

Atributos y Métodos

Prof. Francisco Velasco Anguita

Dpto. Lenguajes y Sistemas Informáticos
Universidad de Granada

Programación y Diseño Orientado a Objetos

Doble Grado en Ingeniería Informática
y Administración y Dirección de Empresas
(Curso 2024-2025)

Créditos

- Las siguientes imágenes e ilustraciones son libres y se han obtenido de:
 - ▶ Emojis, <https://pixabay.com/images/id-2074153/>
- El resto de imágenes e ilustraciones son de creación propia, al igual que los ejemplos de código

Objetivos

- Aprender la utilidad, significado y uso de:
 - ▶ Atributos de instancia
 - ▶ Atributos de instancia de la clase
 - ▶ Atributos de clase
- Aprender a usar métodos de instancia y de clase
- Aprender las diferencias entre Java y Ruby en cuanto a:
 - ▶ Atributos de clase
 - ▶ Visibilidad privada
- Tomar nota de los errores más frecuentes que soléis cometer
- Usar correctamente las pseudovariables `this` y `self`
- Conocer los especificadores de acceso

Contenidos

1 Atributos y métodos de instancia

2 Atributos y métodos de clase

- Ejemplos

3 Pseudovariables

4 Especificadores de acceso. Visibilidad

- Ejemplos

Atributos de instancia

- La definición de las clases incluye los atributos de instancia que tendrá cada objeto que sea instancia de esa clase
- Los atributos de instancia son variables que están asociadas a cada objeto
- Cada instancia tiene su propio espacio de atributos o variables de instancia.
 - ▶ Así, cada instancia tendrá los mismos atributos que otra instancia de la misma clase, pero en zonas de memoria distintas
- El estado de cada instancia se describe mediante los valores de estos atributos

Java:

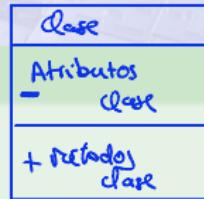
Atributos de instancia

Ejemplo: La clase Persona

```

1   class Persona {
2       private String nombre;
3           // ...
4   }

```

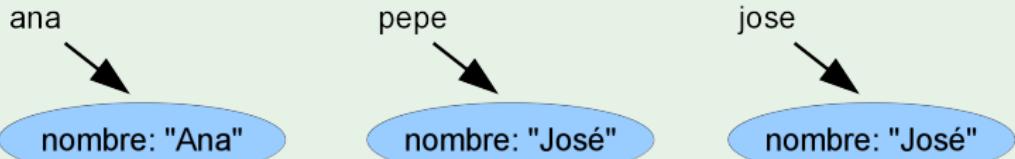


- Todas las instancias de la clase Persona tendrán un atributo denominado nombre
 - ▶ Existirá una variable denominada nombre para cada instancia
 - ▶ Dos instancias de Persona distintas almacenan el nombre en variables distintas (almacenadas en zonas de memoria distintas) aunque se llamen igual

```

1   Persona ana = new Persona ("Ana");
2   Persona pepe = new Persona ("José");
3   Persona jose = new Persona ("José");

```



Métodos de instancia

- Son **funciones o métodos** definidos en una clase y que estarán **asociados a los objetos de dicha clase**.
- Desde los **métodos asociados a un determinado objeto** son accesibles los **atributos de instancia** de dicho objeto.
Tanto para lectura como para escritura.

Ejemplo: La clase Persona

```

1 class Persona {
2     private String nombre;
3     // ...
4     String saludar () {
5         return "Hola, me llamo " + nombre;
6     }
7     void cambiaNombre (String otroNombre) {
8         nombre = otroNombre;
9     }
10 }
```

Ejemplo: La clase Persona

```

1 Persona pepe = new Persona ("José");
2 Persona ana = new Persona ("Ana");
3
4 System.out.println (ana.saludar ());
5 // Hola, me llamo Ana
6
7 ana.cambiaNombre ("Ana Belén");
8
9 System.out.println (ana.saludar ());
10 // Hola, me llamo Ana Belén
11
12 System.out.println (pepe.saludar ());
13 // Hola, me llamo José
```

clase atributo
 Math.PI

Atributos de clase

- Almacenan información que se considera asociada a la propia clase y no a cada instancia
- Son por tanto variables asociadas a la clase y globales a todas las instancias de esa clase
- Cada atributo de clase existe de forma única

```
class Persona
```

```
    numPersonas: 2
```

ana

nombre: "Ana"

pepe

nombre: "José"

Ejemplo: Contador de instancias

- Tanto ana, como pepe tienen accesible el atributo de clase numPersonas
- Cuando su valor cambia, lo hace para todas las instancias
- El atributo es único, está en una zona de memoria asociada a la clase, no a las instancias

Atributos de clase

→ Diseño ←

- ¿Cuándo usarlos?

- ▶ Se debe pensar en ellos cuando se necesite almacenar información que **siempre** va a ser **común** a **todas** las instancias de la clase
- ▶ No tendría sentido que cada instancia de la clase (Persona en el ejemplo anterior) guardase una copia del valor almacenado en ese atributo (numPersonas)
- ▶ Esto además haría su actualización extremadamente costosa.

