

## UD 2

# ***CONCEPTOS BÁSICOS de la programación orientada a objetos***

Versión 0.1  
(Curso 11/12)

*Pedro J. Ponce de León*



- Objetos
- Clases
- Atributos
- Operaciones
- UML
- Relaciones
  - Asociación
  - Todo-parte
  - Uso
- Metaclases

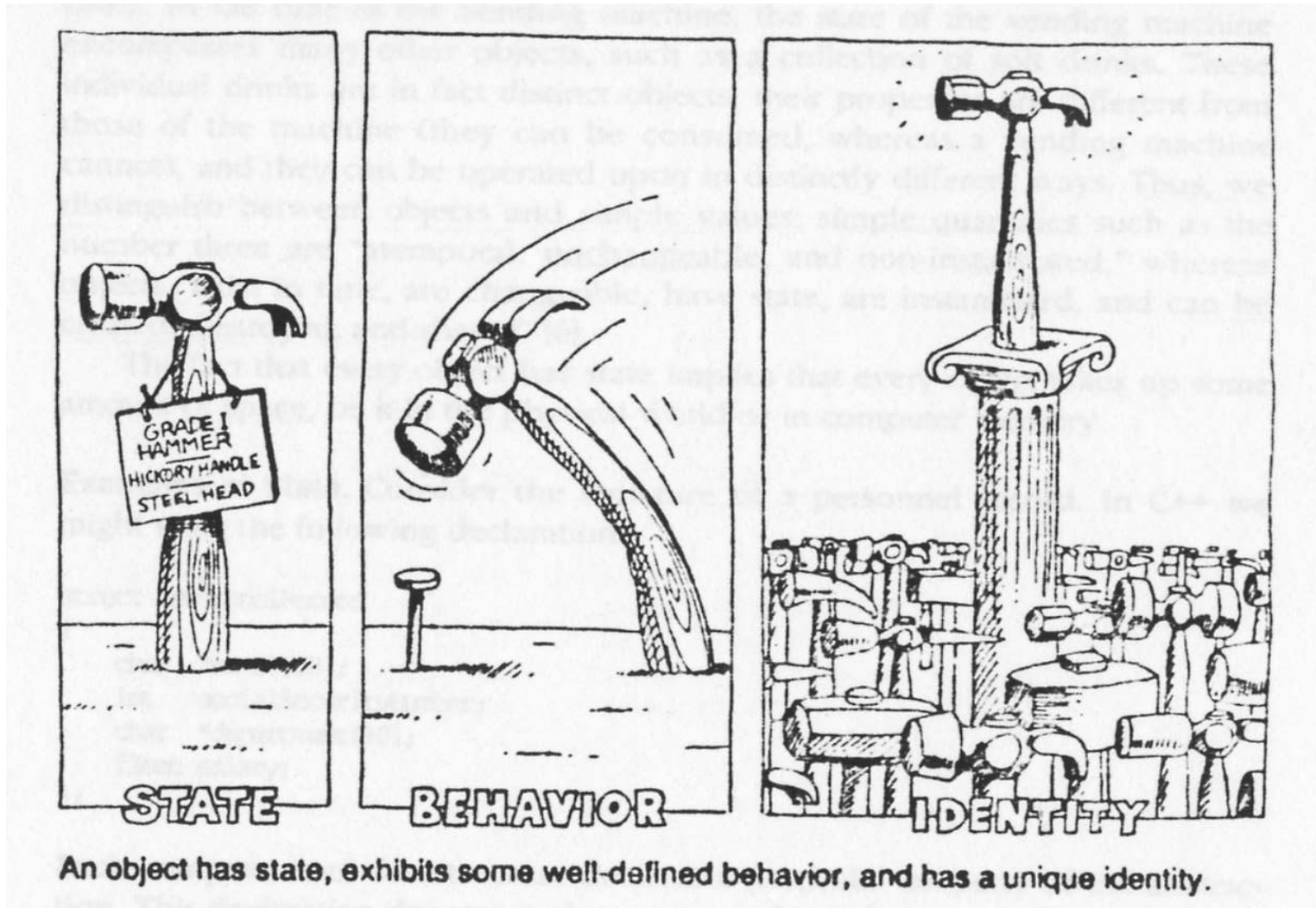
- Un objeto es cualquier cosa a la que podemos asociar unas determinadas **propiedades y comportamiento**.
- Desde el punto de vista del analista: un objeto representa una entidad (real o abstracta) con un papel bien definido en el dominio del problema.
- Desde el punto de vista del programador: un objeto es una estructura de datos sobre la cual podemos realizar un conjunto bien definido de operaciones.

- Según *Grady Booch*: Un objeto tiene un estado, un comportamiento y una identidad:
  - **Estado**: conjunto de propiedades del objeto y valores actuales de esas propiedades.
  - **Comportamiento**: modo en que el objeto actúa y reacciona ante los mensajes que se le envían (con posibles cambios en su estado). Viene determinado por la **clase** a la que pertenece el objeto.
  - **Identidad**: propiedad que distingue a unos objetos de otros (nombre único de variable)

# Objeto

## Definición

- Definición de Booch:



- Abstracción de los atributos (características), operaciones, relaciones y semántica comunes a un conjunto de objetos.
- Así, una clase representa al conjunto de objetos que comparten una estructura y un comportamiento comunes. Todos ellos serán **instancias** de la misma clase.

- **Identificador de Clase:** nombre
- **Propiedades**
  - **Atributos** o *variables*: datos necesarios para describir los objetos (*instancias*) creados a partir de la clase.
    - La combinación de sus valores determina el estado de un objeto.
  - **Roles**: relaciones que una clase establece con otras clases.
  - **Operaciones, métodos, servicios**: acciones que un objeto conoce cómo ha de ejecutar.

Nombre
atributos
operaciones

rol

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

# ¿Objetos o clases?



- Película **CLASE**
- Carrete de película **CLASE**
- Carrete con nº de serie 123456 **OBJETO**
- Pase de la película 'La vida de Brian' en el cine Muchavista a las 19:30 **OBJETO**

En general:

- Algo será una clase si puede tener instancias.
- Algo será un objeto si es algo único que comparte características con otras cosas similares

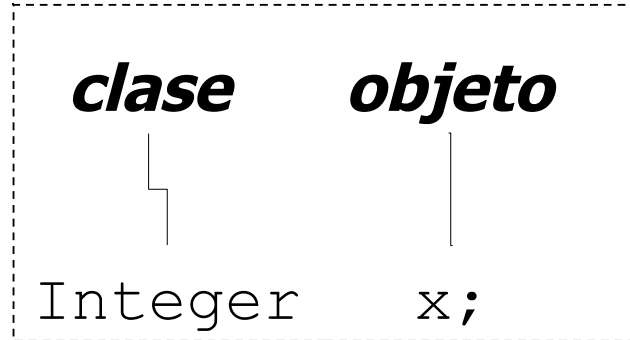


# Objeto

## Objetos y clases en un lenguaje de programación



- Clase  $\longleftrightarrow$  Tipo
- Objeto  $\longleftrightarrow$  Variable o constante



Clase: caracterización de un conjunto de objetos que comparten propiedades

- **Atributo** (*dato miembro o variable de instancia*)
  - Porción de información que un objeto posee o conoce de sí mismo.
    - Suelen ser a su vez objetos
    - Se declaran como 'campos' de la clase.
  - **Visibilidad** de un atributo
    - *Indica desde donde se puede acceder a él.*
    - + Pública (interfaz)                      -> desde cualquier lugar
    - - Privada (implementación)              -> sólo desde la propia clase
    - # Protegida (implementación)           -> desde clases derivadas
    - ~ De paquete (en Java)                   -> desde clases definidas en el mismo paquete

Es habitual que los atributos formen parte de la implementación (parte oculta) de una clase, pues conforman el estado de un objeto.

### ■ **Constantes / Variables**

- Constante: ej., una casa se crea con un número determinado de habitaciones (característica estable):

- `private final int numHab;`

- Variable: ej., una persona puede variar su sueldo a lo largo de su vida:

- `private int sueldo;`

### ■ **De instancia / De clase**

- De instancia: atributos o características de los objetos representados por la clase. Se guarda espacio para una copia de él por cada objeto creado:

- `private String nombre; // nombre de un Empleado`

- De clase: características de una clase comunes a todos los objetos de dicha clase:

- `private static String formatoFecha; // de la clase Fecha`

- Implican una sola zona de memoria reservada para todos los objetos de la clase, y no una copia por objeto, como sucede con las variables de instancia.
  - **Sirven para:**
    - **Almacenar características comunes (constantes) a todos los objetos**
      - Número de ejes de un coche
      - Número de patas de una araña
    - **Almacenar características que dependen de todos los objetos**
      - Número de estudiantes en la universidad
  - Un atributo estático puede ser accedido desde cualquier objeto de la clase, ya que es un dato miembro de la clase.

