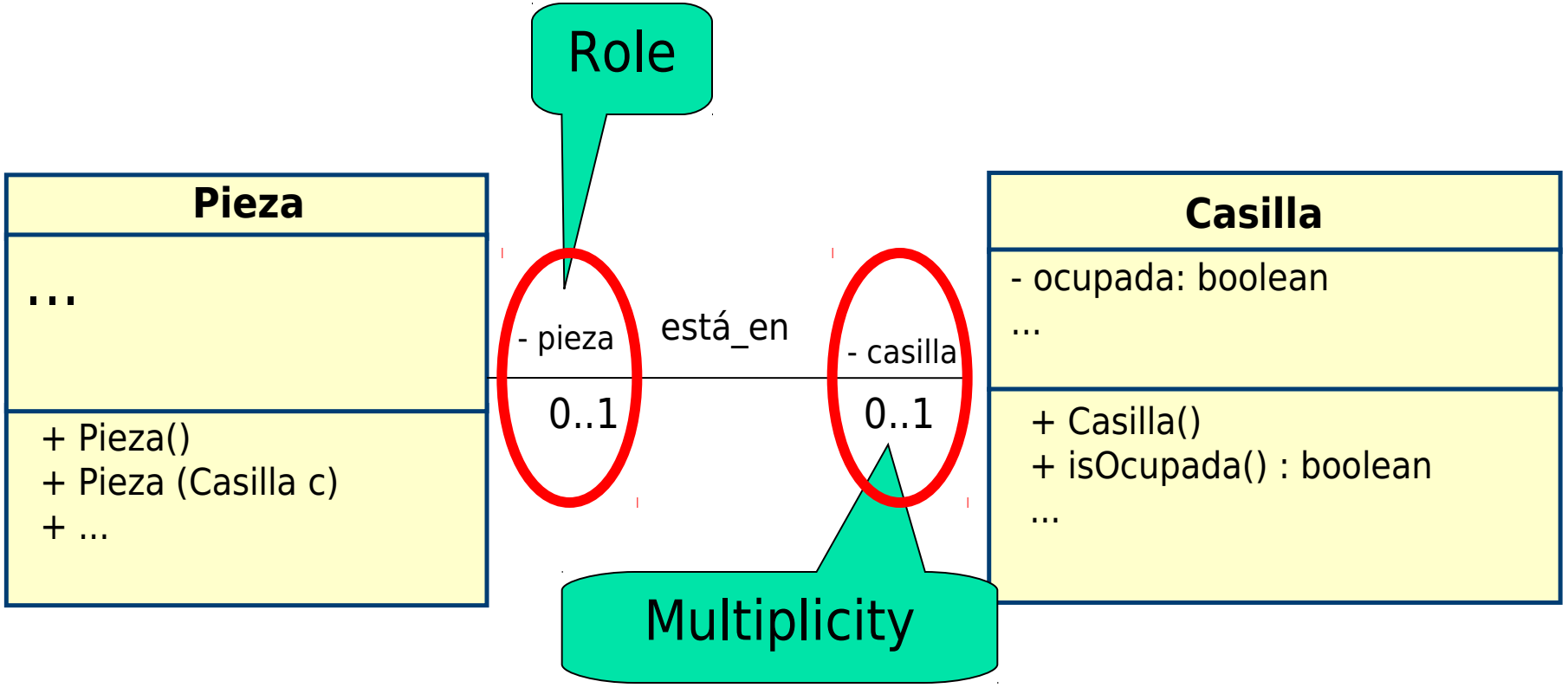
1	Exactly one. One-to-one (default)
0..1	Zero or one. Also (0,1)
M..N	From M to N (natural integers)
*	Zero or more
0..*	Zero or more
1..*	One or more
1,5,9	One, five or nine

Relationships among objects
Association

- The two objects A and B linked in an association exist in an independent manner.
  - Creation or destruction of one of them only implies the creation or destruction of the relationship; it does not imply the creation or destruction of the other object.
- Implementation
  - One single reference or a reference vector with size equal to the maximum cardinality.
    - Navigation of the association determines the class where the new member should be inserted.
    - When maximum is *: use a dynamic reference array (Java: Vector, ArrayList, LinkedList,...).

Relationships among objects
Association: chess example

- Define class Pieza based on the information in next figure
  - One piece is related to 0..1 squares
  - One square is related to 0..1 pieces