- Ejemplos típicos:

- ▶ Contador de instancias
- ▶ Constantes
- ▶ Para evitar el uso de *números mágicos*

★ ¿Quién sabría decirme qué es un número mágico? → *5, 10, ...*

Usar números mágicos en los exámenes será **penalizado**

• Tanto en Java como en Ruby hay que inicializar lo que se declara

Ejemplo

Java: Constante vs. número mágico

```

1 class Factura {
2
3     float calculaIVA (float baseImponible) {
4         return baseImponible * 0.21;    // Número mágico, fuente de errores
5     }
6 }
7
8 // Otro modo de diseñarlo
9
10 class GestionTributaria {
11     static float IVA = 0.21;        // Variable de clase
12 }
13 } → es de clase
14
15 class Factura {
16
17     float calculaIVA (float baseImponible) {
18         return baseImponible * GestionTributaria.IVA;    // Uso de la variable de clase
19     }
20 }
```

•) Ruby: @@ → atributos de clase
def self.... → enclados clase
Java → uso static

Métodos de clase

- Son funciones y procedimientos asociados a la propia clase
- Es habitual que accedan/actualicen atributos de clase
- No se puede acceder *directamente* a atributos/métodos de instancia desde un método de clase

► Sería necesario solicitar ese elemento a una instancia concreta

Java: Contador de instancias

```

1 class Persona {
2     // atributo y método de clase
3     static private int numPersonas = 0;
4     static int getNumPersonas () {
5         return numPersonas;
6     }
7     // atributo de instancia
8     private String nombre;
9     // inicializador
10    Persona (String unNombre) {
11        nombre = unNombre;
12        numPersonas++;
13    }
14 }
```

Ruby: Contador de instancias

```

1 class Persona
2     # atributo y método de clase
3     @@num_personas = 0
4     def self.num_personas
5         @@num_personas
6     end
7     # inicializador
8     def initialize (un_nombre)
9         # atributo de instancia
10        @nombre = un_nombre
11        @@num_personas += 1
12    end
13 end
14 }
```

★ ¿Cómo se mostraría el número de instancias creadas?
 añadir un controlador que devuelva el valor de esa variable

Atributos de clase: Particularidad de Ruby

- Existen **dos tipos** de atributos de clase
 - ▶ Atributos de clase (`@@atributo_de_clase`)
 - ▶ Atributos de instancia de la clase (`@atributo_instancia_clase`)
- Los atributos de clase son accesibles directamente desde el ámbito de instancia.
 - ▶ Los atributos de instancia de la clase, no
- Los atributos de clase se comparten con las subclases (herencia).
Esto puede ser muy peligroso
 - ▶ Los atributos de instancia de la clase, no

clase clase

③ @ atributo - clase = 0 → accesible desde cualquier ámbito

④ atributo - instancia - clase = 1 → solo son accesibles desde el ámbito de CLASE.

def initialize

⑤ atributo - instancia = i # de los objetos ✗

end

end

) (como se que atributo es ???

- Depende del ámbito en el que nte.

✗ Intuir método instancia → amb. instancia

Nosotros → ámbito clase

Errores frecuentes en Ruby

- Confundir atributos de instancia con atributos de instancia de la clase
 - ▶ Hay que tener en cuenta en qué ámbito se está
 - ▶ En una clase, cualquier punto dentro de un método de instancia está en ámbito de instancia, lo demás está en ámbito de clase
 - ▶ En un **ámbito de instancia**,
@variable alude a un **atributo de instancia**
 - ▶ En un **ámbito de clase**,
@variable alude a un **atributo de instancia de la clase**
- Añadir **atributos** de instancia, atributos de instancia de la clase o atributos de clase **cuando hay que usar variables locales**
 - ▶ Las variables locales y los parámetros de método no llevan @
- Estos errores, en los exámenes, serán **penalizados**

Ejemplos

Ruby: Confusión entre atributos

```

1 class Clase
2   @@variable = "De clase"
3   @variable = "De instancia de la clase"
4
5   def initialize
6     @variable = "De instancia"
7   end
8   → no tiene self
9   def muestraValores (variable)
10    puts @@variable
11    puts @variable
12    puts variable
13  end
14
15  def self.muestraValores
16    variable = "Local"
17    puts @@variable
18    puts @variable
19    puts variable
20  end
21 end
22 objeto = Clase.new
23 objeto.muestraValores ("Parámetro")
24 Clase.muestraValores

```

★ ¿Cuál es el resultado de ejecutar este programa?

6,10,11,12 → ámbito instancia

→ Diseño ←

- Los nombres de las variables deben ser significativos
- Debe evitarse nombrar a cosas distintas con el mismo nombre
- En el ejemplo anterior  no se han seguido estas recomendaciones por motivos docentes

Ejemplos

Java: Atributos y métodos de clase y de instancia, variables locales

```
1 public class Persona {  
2  
3     private static final int MAYORIAEDAD=18; // Atributo de clase  
4     private LocalDateTime fechaNacimiento; // Atributo de instancia  
5  
6     Persona(LocalDateTime fecha) {  
7         fechaNacimiento=fecha;  
8     }  
9  
10    public boolean mayorDeEdad() { // Método de instancia  
11        LocalDateTime ahora= LocalDateTime.now(); //Llamada a método de clase  
12        // "ahora" es una variable local  
13  
14        //Años completos transcurridos  
15        long edad=ChronoUnit.YEARS.between(fechaNacimiento ,ahora);  
16  
17        return (edad>=MAYORIAEDAD);  
18    }  
19 }
```

★ ¿Qué efecto tiene la palabra final en la declaración de MAYORIAEDAD?

Indica que no se va a modificar

Ejemplos

Ruby: Atributos y métodos de clase y de instancia, variables locales

```
1  require 'date'
2
3  class Persona
4      @@MAYORIA_EDAD = 18 # Atributo de clase
5
6  def initialize(fecha)
7      @fecha_nacimiento=fecha # Atributo de instancia
8  end
9
10 def mayor_de_edad # Método de instancia
11     ahora = Date.today # "ahora" es una variable local
12     edad = ahora.year - @fecha_nacimiento.year - 1
13     if (ahora.month > @fecha_nacimiento.month)
14         edad += 1
15     else
16         if (ahora.month == @fecha_nacimiento.month)
17             if (ahora.day >= @fecha_nacimiento.day)
18                 edad += 1
19             end
20         end
21     end
22     return (edad >= @@MAYORIA_EDAD)
23 end
24 end
```

★ ¿Qué significa la línea 1?