- **Operación** (función miembro, método o servicio de la clase)
  - Acción que puede realizar un objeto en respuesta a un mensaje. Definen el **comportamiento** del objeto.
  - Tienen asociada una **visibilidad** (como los atributos)
  - Pueden ser **de clase o de instancia** (como los atributos)
  - Pueden modificar el estado del sistema (**órdenes**) o no (**consultas**)
  - **Signatura de una operación en :**  
`TipoRetorno`  
`NombreClase.NombreFuncionMiembro (parametros)`

### ■ De instancia/De clase

#### ■ ***Operaciones de instancia:***

- Operaciones que pueden realizar los objetos de la clase.
- Pueden acceder directamente a atributos tanto de instancia como de clase.
- Normalmente actúan sobre el objeto receptor del mensaje.

```
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;  
}
```

# Operaciones

## Tipos de Operaciones



- **De instancia/De clase**

- ***Operaciones de clase:***

- Operaciones que acceden exclusivamente a atributos de clase.
- No existe receptor del mensaje (a menos que se pase explícitamente como parámetro).
- Se pueden ejecutar sin necesidad de que exista ninguna instancia.

```
class Circulo {  
  
    private static final double pi=3.141592;  
  
    public static double getRazonRadioPerimetro()  
    {  
        return 2*pi;  
    }  
  
    ...  
};
```

### ■ Órdenes

- Pueden modificar el estado del objeto receptor.

```
c.setRadio(3); // modifica el radio de 'c'
```

### ■ Consultas

- No modifican al objeto receptor.

```
c.getRadio(); // consulta el radio de 'c'
```



- Algunos LOO soportan la **sobrecarga** de operaciones.
  - Consiste en la existencia, dentro de un mismo ámbito, de más de una operación definida con el mismo nombre (selector), pero diferente número y/o tipo de argumentos.

```
class Circulo {  
    // pinta sin relleno  
    public void pintar() {...}  
    // pinta con relleno  
    public void pintar(Color) {...}  
}
```

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

# Operaciones

## Referencia al objeto receptor



- En muchos LOO, en los métodos el receptor es un argumento implícito.
- Para obtener una referencia a él dentro de un método de instancia, existe una *pseudo-variable*:
  - En C++ y Java, se llama **this**.
  - En otros lenguajes, es **self**
- Ejemplo en Java:

*receptor.selector( this , <argumentos> )*



```
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(); }  
}
```

- Operación cuyo objetivo es crear e inicializar objetos.
  - Se invoca siempre que se crea un objeto, mediante el operador **new** (en Java).
  - El enlazado de creación e inicialización asegura que un objeto nunca puede ser utilizado antes de que haya sido correctamente inicializado.
  - En Java y C++ tienen el mismo nombre que la clase y no devuelven nada (ni siquiera void).

```
class Circulo {  
  
    public Circulo() {...}; // Constructor por defecto  
    public Circulo(double r) {...}; // Constructor sobrecargado  
}
```

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

- **Constructor por defecto:**

- Es conveniente definir siempre uno que permita la inicialización **sin parámetros** de un objeto, donde los atributos de éste se inicialicen con valores por defecto.

```
public Circulo()  
{  
    super(); // llamada a ctor. de Object (automático)  
    radio = 1.0;  
}
```

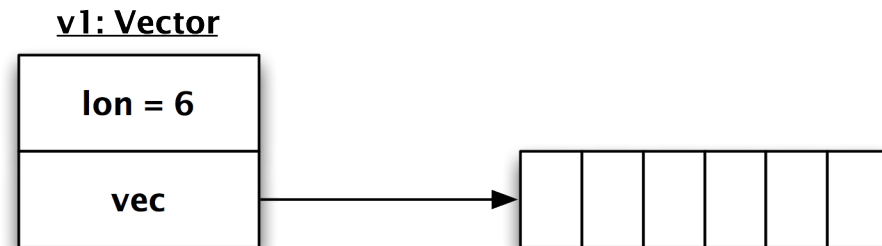
- En Java y C++, si no se define ninguno de manera explícita, el compilador genera uno con visibilidad pública, llamado **constructor de oficio**.

```
public Circulo()  
{  
    super();  
    /* Java: todos los atributos = 0 ó null */  
}
```

- Existen dos formas de 'copiar' o 'clonar' objetos
  - Shallow copy (copia superficial)
    - Copia bit a bit de los atributos de un objeto
  - Deep copy (copia completa)
- Supongamos una clase Vector:

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

**Vector v1 = new Vector(6);**



- Shallow copy
  - En C++ y Java, es el modo de copia por defecto

**C++:**

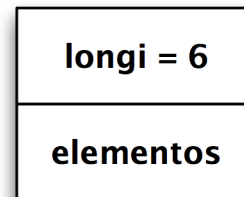
```
Vector v1(6);  
Vector v2(v1);
```

**Java:**

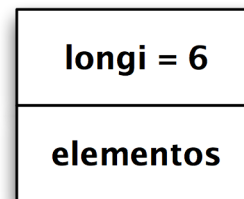
```
Vector v1 = new Vector(6);  
Vector v2 = v1.clone();
```

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

v1: Vector



v2: Vector



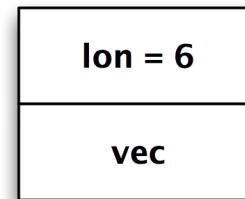
- Deep copy
  - Es necesario implementar explícitamente
    - C++: un **constructor de copia**
    - Java: un ctor. de copia ó el método **clone()**

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

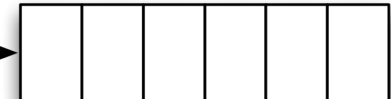
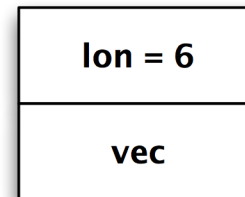
## Java: constructor de copia

```
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];  
}
```

### v1: Vector



### v2: Vector



- Deep copy con **clone()**

La diferencia con el ctor. de copia es que el método `clone()` también copia la parte del objeto definida en la clase `Object` (esto quedará más claro al estudiar herencia).

**Java:**

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

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

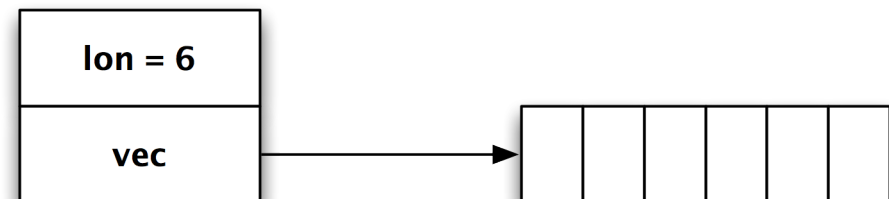
    for (int i=0; i<vec.length; i++)
        clon.vec[i] = vec[i];