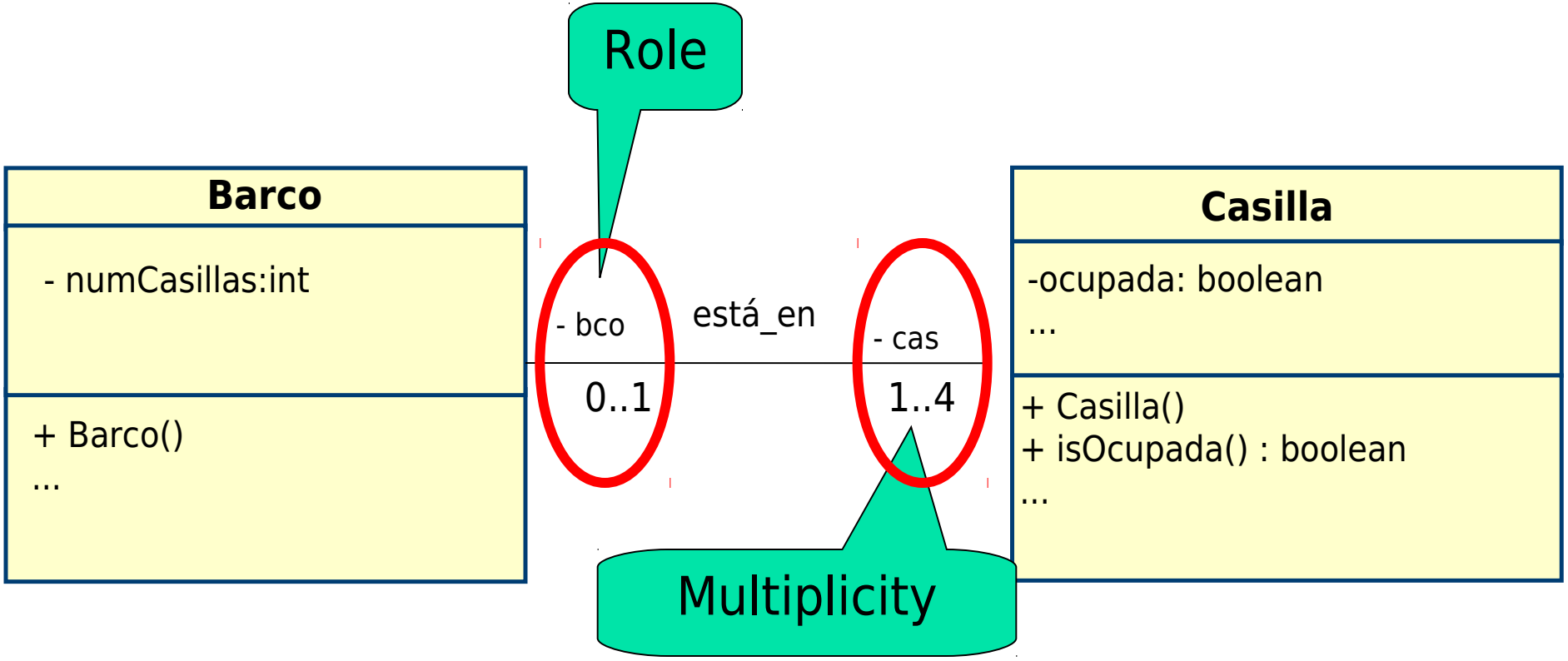
Relationships among objects
Association: chess example

```
class Pieza{
  public Pieza() {casilla=null;}  // Default constructor
  public Pieza(Casilla c) {  // Overloaded constructor
      casilla=c;
  }

  public Casilla getCasilla() { return casilla; }
  public void setCasilla(Casilla c) { casilla = c; }

  private Casilla casilla; // 'casilla': role name
    ...
 }
```

(Implement class Casilla in a similar way)

Relationships among objects
Association: battleship example

- Define class Barco based on the information in next figure
  - A ship is related to 1..4 squares
  - A square is related to 0..1 ships



Relationships among objects
Association: battleship example

```
class Barco{
  private static final int MAX_CAS=4;
  private Casilla cas[] = new Casilla[MAX_CAS];
  private int numCasillas;

  public Barco() {
      numCasillas=0;
      for (int x=0;x<MAX_CAS;x++)
        cas[x]=null;
  }
}
```

- **Which situation is not controlled in this code? Introduce as many changes as necessary.**

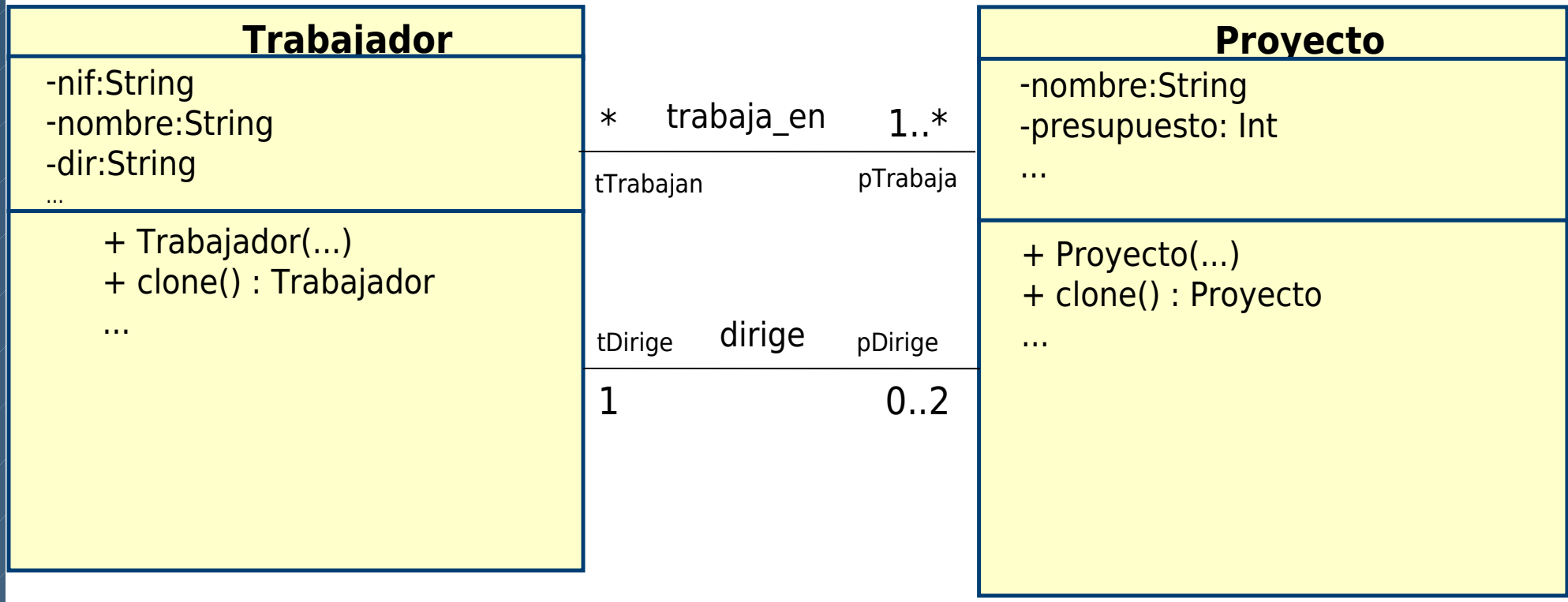
- **Is the code of class Casilla defined in the previous example still valid (after changing reference to Pieza to Barco)?**