Ejemplos

Ruby: Atributos y métodos de clase y de instancia, variables locales

```

1  class Persona
2      @MAYORIA_EDAD=18 # Atributo de instancia de la clase
3
4      def self.edad_legal # Método de clase (Persona.edad_legal)
5          @MAYORIA_EDAD
6      end
7
8      def initialize(fecha)
9          @fecha_nacimiento = fecha # Atributo de instancia
10     end
11
12     def mayor_de_edad # Método de instancia
13         ahora = Date.today
14         edad = ahora.year - @fecha_nacimiento.year - 1
15         if (ahora.month > @fecha_nacimiento.month)
16             edad += 1
17         else
18             if (ahora.month == @fecha_nacimiento.month) && (ahora.day >= @fecha_nacimiento.day)
19                 edad += 1
20             end
21         end
22         return (edad >= self.class.edad_legal) # (Persona.edad_legal)
23     end
24 end

```

→ No yo que uso variables de instancia

- ★ ¿Se puede prescindir del método `edad_legal`? ★ ¿Cómo quedaría la línea 22?

Sin un error hered

edad >= @ MAYORIA_EDAD

✓ ya que es de instancia, cuando
solo hay definido de instancia de clase.

.) Puedo usar un consultar de clase

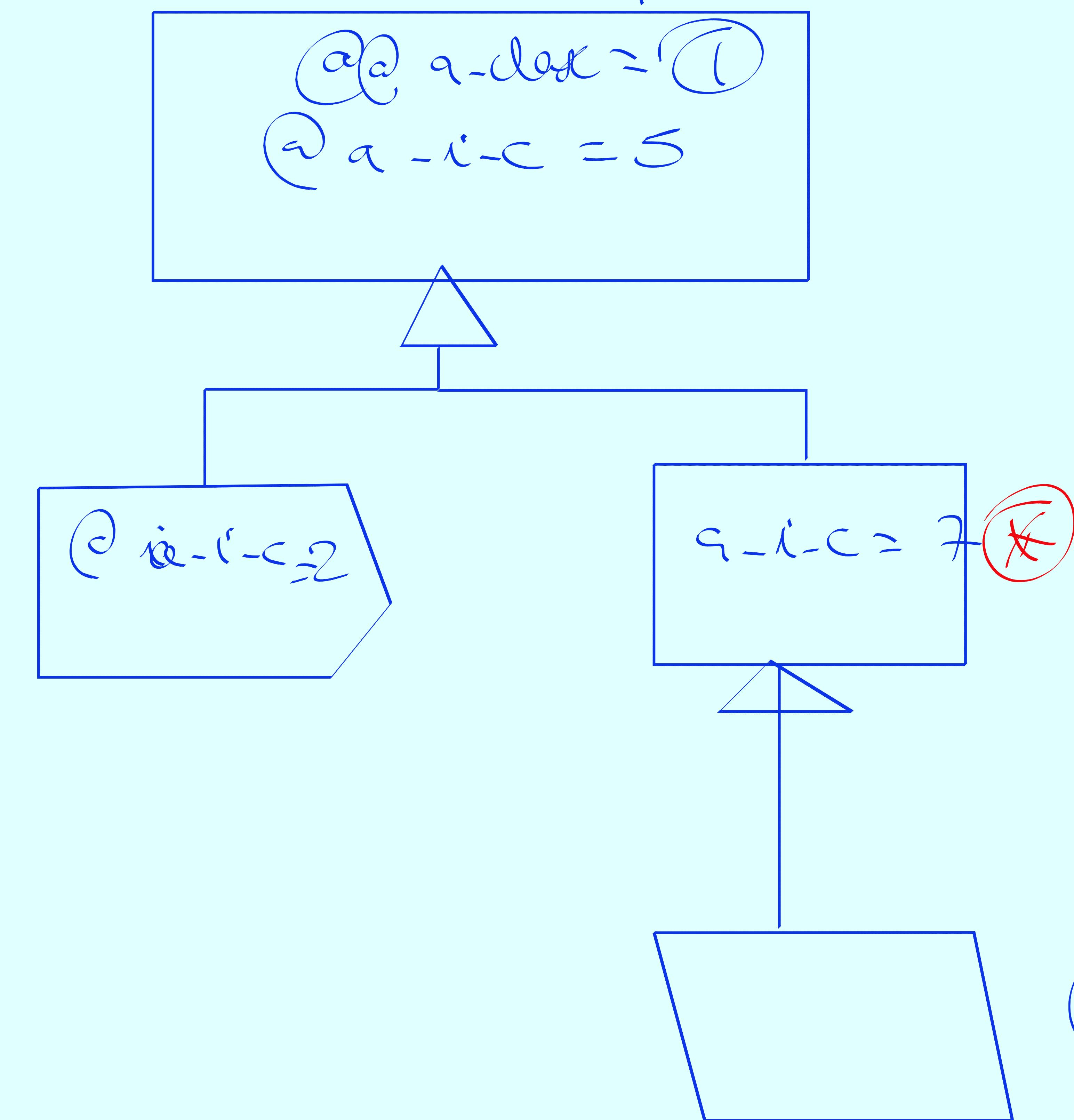
.) Los de @ puedo acceder solo desde
la clase ipro puedo hacer un método.

self → alude al punto objetos

self.dos → "objeto dime tu clase".