    return clon;
}
```

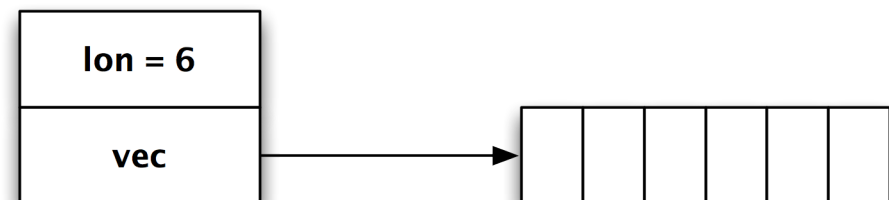
`Vector v1 = new Vector(6);`

`Vector v2 = v1.clone();`

v1: Vector



v2: Vector





- La mayoría de lenguajes OO disponen de alguna forma de ejecutar código de usuario cuando un objeto es destruido.
  - C++: Destructores
  - Java: métodos `finalize()`
- En Java, el recolector de basura se encarga de destruir los objetos para los cuales ya no existen referencias en la aplicación.

Implica que el programador no tiene control sobre cuándo exactamente se libera la memoria de un objeto.

En Java y Eiffel podemos definir métodos `finalize()`, que serán ejecutados justo antes de que el objeto sea destruido.

- `protected void finalize() {} // def. en clase Object`

Los métodos `finalize()` se usan para liberar recursos que el objeto haya podido adquirir (cerrar ficheros abiertos, conexiones con bases de datos,...).

→ Normalmente no es necesario definirlos.

- Es el conjunto de métodos que toda clase, independientemente de su funcionalidad específica, debería definir. Suelen existir una definición 'de oficio' proporcionada por el compilador y/o máquina virtual.
- En C++, p. ej. son éstos:
  - Constructor por defecto
  - Constructor de copia
  - Operador de asignación
  - Destructor
- En Java, son
  - Constructor por defecto (constructor de oficio)
  - `public String toString()` - Representación del objeto en forma de cadena
  - `public boolean equals(Object o)` – comparación de objetos
  - `public int hashCode()` - identificador numérico de un objeto

# Forma canónica de una clase en Java



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

    public String toString() { return ... }

    public boolean equals(Object o)
    {
        ... Define una relación de equivalencia:
        Reflexiva: a.equals(a) debe devolver true.
        Simétrica: si a.equals(o) entonces o.equals(a).
        Transitiva: si a.equals(b) y b.equals(c), entonces a.equals(c).
        (Por defecto devuelve true siambos objetos son el mismo objeto
        en memoria.)
    }

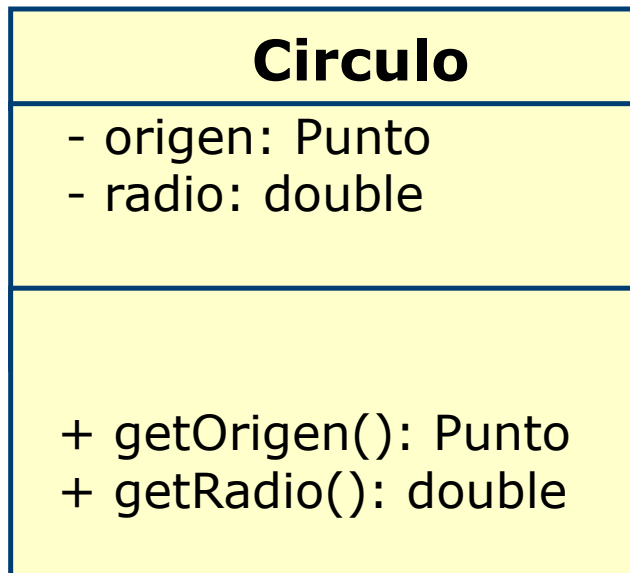
    public int hashCode()
    {
        ... Consistente con 'equals': Dos objetos "equals"
        deben tener el mismo código hash.
    }
}
```

# Notación UML de Clases, Atrib. y Oper.

## Equivalencia C++



- Distintos LOO poseen distinta sintaxis para referirse a los mismos conceptos.
  - Las líneas de código no son un buen mecanismo de comunicación
- UML resuelve este problema y homogeneiza el modo en que se comunican conceptos OO



// C++

```
class Circulo{  
    public:  
        Punto getOrigen();  
        double getRadio();  
  
    private:  
        Punto origen;  
        double radio;  
}
```

# Notación UML de Clases, Atrib. y Oper.

## Otras equivalencias



### **Circulo**

- origen: Punto  
- radio: double

+ getOrigen(): Punto  
+ getRadio(): double

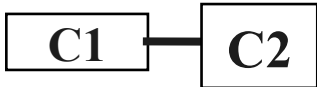



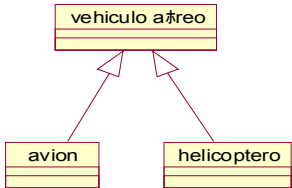
### // Java y C#

```
class Circulo{  
    public Punto getOrigen()  
        {return origen;}  
    public double getRadio()  
        {return radio;}  
    private Punto origen;  
    private double radio;  
}
```

# Relaciones entre Clases y Objetos

- Booch: Un objeto por sí mismo es soberanamente aburrido.
- La resolución de un problema exige la colaboración de objetos.
  - Esto exige que los agentes ***se conozcan***
- El conocimiento entre objetos se realiza mediante el establecimiento de **relaciones**.
- Las relaciones se pueden establecer **entre clases** o **entre objetos**.
- Además, a nivel de objetos, podemos encontrar dos tipos de relaciones:
  - **Persistentes**: recogen caminos de comunicación entre objetos que se almacenan de algún modo y que por tanto pueden ser reutilizados en cualquier momento.
  - **No persistentes**: recogen caminos de comunicación entre objetos que desaparecen tras ser utilizados.

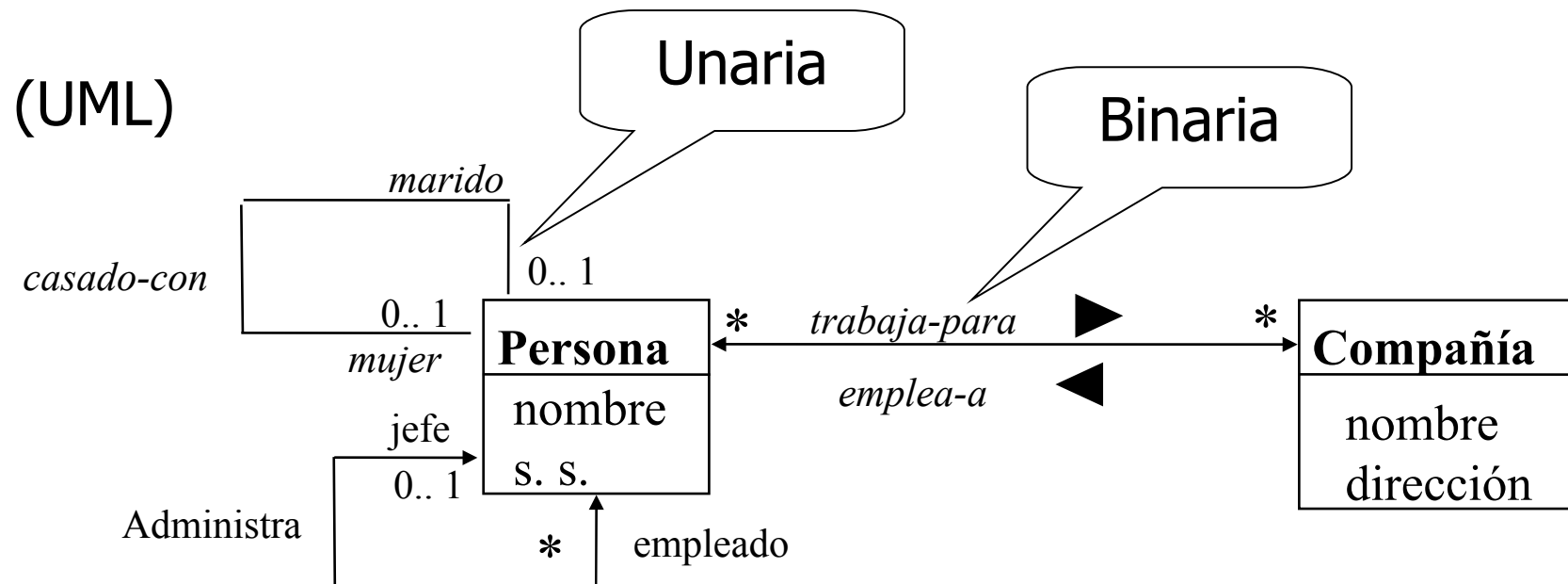
# Relaciones entre Clases y Objetos

	Persistente	No persist.
Entre objetos	<ul style="list-style-type: none"> <li>▪ <b>Asociación</b>  </li> <li>▪ <b>Todo-Parte</b> <ul style="list-style-type: none"> <li>▪ Agregación  </li> <li>▪ Composición  </li> </ul> </li> </ul>	<ul style="list-style-type: none"> <li>▪ <b>Uso (dependencia)</b>  </li> </ul>
Entre clases	<ul style="list-style-type: none"> <li>▪ <b>Generalización (Herencia)</b>  </li> </ul>	

# Relaciones entre Objetos

## Asociación

- Expresa una relación (unidireccional o bidireccional) entre los objetos instanciados a partir de las clases conectadas.
- El sentido en que se recorre la asociación se denomina **navegabilidad** de la asociación:



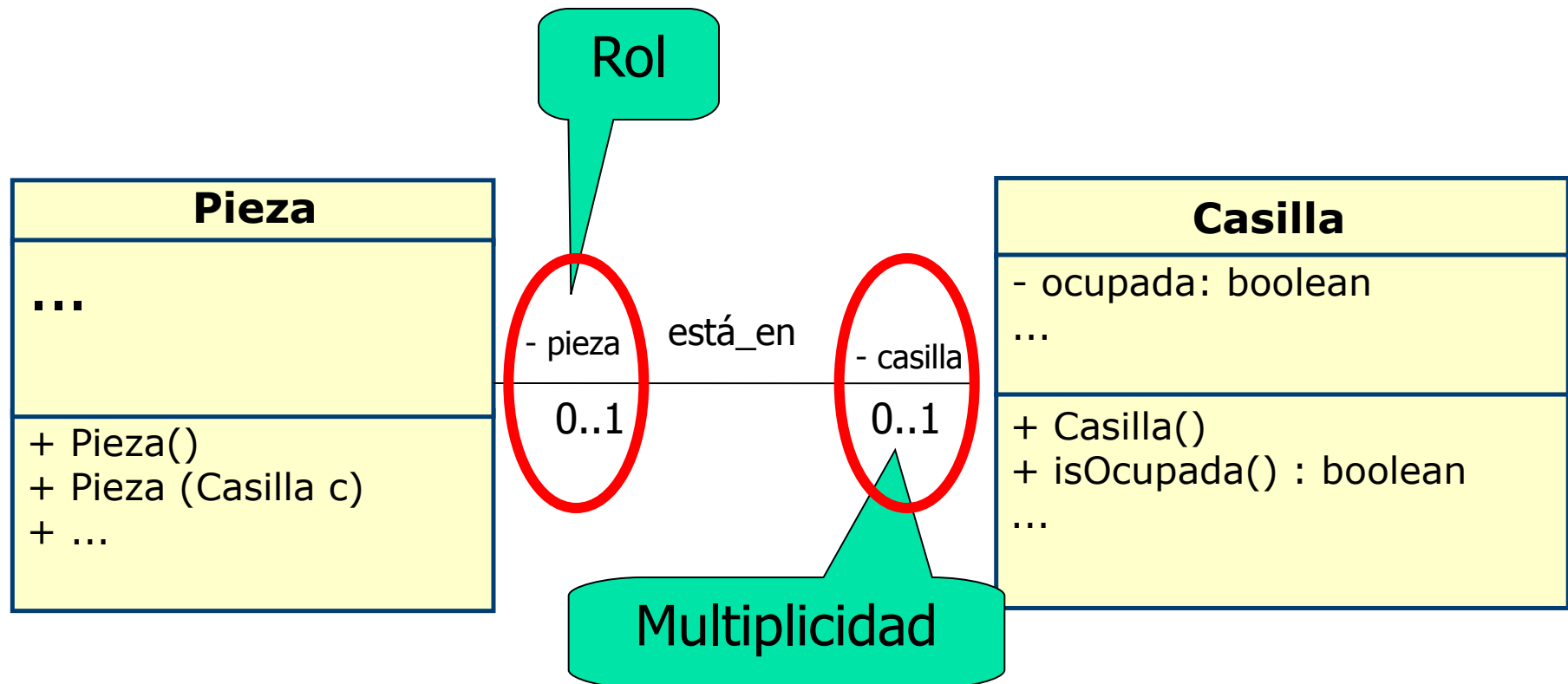


- Cada extremo de la asociación se caracteriza por:
  - **Rol:** papel que juega el objeto situado en cada extremo de la relación en dicha relación
    - Implementación: nombre de la referencia
  - **Multiplicidad:** número de objetos **mínimo** y **máximo** que pueden relacionarse con un objeto del extremo opuesto de la relación.
    - Por defecto 1
    - Formato: *(mínima..máxima)*
      - Ejemplos (notación UML)

<b>1</b>	Uno y sólo uno (por defecto)
<b>0..1</b>	Cero a uno. También (0,1)
<b>M..N</b>	Desde M hasta N (enteros naturales)
<b>*</b>	Cero a muchos
<b>0..*</b>	Cero a muchos
<b>1..*</b>	Uno a muchos (al menos uno)
<b>1,5,9</b>	Uno o cinco o nueve

- En una asociación, dos objetos A y B asociados entre sí existen de forma independiente
  - La creación o desaparición de uno de ellos implica únicamente la creación o destrucción de la relación entre ellos y nunca la creación o destrucción del otro objeto.
- Implementación
  - Una sola referencia o un vector de referencias del tamaño indicado por la cardinalidad *máxima*.
    - La decisión sobre en qué clase se introduce el nuevo dato miembro depende de la navegabilidad de la asociación.
    - Si el máximo es \*: array dinámico de referencias (Java: Vector, ArrayList, LinkedList,...).

- A partir del dibujo de la Fig., define la clase Pieza
  - Una pieza se relaciona con 0..1 casillas
  - Una casilla se relaciona con 0..1 piezas



# Relaciones entre Objetos

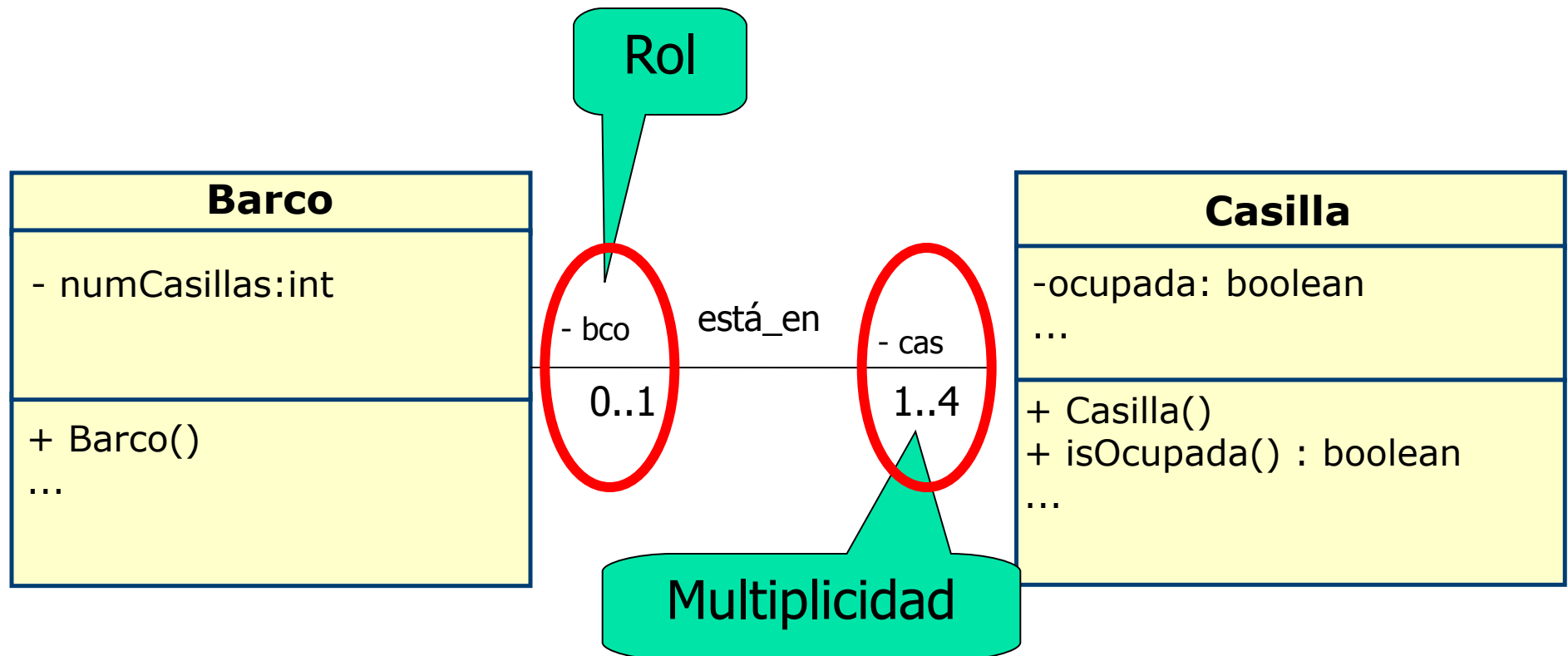
## Asociación: Ejemplo



```
class Pieza{  
    public Pieza() {casilla=null;}    // Constructor por def.  
    public Pieza(Casilla c) {    // Ctor. sobrecargado  
        casilla=c;  
    }  
  
    public Casilla getCasilla() { return casilla; }  
    public void setCasilla(Casilla c) { casilla = c; }  
  
    private Casilla casilla; // 'casilla': nombre del rol  
    ...  
}
```