Relationships among objects
Association: battleship example

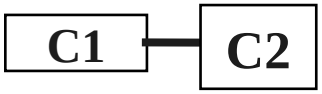
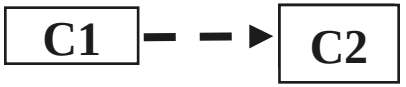
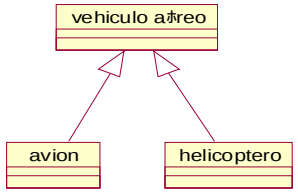
```
class Barco{
  private static final int MAX_CAS=4;
  private Vector<Casilla> cas;
  // Vector is in package java.util.* of Java API

  public Barco(Vector<Casilla> c) {
      cas=null;
      if (c.size() <= MAX_CAS && c.size() > 0)
        cas = c;
  }
  // note: throwing an exception (subtopic 3) in case the
  // number of squares is not correct, would be a better
  // idea
}
```

Relationships among objects
Association: exercise

- Define classes Trabajador y Proyecto based on the information in next figure
  - A worker must be assigned at least to a project and manage two of them at most
  - A project involves n workers and one manager is necessary.

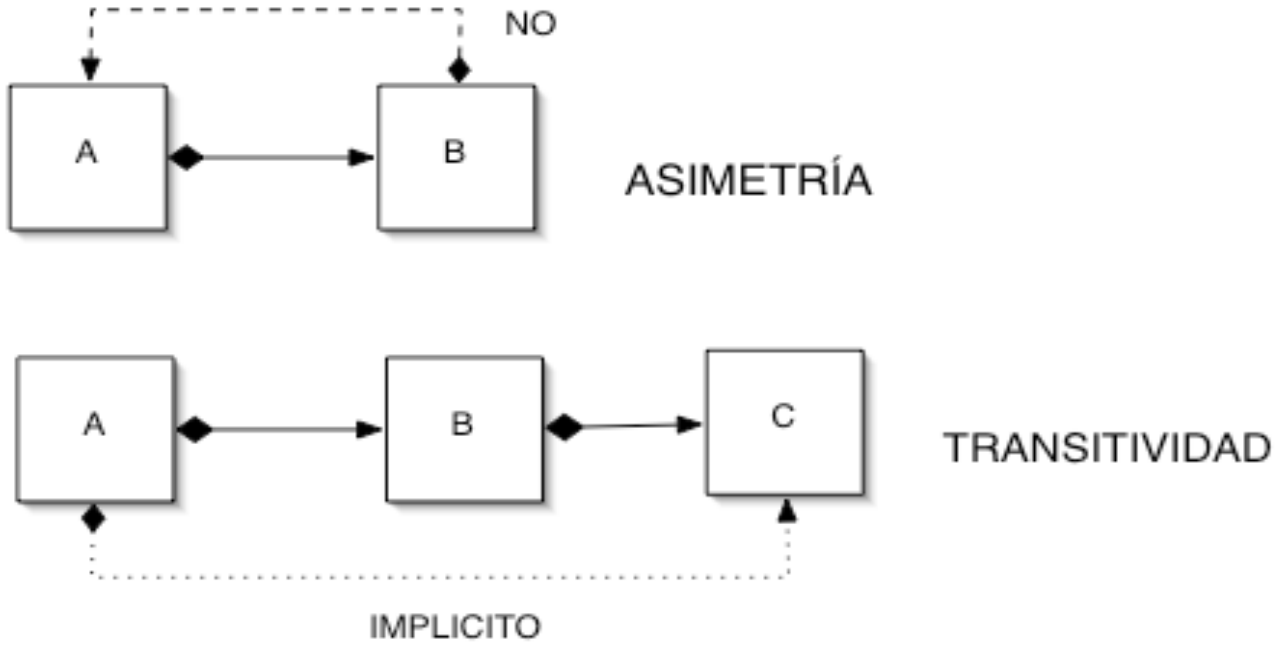


Relationships among classes and objects

	Persistent	Non-persistent
Object level relationships	▪ **Association** Ladybugs and flowers  ▪ **Whole/part** (“part of”/”has a”) ▪ Aggregation  ▪ Composition  A petal is a part of a flower	▪ **Using (dependency)** 
Class level relationships	▪ **Generalization/Specialization** (“is a”) (inheritance)  A rose is a kind of flower	