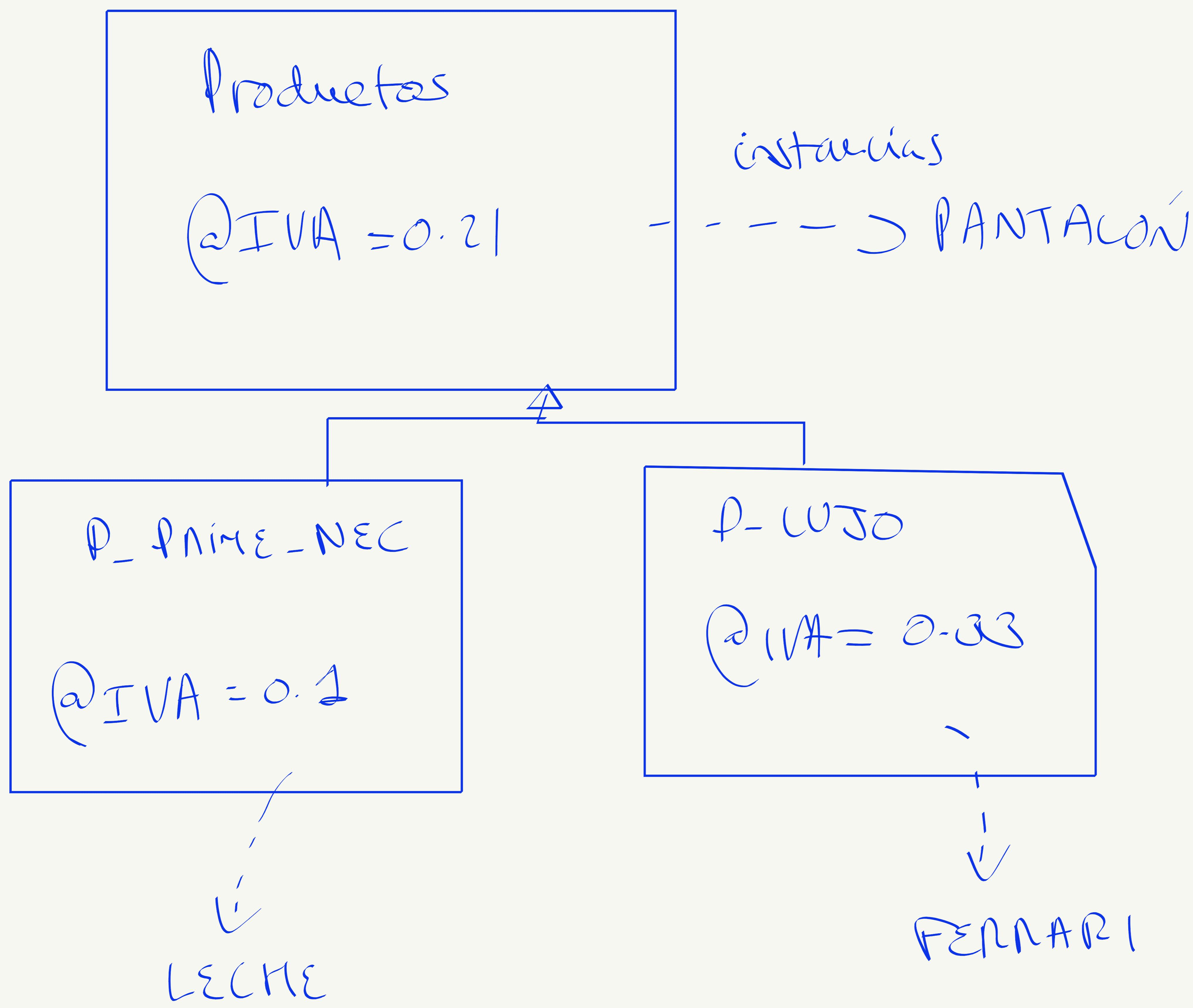
Objetos prácticos:

clases



Advantage: ... puede tener varios con \neq valores

: @@ si modifijo uno, modifiko todos.



c) Neurinas al $\alpha\beta$ cuando $\gamma\delta$ = para todos
 y pueden cambiarse ya que son válido.

Ejemplos

Ruby: Atributos y métodos de clase y de instancia, variables locales

```

1 class Producto
2   @@iva = 21 → Accesible desde cualquier sitio
3
4   def initialize(precio, nombre)
5     @precio = precio
6     @nombre = nombre
7   end
8
9   def instanciaSetIVA(iv)
10    # Acceso directo a un atributo de clase desde un método de instancia
11    @@iva = iv
12    # Esto no es posible con atributos de instancia de la clase
13  end
14
15  def self.claseSetIVA(iv) → Método de la clase que permite cambiar IVA
16    # Acceso directo a un atributo de clase desde un método de clase
17    @@iva = iv
18  end
19
20  def to_s
21    "nombre: #{@nombre}, precio: #{@precio}, iva: #{@@iva}"
22  end
23 end

```

← self = Producto. @@IVA = otro valor

★ ¿Qué diferencia hay entre los métodos de las líneas 9 y 15?

Ejemplos

Ruby: Atributos y métodos de clase y de instancia, variables locales

```
1 # Usando la clase anterior
2
3 p = Producto.new(2, "cosa")
4 puts p.to_s
5
6 p.instanciaSetIVA(25)
7 puts p.to_s
8
9 Producto.claseSetIVA(27)
10 puts p.to_s
11
12 # Lo siguiente no funciona en Ruby
13 # En cualquier caso NO es recomendable
14
15 p.claseSetIVA(50) # esto no funciona en Ruby
16 puts p.to_s
```

★ ¿Qué salida producen las líneas 4, 7 y 10?

JAVA

```
class Producto {  
  
    private static int IVA = 21;  
  
    public static void setIVA(int i){  
        IVA = i;  
    }  
};  
}
```

```
Producto.setIVA(15);
```

```
Producto p = new Producto(...)
```

```
p.setIVA(15); // en Java permitido
```

↓
No recomendable

Recomendable: encircela o la clase como parámetro

class Products

@@ IVA = 2

def self.setIVA(x)

@@ IVA = i

end

:

:

end

Products.setIVA(15)

p = Products.new(...)

p.setIVA(15) #~~ERROR~~

Pseudovariables

- Existen palabras reservadas que referencian al propio objeto, o a la clase
 - ▶ Java: `this` (también en C++, C#, etc.)
 - ▶ Ruby: `self` (también en Python, Rust, etc.)