(Implementa la clase Casilla de forma similar)

- A partir del dibujo de la Fig., define la clase Barco
  - Un barco se relaciona con 1..4 casillas
  - Una casilla se relaciona con 0..1 barcos



# Relaciones entre Objetos

## Asociación: Ejemplo



```
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;
    }
}
```

- **¿Detectáis alguna posible inconsistencia que no controle este código? Modifica lo que sea necesario.**
- **¿Cambiaría en algo el código de la clase Casilla definida en el ejercicio anterior (aparte del nombre de la referencia a Barco)?**

# Relaciones entre Objetos

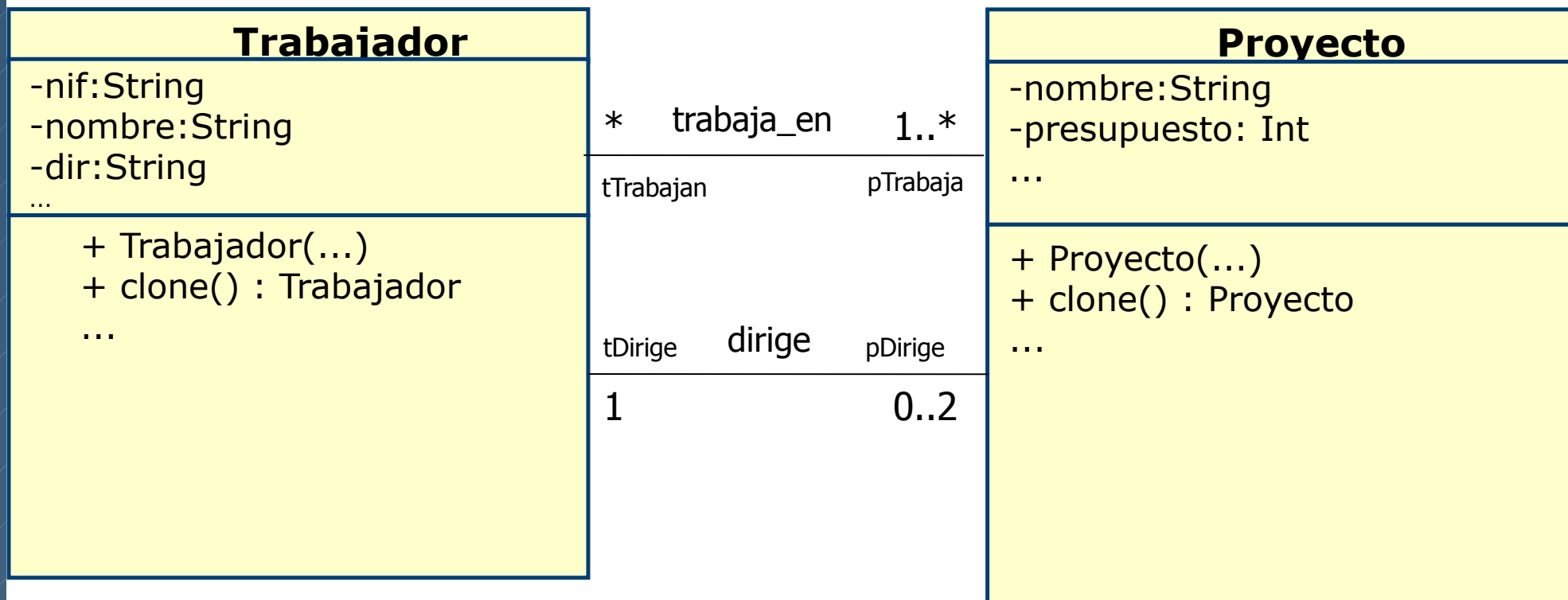
## Asociación: Ejemplo



```
class Barco{
    private static final int MAX_CAS=4;
    private Vector<Casilla> cas;
    // usando Vector del API java.util.*

    public Barco(Vector<Casilla> c) {
        cas=null;
        if (c.size() <= MAX_CAS && c.size() > 0)
            cas = c;
    }
    // nota: lo correcto sería lanzar una excepción (tema 3)
    // si el número de casillas no es correcto
}
```

- A partir del dibujo de la Fig., define las clases Trabajador y Proyecto
  - Un trabajador debe trabajar siempre como mínimo en un proyecto, y dirigir un máximo de dos
  - Un proyecto tiene n trabajadores, y siempre debe tener un director

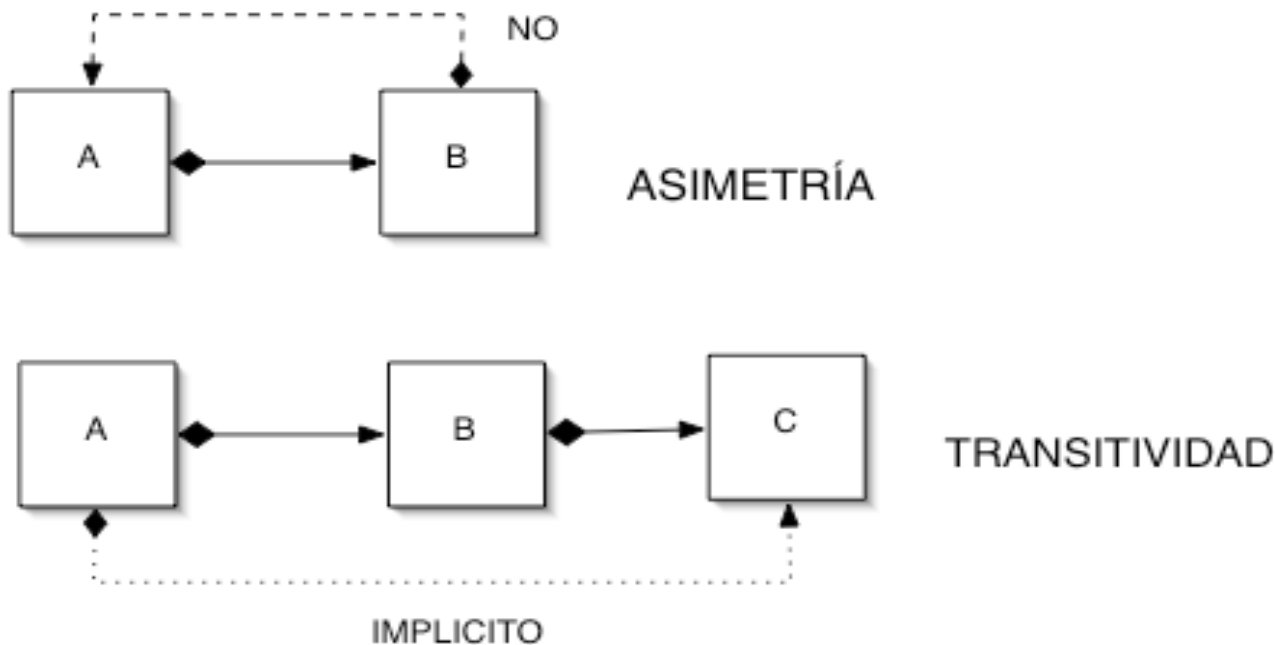




# Relaciones entre Objetos

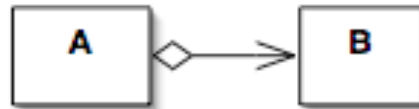
## Todo-Parte

- Una relación Todo-Parte es una relación en la que un objeto forma parte de la naturaleza de otro.
  - Se ve en la vida real: 'A está compuesto de B', 'A tiene B'
  - (en inglés, relaciones 'has-a')
- Asociación vs. Todo-Parte
  - La diferencia entre asociación y relación todo-parte radica en la **asimetría** y **transitividad** presentes en toda relación todo-parte.