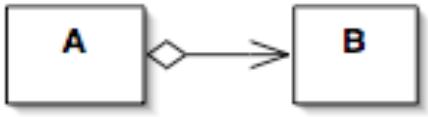
Relationships among objects
Whole/Part

- Whole/part is a form of the "has a" relationship
- Whole/part is more specific than association
- It is an association that represents a part-whole or part-of relationship.
  - 'A is composed of B', 'A has B'
- Association vs. Whole/Part
  - Whole/part is **asymmetric** and **transitive**.



Relationships among objects
Whole/Part

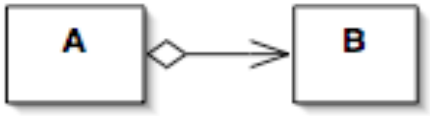
- There are two variants of the Whole/part relationship:
  - **Aggregation** (hollow diamond shape)



    - Binary association representing a whole/part relationship ('has a', 'is part of', 'belongs to')
  - If the container is destroyed, its contents are not.
    - E.g.: a team and its members
  - **Composition**

  - An even stronger form of ownership (filled diamond shape):
    - The multiplicity of the aggregate end may not exceed one; it is unshared. An object may be part of only one composite at a time.
    - If the composite is destroyed, it must destroy all its parts.
      - E.g.: a book and its chapters

Aggregation or composition?



May the part object be shared by more than one composite object?

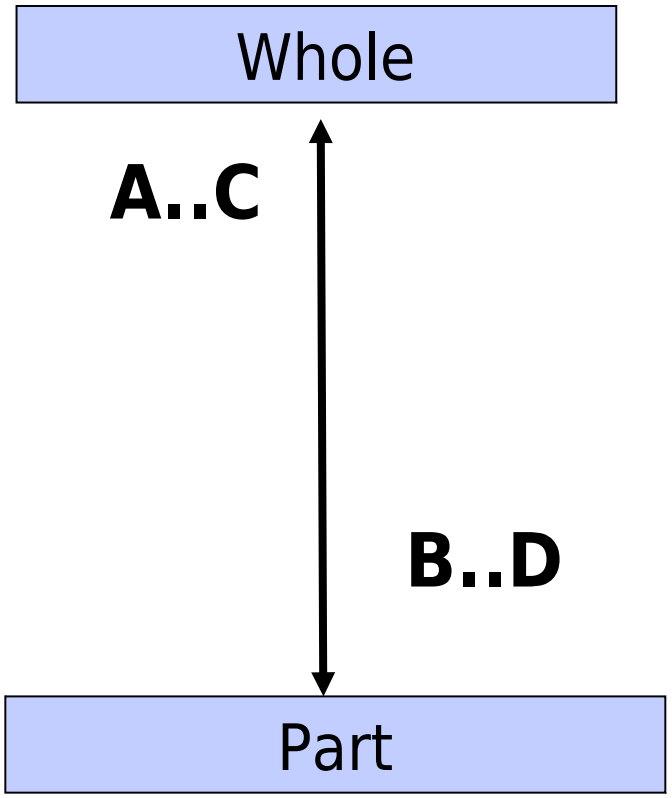
- No => **disjoint** (composition)
- Yes => non-disjoint (aggregation)

May the part object exist without being part of a composite object?

- Yes => flexible (aggregation)
- No => **strict; life cycle dependency on the container** (composition)

Relationships among objects
Whole/Part

- Multiplicity in whole/part relationships



A: Minimum multiplicity

0 → flexible

> 0 → strict

B: Minimum multiplicity

0 → nulls allowed

> 0 → nulls not allowed

C: Maximum multiplicity

1 → disjoint

> 1 → non disjoint

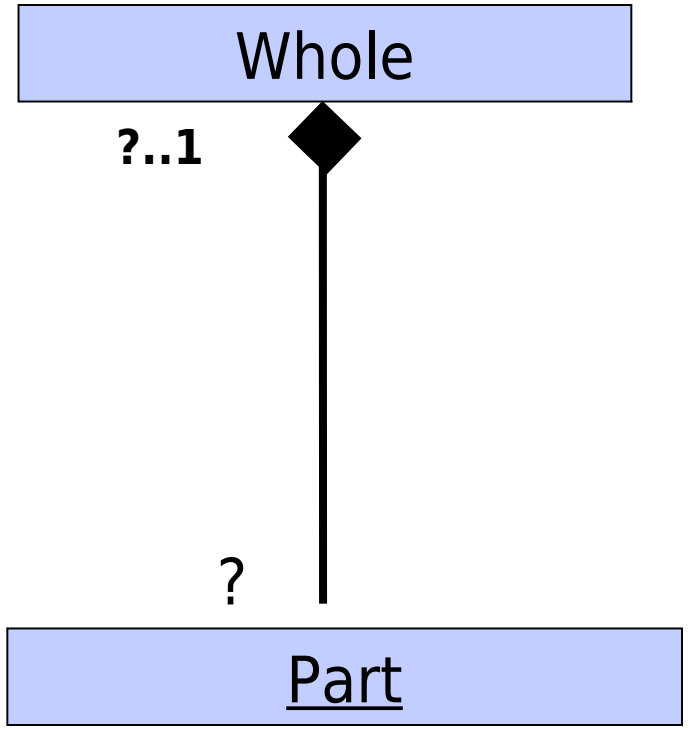
D: Maximum multiplicity

1 → univalued

> 1 → multivalued

Whole/part relationships among objects
Composition

- A composition can be identified by a maximum cardinality of 1 in the composite object:

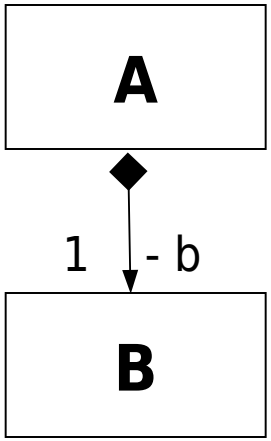


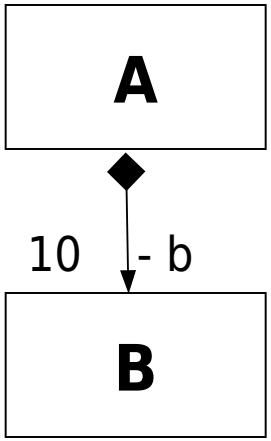
Whole/part relationships among objects
Implementation

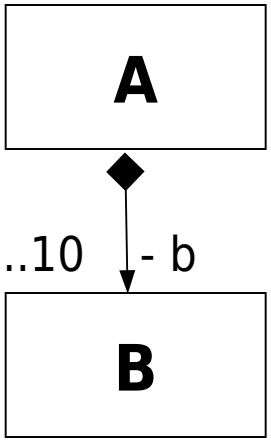
- Aggregation: implemented as an unidirectional association
  - The composite object contains (possibly shared) references to its parts.

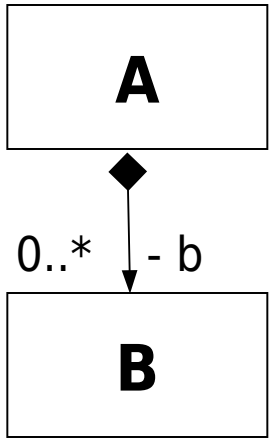
- Composition:
  - The composite object has sole responsibility for the disposition of its parts in terms of creation and destruction.

Whole/part relationships among objects
Java **Implementation**









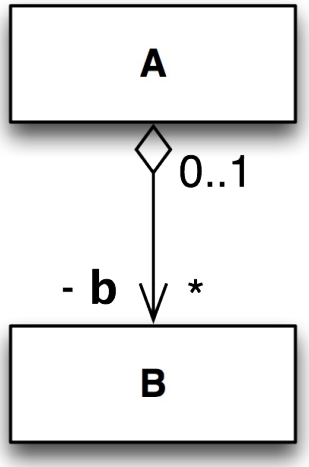

```
class A {
  private B b;
// b is a
// subobject
…}
```

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

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

…};
```

Attribute declaration is exactly the same for both aggregation and composition.

Whole/part relationships among objects
Aggregation **Implementation**



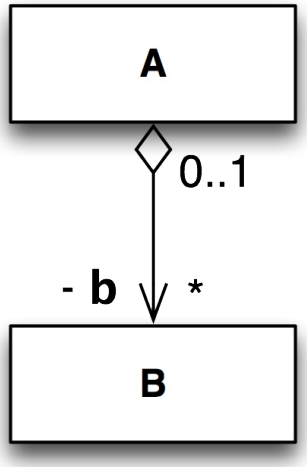

```
class A {
 private Vector<B> b = new Vector<B>();

 public A() {}
 public addB(B unB) {
   b.add(unB);
 }
…}
```

A) Object B can be created out of class A; therefore, external references ('objB') to the aggregate object may exist.