Java: this Dos usos

```

1 class Persona {
2     private String nombre;
3
4     Persona (String nombre) {
5         this.nombre = nombre;    ①
6     }
7
8     Persona () {
9         this ("Anónimo");
10    }  ② Uso común
11 }
```

Dentro de un
const se llama a otro

- ★ Significado de `this` en las líneas 5 y 9
- ★ Significado de `self` en las líneas 3 y 14

Ruby: self (¡mismo)

```

1 class Persona
2     @@MAYORIA_EDAD = 18
3
4     def self.mayoria_edad
5         return @@MAYORIA_EDAD
6     end
7
8     def initialize (nombre)
9         @nombre = nombre
10    end
11
12    def nombre
13        return @nombre
14    end
15
16    def prueba (nombre)
17        puts nombre
18        puts self.nombre
19    end
20 end
```

ponle el
palámetru
otro nombre.

Especificadores de acceso (Visibilidad)

- Existen distintos niveles de acceso a atributos y métodos
- A este respecto hay diferencias importantes entre lenguajes
- En general:
 - ▶ **Privado:** sin acceso desde otra clase y/o desde otra instancia
 - ▶ **Paquete:** sin restricciones dentro del mismo paquete
(no procede en Ruby)
 - ▶ **Público:** sin restricciones de acceso
- Este tema se abordará con detalle en otra lección

Acceso privado: Diferencias entre Java y Ruby

• Java

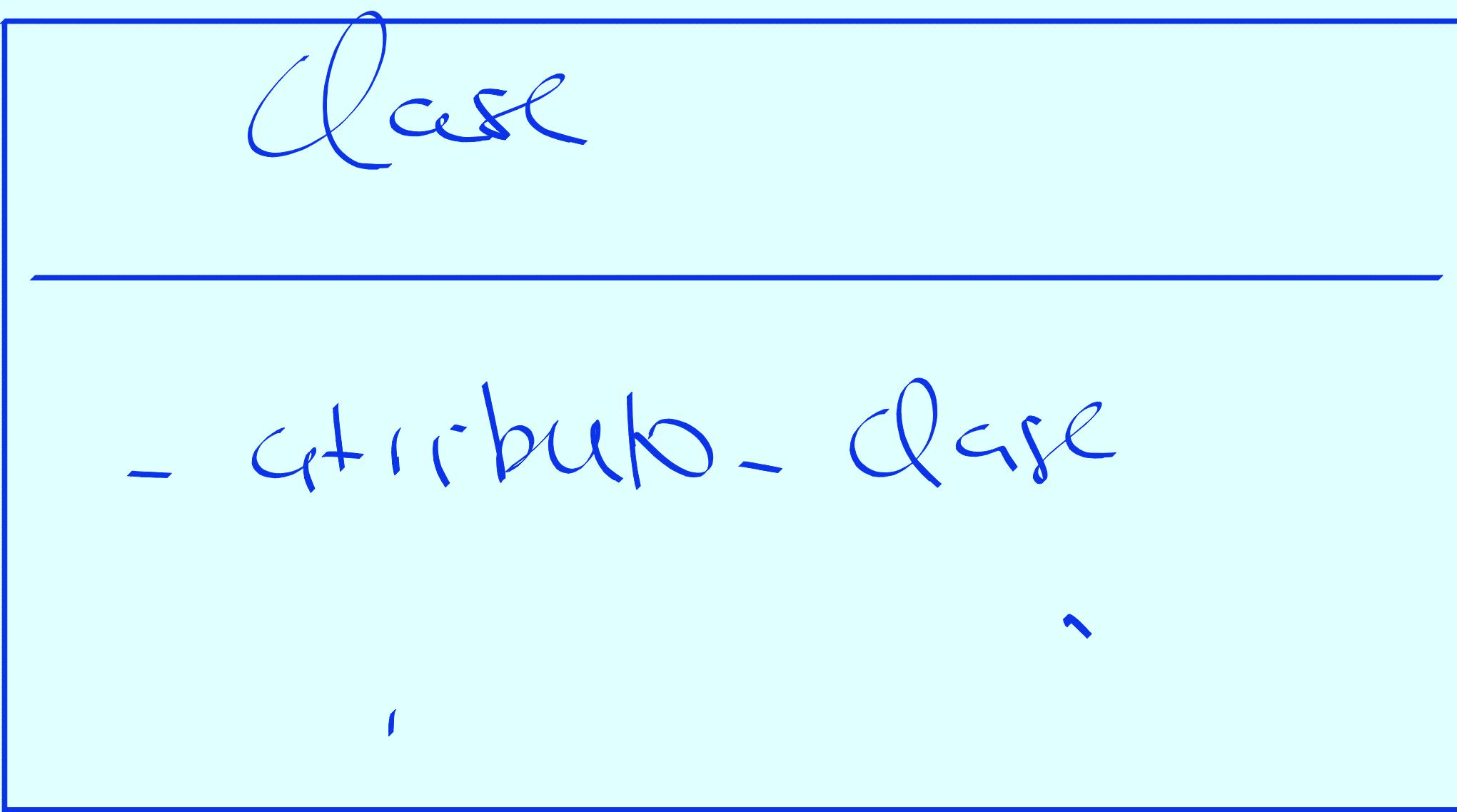
se puede acceder a elementos **privados** (métodos y atributos)

- ▶ Desde una instancia a otra instancia de la misma clase
- ▶ Desde el ámbito de clase a una instancia de esa clase
- ▶ Desde el ámbito de instancia a la clase de la que se es instancia