- Se distinguen dos tipos de relación todo-parte:

- **Agregación**



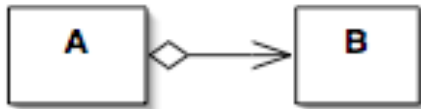
- Asociación binaria que representa una relación todo-parte ('tiene-un', 'es-parte-de', 'pertenece-a')
  - Ejemplo: Un equipo y sus miembros

- **Composición**



- Agregación 'fuerte':
  - Una instancia 'parte' está relacionada, como máximo, con una instancia 'todo' en un instante dado.
  - Cuando un objeto 'todo' es eliminado, también son eliminados sus objetos 'parte' (Es responsabilidad del objeto 'todo' disponer de sus objetos 'parte')
    - Ejemplo: Un libro y sus capítulos

### ¿Agregación o composición?



**¿Puede el objeto parte ser compartido por más de un objeto agregado?**

- No => **disjunta** (Composición)
- Sí => **no disjunta** (Agregación)

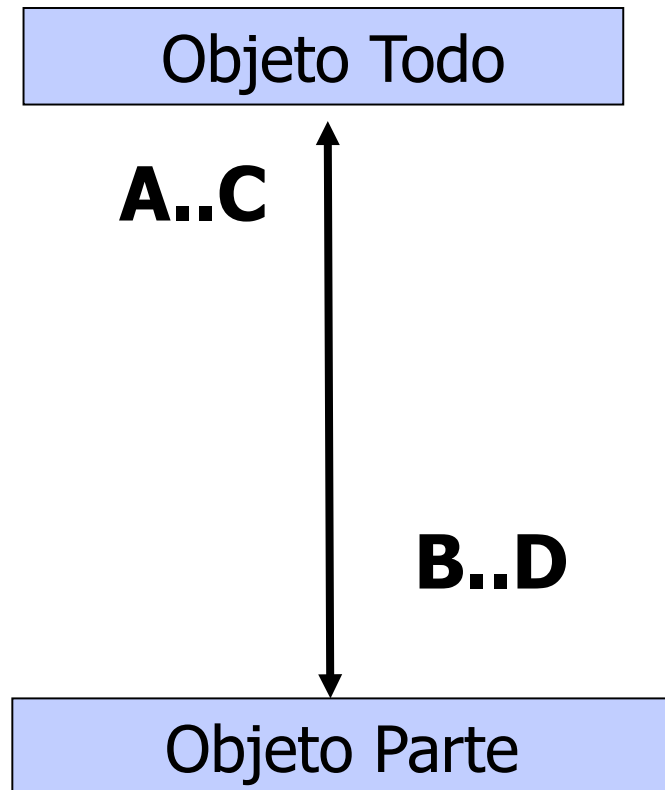
**¿Puede existir un objeto parte sin ser componente de un objeto agregado?**

- Sí => **flexible** (Agregación)
- No => **estricta** (Composición)

# Relaciones entre Objetos

## Caracterización Todo-Parte

- Multiplicidad en una relación todo-parte



**A:** Multiplicidad Mínima

0 → flexible

> 0 → estricta

**B:** Multiplicidad Mínima

0 → nulos permitidos

> 0 → nulos no permitidos

**C:** Multiplicidad Máxima

1 → disjunto

> 1 → no disjunto

**D:** Multiplicidad Máxima

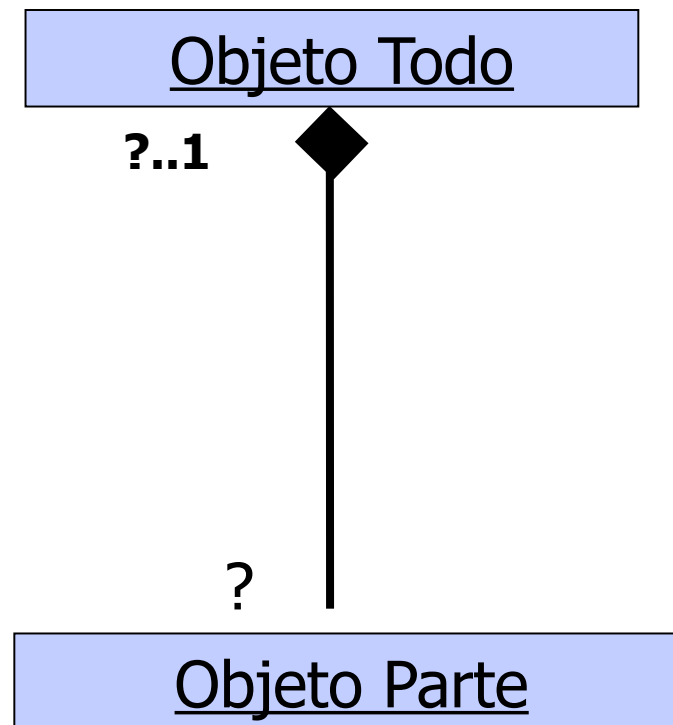
1 → univaluado

> 1 → multivaluado

# Relaciones entre Objetos

## Caracterización composición

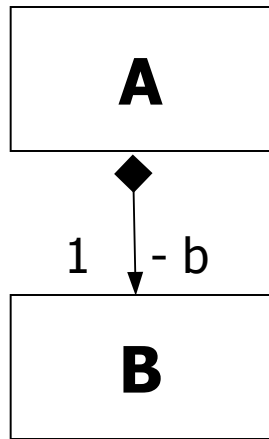
- Si tenemos en cuenta las restricciones 4 y 5, una composición se caracteriza por una cardinalidad máxima de 1 en el objeto compuesto



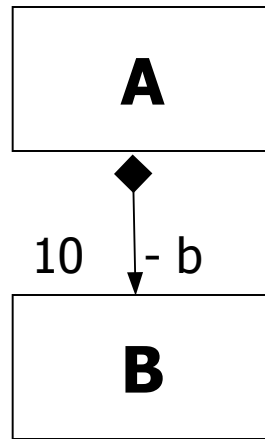
- Agregación: Se implementa como una asociación unidireccional
  - El objeto 'todo' mantiene referencias (posiblemente compartidas con otros objetos) a sus partes agregadas.
- Composición: Dijimos que
  - Es responsabilidad del objeto 'todo' disponer de sus objetos 'parte'...

# Relaciones todo-parte

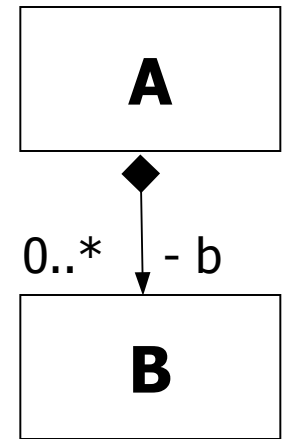
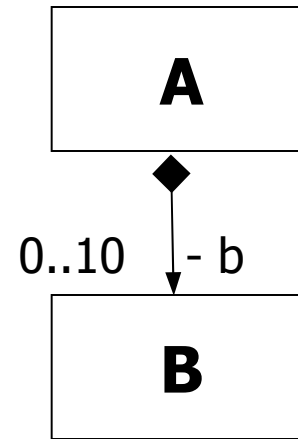
## Implementación (Java)



```
class A {  
    private B b;  
    // b es un  
    // subobjeto  
    ...}
```



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

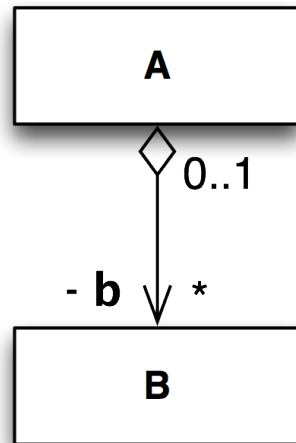


```
class A {  
    private  
        Vector<B> b;  
    ...};
```

La declaración de atributos es la misma para una agregación o una composición.

# Relaciones todo-parte

## Implementación de la agregación



A) El objeto B puede ser creado fuera de A, de forma que pueden existir referencias externas ('objB') al objeto agregado.