```
// Somewhere else, possibly out of A
B objB = new B();
if (…) {
  A objA = new A();
  objA.addB(objB);
}
```

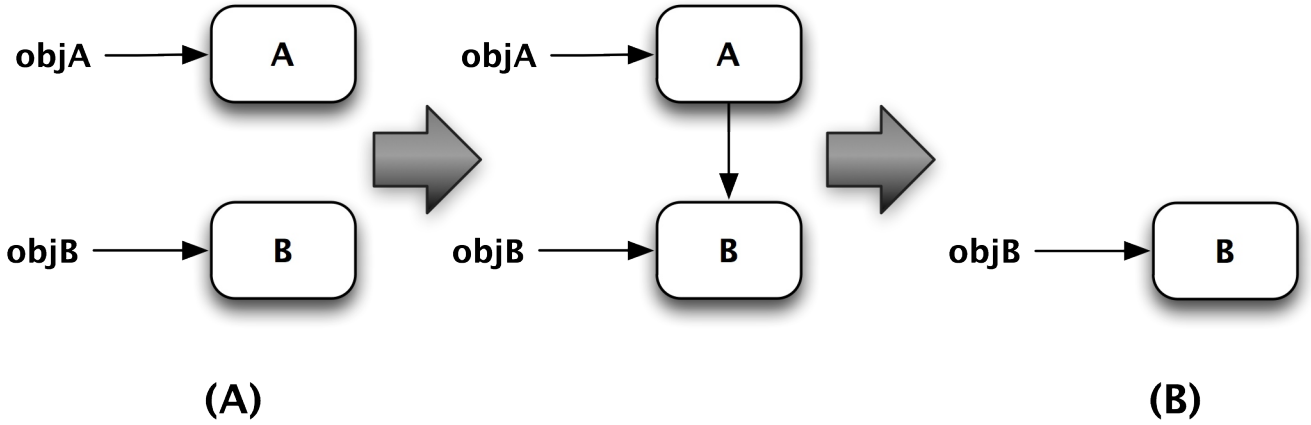
B) After 'objA' is destroyed, 'objB' keeps on existing.

Whole/part relationships among objects
Aggregation **Implementation**

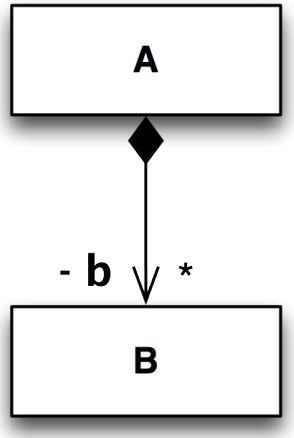


A) Object B can be created out of class A; therefore, external references ('objB') to the aggregate object may exist.

B) After 'objA' is destroyed, 'objB' keeps on existing, since other references are still pointing to it.



Whole/part relationships among objects
Composition **Implementation**




```
class A {
 private Vector<B> b = new Vector<B>();

 public A() {}
 public addB(...) {
   b.add(new B(...));
 } '...' stands for the information
   necessary to create B
 ...}
```

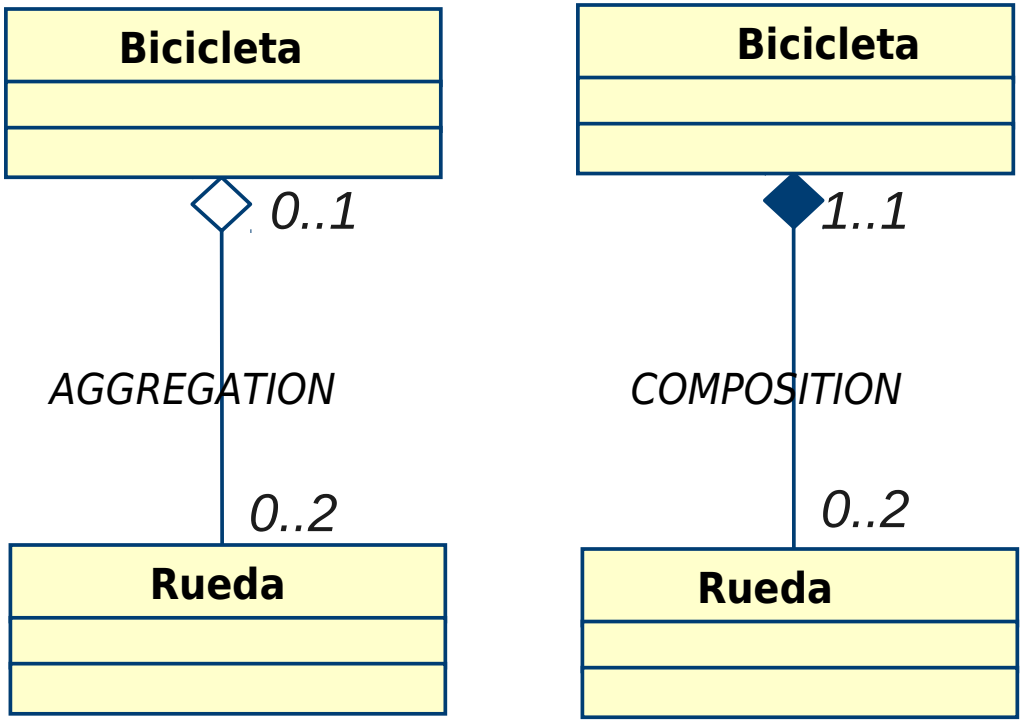
A) Object B is created inside of A; therefore, A is the only place keeping references to its part B.

```
// Somewhere out of A...