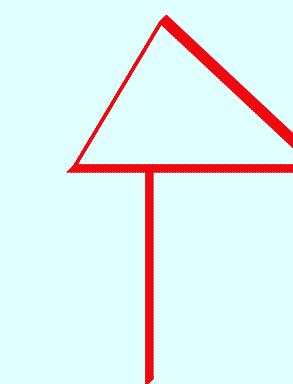
• Ruby

- ▶ Todo lo anterior no está permitido en Ruby
- ▶ Los atributos siempre son privados

JAVA

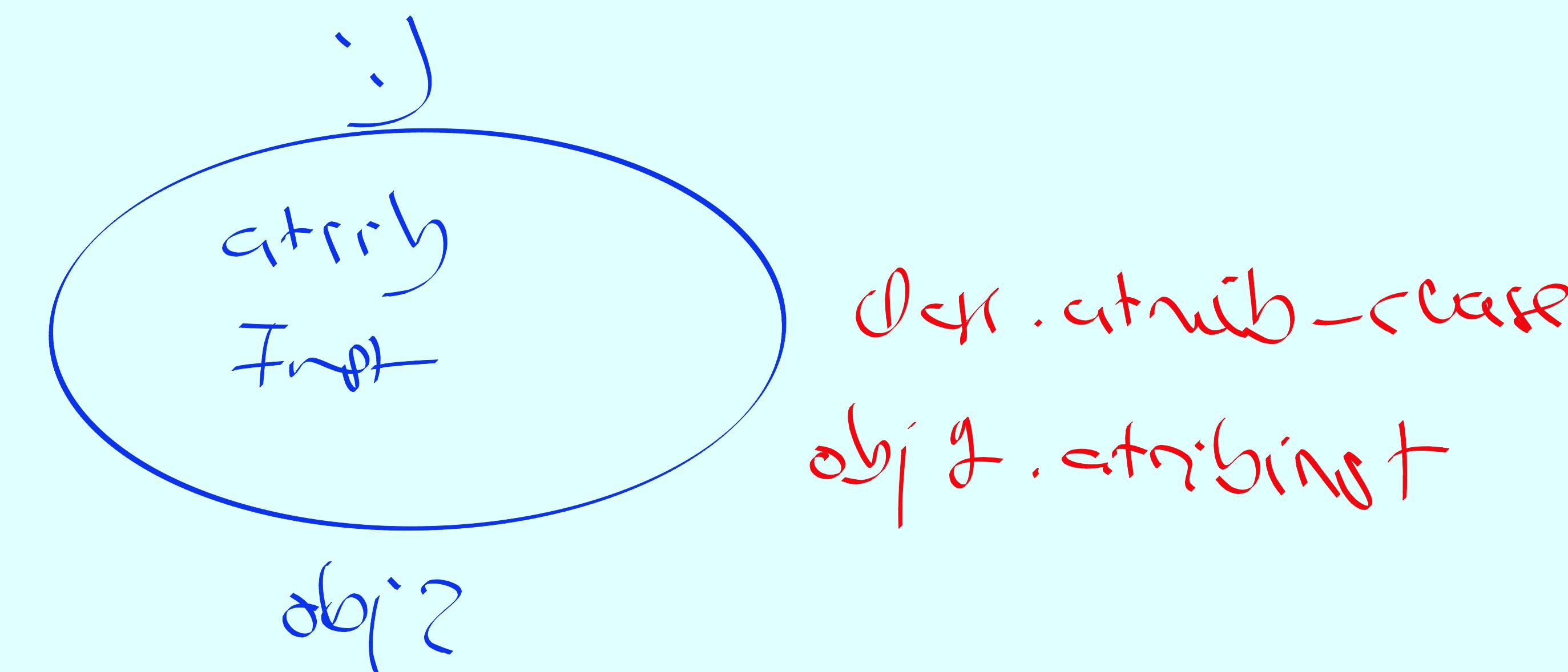
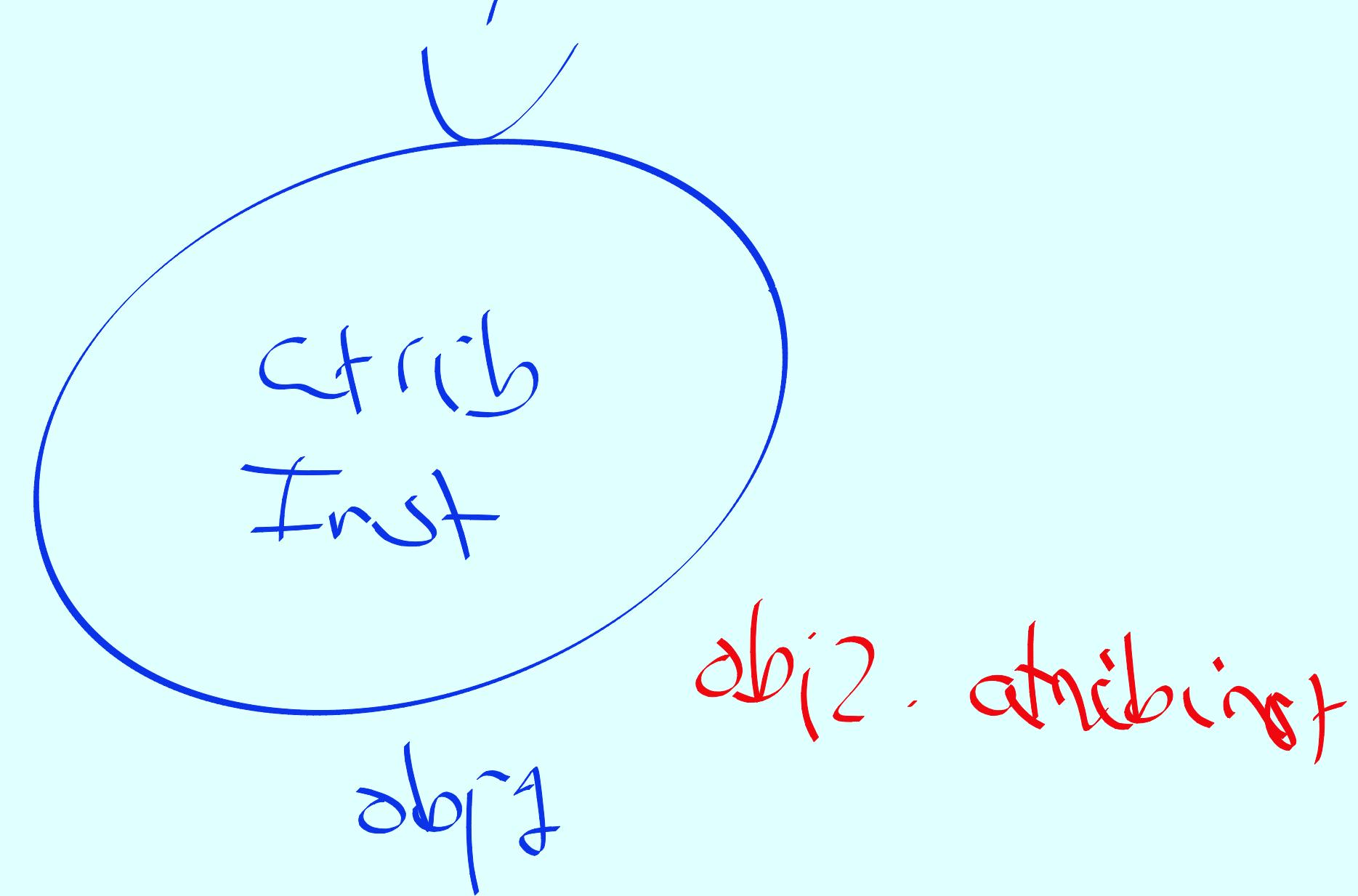