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

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

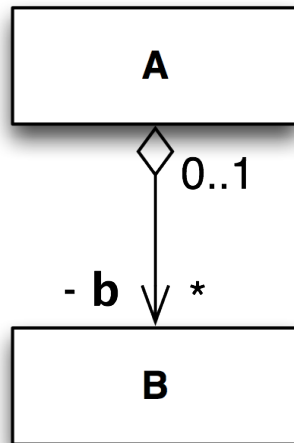
```
// En otro lugar (código cliente),
// quizás fuera de A...
B objB = new B();
if (...) {
    A objA = new A();
    objA.addB(objB);
}
```

B) Cuando 'objA' desaparece, 'objB' sigue existiendo



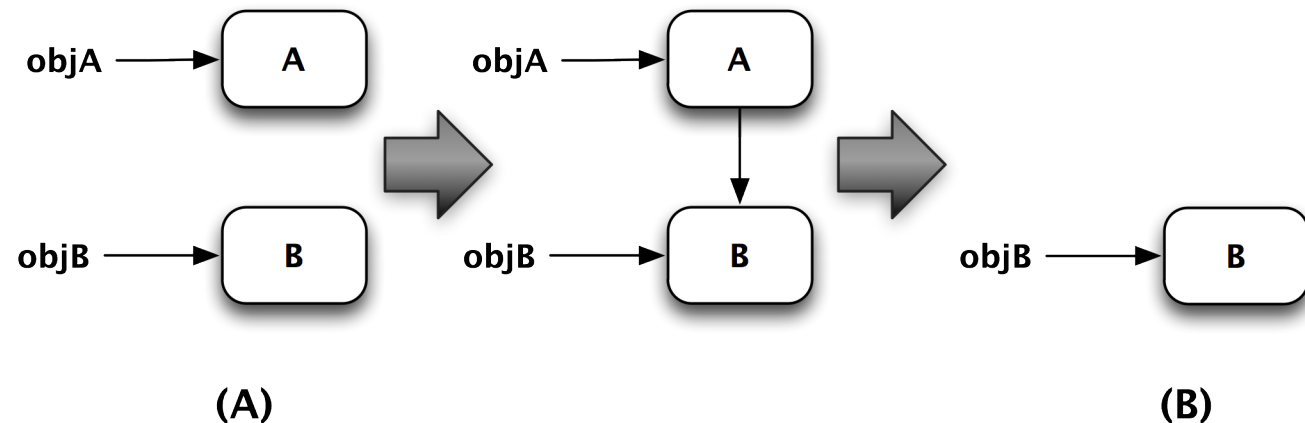
# Relaciones todo-parte

## Implementación de la agregación



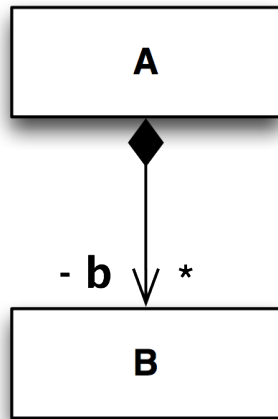
A) El objeto B puede ser creado fuera de A, de forma que pueden existir referencias externas ('objB') al objeto agregado.

B) Cuando 'objA' desaparece, el objeto B sigue existiendo, pues aún hay referencias a él ('objB').



# Relaciones todo-parte

## Implementación de la composición



A) El objeto B es creado 'dentro' de A, de forma que A es el único que mantiene referencias a su componente B.

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

    public A() {}
    public addB(...) {
        b.add(new B(...));
    } '...' es la información necesaria
    para crear B
}
```

```
// En otro lugar (código cliente),
// fuera de A...
```

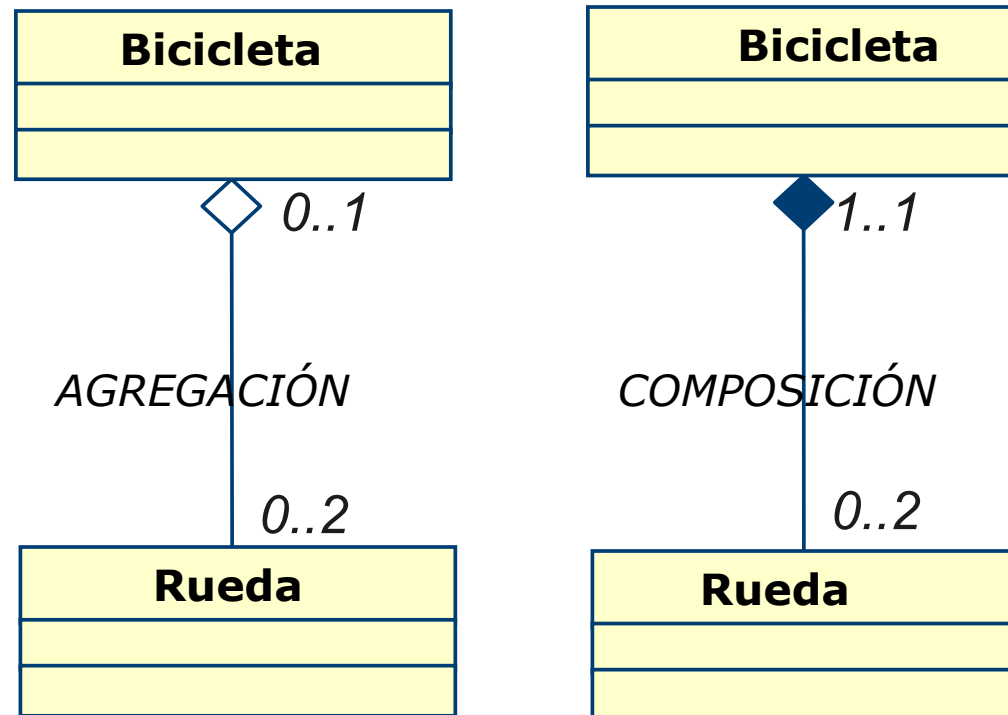
```
if (...) {
    A objA = new A();
    objA.addB(...);
}
```

B) Cuando 'objA' desaparece, también desaparecen los objetos B que forman parte de él

# Relaciones todo-parte

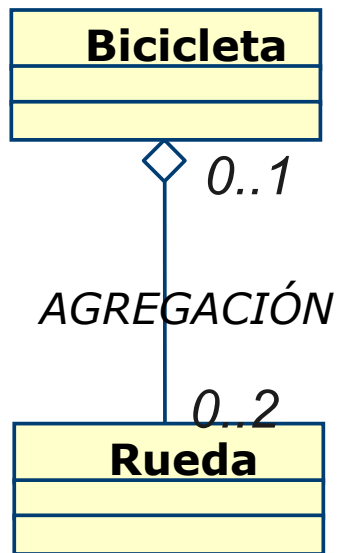
## Ejemplo Bicicleta

Algunas relaciones pueden ser consideradas agregaciones o composiciones, en función del contexto en que se utilicen



# Relaciones todo-parte

## Ejemplo Bicicleta



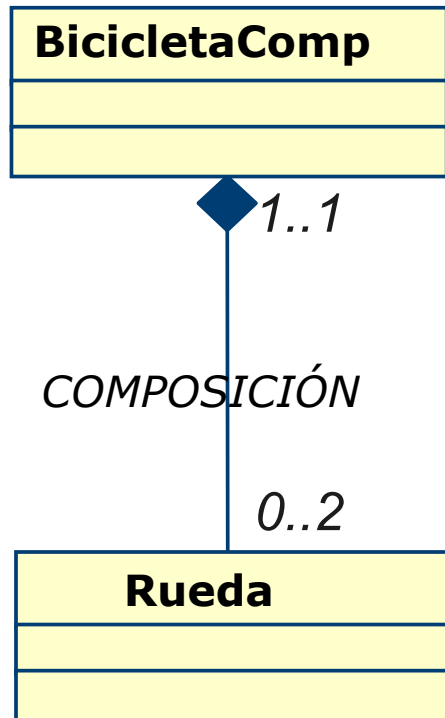
```
class Rueda {
    private String nombre;
    public Rueda(String n){nombre=n;}
}
```