if (…) {
  A objA = new A();
  objA.addB(...);
}
```

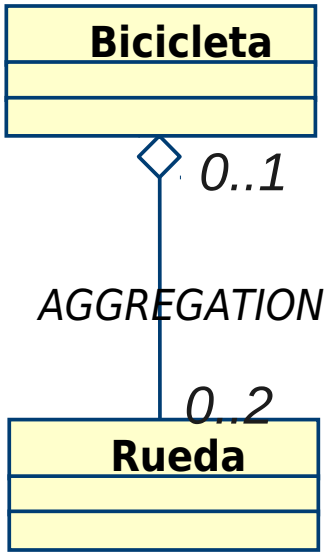
B) When 'objA' is destroyed, its components B are destroyed as well.

Whole/part relationships among objects
Bicycle example

Some relationships may be considered as aggregations or compositions depending on the context in which they are used.

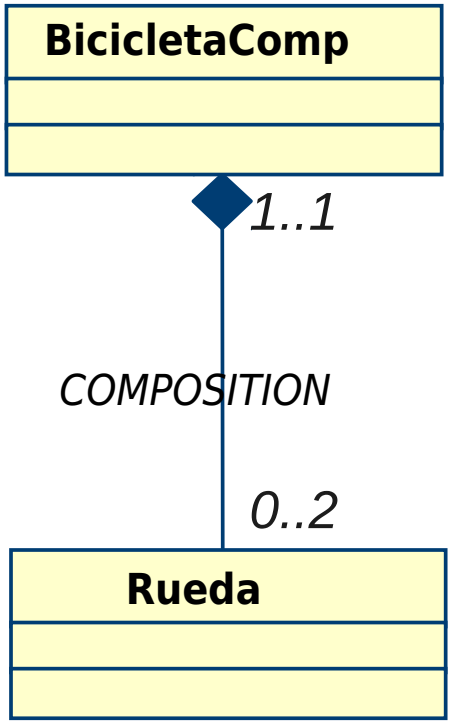


Whole/part relationships among objects
Bicycle example




```
class Rueda {
    private String nombre;
    public Rueda(String n){nombre=n;}
}

class Bicicleta {
  private Vector<Rueda> r;
  private static final int MAXR=2;
  public Bicicleta(Rueda r1, Rueda r2){
    r.add(r1);
    r.add(r2);
  }
  public void cambiarRueda(int pos, Rueda raux){
    if (pos>=0 && pos<MAXR)
      r.set(pos,raux);
  }

  public static final void main(String[] args)
  {
    Rueda r1=new Rueda("1");
    Rueda r2=new Rueda("2");
    Rueda r3=new Rueda("3");
    Bicicleta b(r1,r2);
    b1.cambiarRueda(0,r3);
  }
}
```

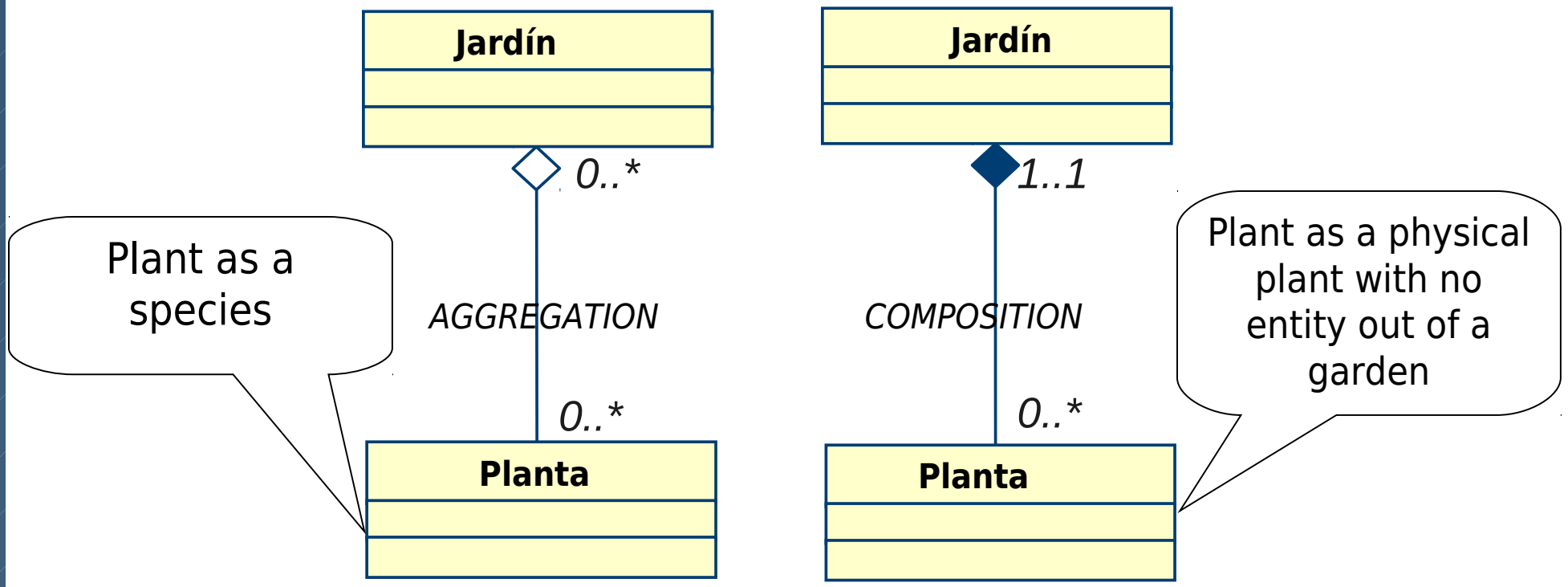
Whole/part relationships among objects
Bicycle example




```
class BicicletaComp{
  private static const int MAXR=2;
  private Vector<Rueda> r;
  public BicicletaComp(String p,String s){
        r.add(new Rueda(p));
        r.add(new Rueda(s));
      }
  public static final void main(String[] args) {
    BicicletaComp b2 = new BicicletaComp(“1”,”2”);
    BicicletaComp b3 = new BicicletaComp(“1”,”2”);

  // they are different wheels, although they have
  // identical names
  }
}
```

Whole/part relationships among objects
Garden example

Notice the differences in semantics of class Plant depending on the kind of relationship with Jardín



Whole/part relationships among objects
Garden example

Consider this code:

```
class Planta {
 public Planta(String n, String e)
   {...}

   public String getNombre() {...}
   public String getEspecie() {...}
   public String getTemporada() {...}
   public void setTemporada(String t)
   {...}

   private String nombre;
   private String especie;
   private String temporada;
}
```

```
class Jardin {
 public Jardin(String e) {...}

 public Jardin clone() {...}
 public void plantar(String n, String e,
   String t)
 {
   Planta lp = new Planta(n, e);
   lp.setTemporada(t);
   p.add(lp);
 }

 private Vector<Planta> p
   = new Vector<Planta>();
 private String emplazamiento;
}
```

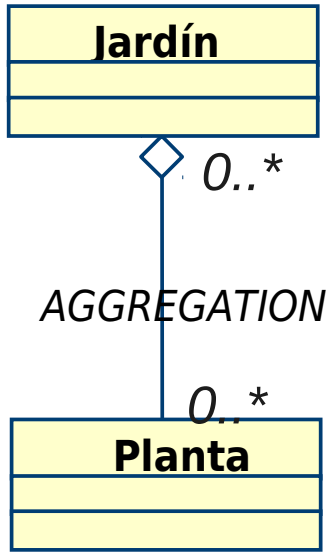
What kind of relationship between Jardín and Planta is shown here?

Whole/part relationships among objects
Garden example

Now implement the method Jardin.clone()...

Whole/part relationships among objects Exercise

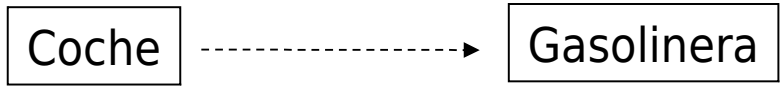
Which changes should be implemented in case the garden is viewed as an aggregation of plants?



Relationships among objects
Using

- A class A **uses** a class B when A does not contain attributes of class B, but:
  - It uses some instance of B passed as a parameter (or defined as a local variable) in some of its methods.
  - **It access B's private variables** (friend classes)
  - It uses some **class method** of B.
- In UML, this kind of relationship is represented with **dependencies**.

Relationships among objects
Using: example

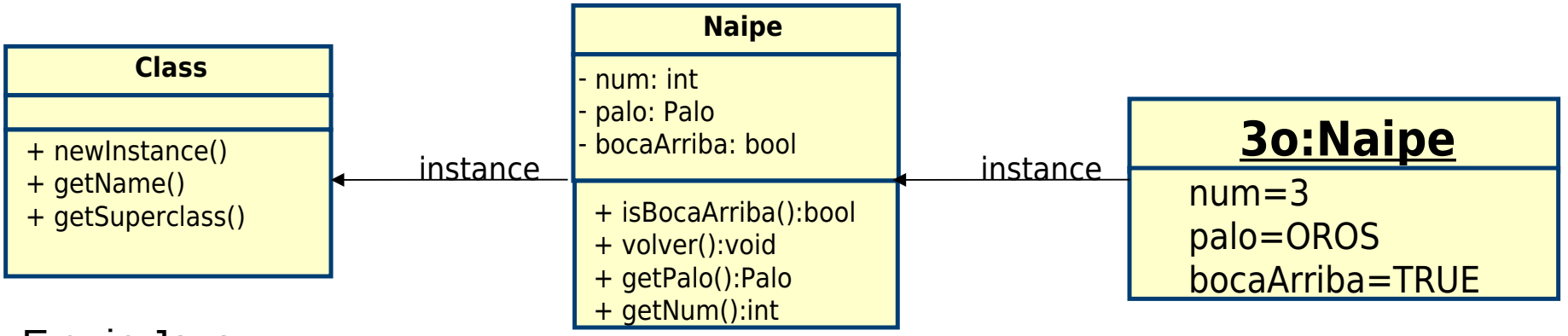


- If we are not interested in storing the gas stations where gas has been obtained, this is not an association.
- However, an interaction exists:

```
class Coche {
  Carburante tipoC;
  float lGaso;
  float respostar(Gasolinera g, float litros){
    float importe=g.dispensarGaso(litros,tipoC);
    if (importe>0.0) // get some gas if true
       lGaso=lGaso+litros;
   return (importe);
  }
}
```

Metaclasses

- Methods may be assigned to classes instead of objects:
  - `new, delete`
  - Class methods
- In Smalltalk, Java and other languages, a class is an instance of another class known as **metaclass**.
  - Therefore, classes themselves may respond to some messages like the message for object creation new



E.g. in Java:

```
Naipe n1 = new Naipe();
Class c = n1.getClass();
Naipe n2 = (Naipe) c.newInstance();
```

Bibliography

- Cachero et. al.
  - ***Introducción a la programación orientada a Objetos***
    - Ch. 2
- T. Budd.
  - ***An Introduction to Object-oriented Programming, 3rd ed.***
    - Ch. 4 and 5; ch. 6: case study for different OOL
- G. Booch.
  - ***Object Oriented Analysis and Design with Applications***
    - Ch. 3 and 4
- G. Booch et. al.
  - ***El lenguaje unificado de modelado***. Addison Wesley. 2000
    - Section 2 (ch. 4-8)