En Ruby NADA



* Puedo acceder
a otros que
son privados

instancias



Obj.attrib - clase
obj2.attribinst

Java

```
class Person  
    private String nombre;  
  
    Person (String n) {  
        nombre = n;  
    }  
  
    Person (Person otro) {  
        this.nombre = otro.nombre  
    }  
}
```

Ruby

```
class Person  
    # No se declaran atributos  
    def initialize(n)  
        @nombre = n  
    end  
    def copia(otro) → method	instance  
        @nombre = otro.nombre # error  
    end
```

Ejercicio → Posible solución del error anterior

Prado hay queridos

Ejemplos de visibilidad

Ruby: Visibilidad

```
1 class Prueba
2
3   def self.metodoClasePublico
4     puts "público de clase"
5   end
6
7   private # solo afecta a los métodos de instancia
8   def self.metodoClasePrivado # sigue siendo público
9     puts "privado de clase"
10  end
11
12  def metodoInstanciaPrivado
13    puts "privado de instancia"
14  end
15
16  # Así también se hace privado el método de instancia
17  private :metodoInstanciaPrivado
18
19  # Así se hacen privados los métodos de clase
20  private_class_method :metodoClasePrivado
21 end
22
23 Prueba.metodoClasePublico
24 #Prueba.metodoClasePrivado          # Error , es privado
25 #Prueba.metodoInstanciaPrivado      # Error , es de instancia
26 #Prueba.new.metodoInstanciaPrivado # Error , privado
27 #Prueba.new.metodoClasePublico    # Error , es de clase
```

Ejemplos de visibilidad

Java: Visibilidad

```
1 public class UnaClase {  
2  
3     public void metodoPublico() { System.out.println("Público"); }  
4  
5     private void metodoPrivado() { System.out.println("Privado"); }  
6  
7     // Todo esto funciona en Java aunque llame la atención  
8     public void usoDentroDeClase() {  
9         metodoPrivado();  
10    }  
11  
12    public void usoConOtroObjeto() {  
13        UnaClase obj2 = new UnaClase();  
14        obj2.metodoPrivado();  
15    }  
16  
17    public static void main(String []args) { // Seguimos en UnaClase  
18        UnaClase obj = new UnaClase();  
19        obj.metodoPublico();  
20        obj.metodoPrivado();  
21        obj.usoDentroDeClase();  
22        obj.usoConOtroObjeto();  
23    }  
24 }
```

Ejemplos de visibilidad

Ruby: Visibilidad

```
1 class UnaClase
2
3   def metodoPublico
4     puts "Publico"
5   end
6
7   def usoDentroDeClase
8     metodoPrivado
9   end
10
11  def usoConOtroObjeto
12    obj2 = UnaClase.new
13    # obj2.metodoPrivado # error, privado de otra instancia
14  end
15
16  private
17  def metodoPrivado
18    puts "Privado"
19  end
20 end
21
22 obj = UnaClase.new
23 obj.metodoPublico
24 # obj.metodoPrivado # error, privado
25 obj.usoDentroDeClase
26 obj.usoConOtroObjeto
```

Visibilidad

→ Diseño ←

- ¿Qué visibilidad asignar a atributos y métodos?
 - ▶ Por regla general, la más restrictiva.
 - ▶ Privada para los atributos
 - ★ Para los que necesiten ser leídos desde fuera de la clase, se creará un método con visibilidad de paquete o público (según corresponda) que proporcionará dicho atributo *al exterior* (**consultor**)
 - ★ ¡Cuidado! Si lo que se proporciona es una **referencia**, el atributo podría ser modificado desde fuera de la clase
 - ★ Para los que necesiten ser modificados desde fuera de la clase, se creará un método con la visibilidad adecuada que reciba los parámetros necesarios y, tras realizar las comprobaciones pertinentes, realice la modificación (**modificador**)
 - ★ Los atributos de un objeto no deberían ser modificados por métodos distintos de los propios de dicho objeto (o de su clase)
 - ★ Solo se crearán los consultores y modificadores necesarios, y con la visibilidad más restrictiva que sea posible