```
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);
    }
}
```

# Relaciones todo-parte

## Ejemplo Bicicleta



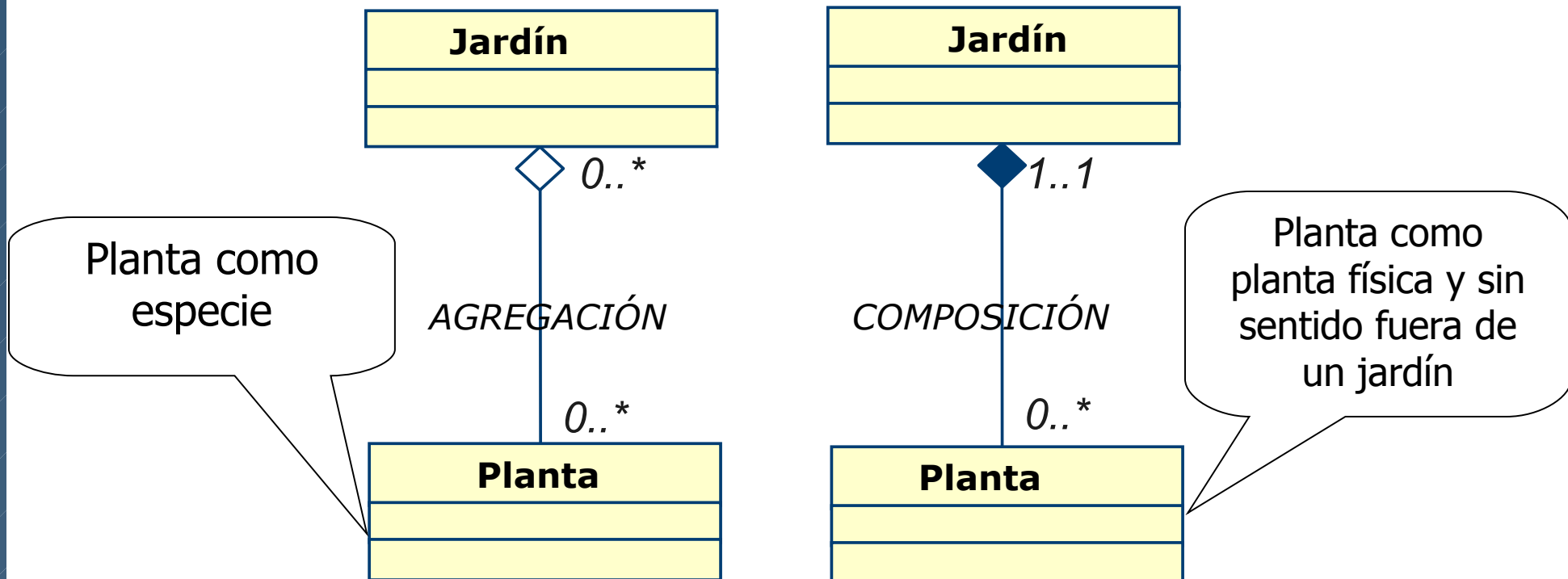
```
class BicicletaComp{
    private static const int MAXR=2;
    private Vector<Rueda> r;
    public BicicletaComp(String p,String s){
        r.add(new Rueda(p));
        r.add(new Rueda(s));
    }
    public static final void main(String[] args) {
        BicicletaComp b2 = new BicicletaComp("1","2");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

        //son ruedas distintas aunque con el mismo nombre
    }
}
```

# Relaciones todo-parte

## Ejemplo Jardín

Observad el diferente significado de la clase Planta en función del tipo de relación que mantiene con Jardín



### Supongamos que tenemos el código

```
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;  
}
```



**¿Qué relación existe entre Jardín y planta?**

# Relaciones todo-parte

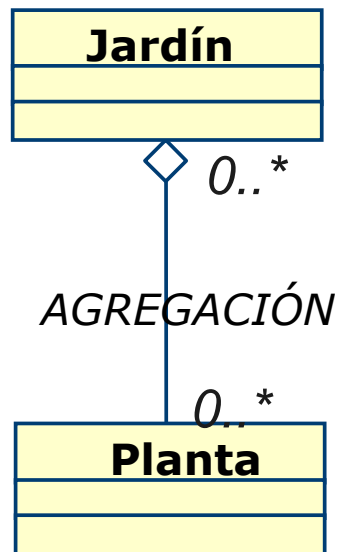
## Ejemplo Jardín



Implementa ahora el método clone() de Jardin



¿Cómo cambiaría el código si decidiésemos implementar el jardín como una agregación de plantas?

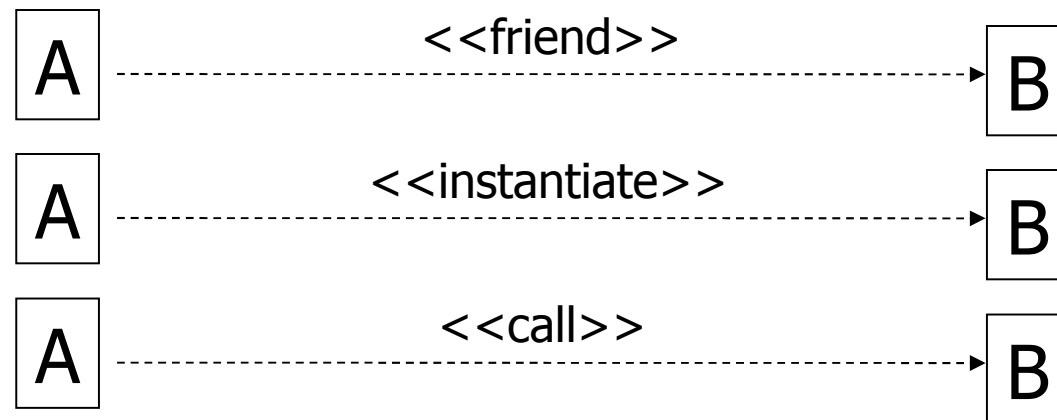


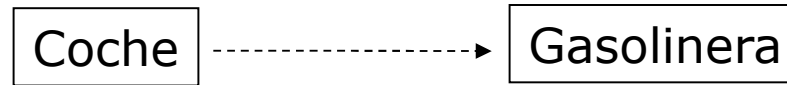
# Relaciones entre Objetos

## Relación de Uso (Dependencia)



- Una clase A **usa** una clase B cuando no contiene datos miembros del tipo especificado por la clase B pero:
  - Utiliza alguna **instancia de la clase B como parámetro** (o variable local) en alguno de sus métodos para realizar una operación.
  - **Accede a sus variables privadas** (clases amigas)
  - Usa algún **método de clase** de B.
- En UML este tipo de relaciones se diseñan mediante **dependencias**.

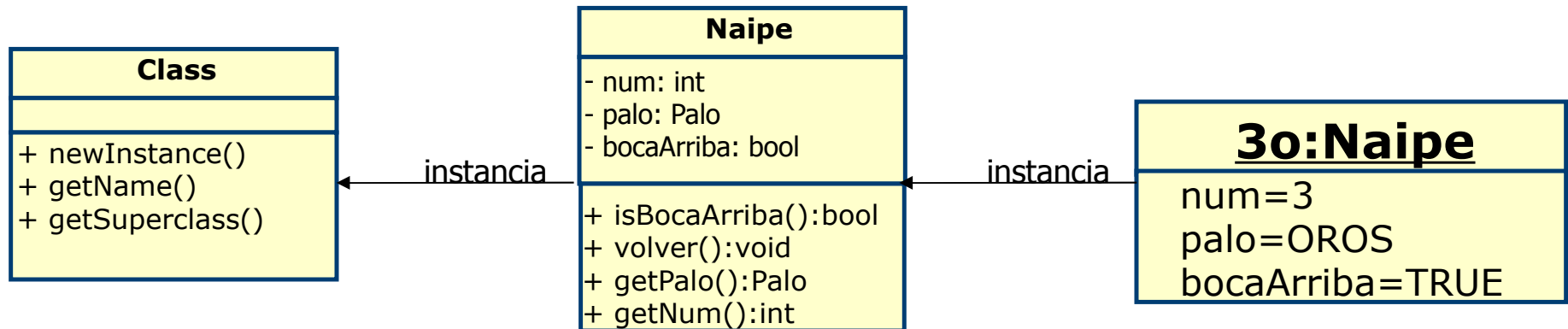




- Supongamos que no nos interesa guardar las gasolineras en las que ha repostado un coche: no es asociación.
- Sin embargo sí existe una interacción:

```
class Coche {  
    Carburante tipoC;  
    float lGaso;  
    float repostar(Gasolinera g, float litros){  
        float importe=g.dispensarGaso(litros,tipoC);  
        if (importe>0.0) //si éxito dispensar  
            lGaso=lGaso+litros;  
        return (importe);  
    }  
}
```

- Existen métodos que se asocian no con métodos sino con clases
  - `new`, `delete`
  - Métodos estáticos
- En Smalltalk, Java y otros lenguajes una clase es una instancia de otra clase, llamada **metaclass**.
  - Por tanto, las clases en sí mismas pueden responder a ciertos mensajes, como es el mensaje de creación de objetos `new`



P. ej. en Java:

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

- Cachero et. al.
  - ***Introducción a la programación orientada a Objetos***
    - Capítulo 2
  
- T. Budd.
  - ***An Introduction to Object-oriented Programming, 3rd ed.***
    - Cap. 4 y 5; cap. 6: caso de estudio en varios LOO
  
- G. Booch.
  - ***Object Oriented Analysis and Design with Applications***
    - Cap. 3 y 4
  
- G. Booch et. al.
  - ***El lenguaje unificado de modelado***. Addison Wesley. 2000
    - Sección 2 (cap. 4-8)