Atributos y Métodos

Prof. Francisco Velasco Anguita

Dpto. Lenguajes y Sistemas Informáticos
Universidad de Granada

Programación y Diseño Orientado a Objetos

Doble Grado en Ingeniería Informática
y Administración y Dirección de Empresas
(Curso 2024-2025)

Construcción de objetos

Prof. Francisco Velasco Anguita

Dpto. Lenguajes y Sistemas Informáticos
Universidad de Granada

Programación y Diseño Orientado a Objetos

Doble Grado en Ingeniería Informática
y Administración y Dirección de Empresas
(Curso 2024-2025)

Créditos

- Las siguientes imágenes e ilustraciones son libres y se han obtenido de:
 - ▶ Emojis, <https://pixabay.com/images/id-2074153/>
- El resto de imágenes e ilustraciones son de creación propia, al igual que los ejemplos de código

Objetivos

- Saber diseñar e implementar constructores
- Saber cómo crear varios constructores para una misma clase, tanto en Java como en Ruby
- Saber cómo reutilizar código que sea común a varios constructores
- Conocer cómo se libera la memoria ocupada por los objetos cuando dejan de ser útiles

Contenidos

1 Constructores

- Java
- Ruby

2 Constructores de copia

- Java
- Ruby

3 Memoria dinámica y pila

Cuestiones generales

- Antes de usar los objetos es necesario crearlos
- La creación implica la reserva de memoria y la inicialización
- Normalmente el programador no tiene que ocuparse de la reserva de memoria en sí misma, pero sí de la inicialización
- En algunos lenguajes el programador decide el lugar donde se alojará cada objeto (pila o *heap*)

Constructores

- Los lenguajes orientados a objetos suelen disponer de unos métodos especiales denominados **constructores**
- A pesar de su nombre, estos métodos **solo se encargan de la inicialización de las instancias**

Se deben inicializar **TODOS** los atributos de instancia

- No son métodos de instancia y no especifican ningún tipo de retorno
- Existen diferencias importantes a este respecto en los distintos lenguajes de programación orientados a objetos

Clases-plantilla / Clases-objeto

- Con relación a los constructores:
- **Clases-plantilla**
 - ▶ En muchos casos tienen el mismo nombre de la clase
 - ▶ Son invocados automáticamente utilizando la palabra reservada `new`
- **Clases-objeto**
 - ▶ Pueden tener un nombre arbitrario
 - ▶ Suelen ser métodos de clase

Java

- Tienen el mismo nombre que la clase y no devuelven nada (tampoco `void`)
- Los constructores se utilizan únicamente para **asegurar la inicialización de los atributos**
- Al permitir la sobrecarga de métodos, **puede haber varios**, con **distintos parámetros**
- **Se puede reutilizar** un constructor desde otro constructor
- Si no se crea ningún constructor existe **uno por defecto** sin parámetros
- Para construir un objeto se antepone la palabra reservada **new** al nombre de la clase

Ejemplos

Java: Constructor básico

```
1 class Point3D {  
2  
3     // Atributos de instancia  
4     private int x;  
5     private int y;  
6     private int z;  
7  
8     Point3D (int a,int b,int c) { // Constructor  
9         // Se inicializan los atributos de instancia , TODOS  
10        x = a;  
11        y = b;  
12        z = c;  
13    }  
14 }
```

Ejemplos

Java: Clase con varios constructores y código común

```
1 class RestrictedPoint3D {  
2     private static int LIMITMAX = 100; // Atributos de clase  
3     private static int LIMITMIN = 0;  
4     private int x; // Atributos de instancia  
5     private int y;  
6     private int z;  
7  
8     private int restricToRange (int a) { // Método de instancia  
9         int result = Math.max (LIMITMIN, a);  
10        result = Math.min (result, LIMITMAX);  
11        return result;  
12    }  
13  
14    RestrictedPoint3D (int x, int y, int z) { // Constructor  
15        this.x = restricToRange (x);  
16        this.y = restricToRange (y);  
17        this.z = restricToRange (z);  
18        // Debido a la igualdad de nombres,  
19        // es necesario usar "this" para referirse a los atributos  
20    }  
21  
22    RestrictedPoint3D (int x, int y) { // Constructor  
23        this (x, y, 0); // Se llama al otro constructor  
24    }  
25 }
```

★ ¿Qué me decís sobre el método max?

Ejemplos

Java: Uso de la clase anterior

```
1 public static void main (String [] args) {  
2     RestrictedPoint3D p1 = new RestrictedPoint3D (-1, 101, -2000);  
3     RestrictedPoint3D p2 = new RestrictedPoint3D (1, 99);  
4     RestrictedPoint3D p3 = new RestrictedPoint3D (50, 51, 52);  
5     RestrictedPoint3D p4 = new RestrictedPoint3D (-2000, 50, 2000);  
6 }
```

- ★ ¿Cuál es el estado de cada punto creado?
- ★ ¿Qué métodos son llamados en cada construcción?

Ruby

- El equivalente al constructor es un método especial llamado `initialize`
- Es un método de instancia privado que es llamado automáticamente por el método de clase `new`
- Se ocupa de la **creación e inicialización de atributos de instancia**
 - ▶ Cualquier método de instancia puede crear atributos de instancia
 - ▶ Lo recomendable es limitar esta labor al método `initialize`
- **No se puede sobrecargar `initialize` (ni ningún otro método)**
 - ▶ Entonces, **¿se pueden tener varios constructores?** Opciones:
 - ★ Creando métodos de clase que cumplan el cometido de los constructores (igual que `new`)
 - ★ Haciendo que `initialize` admita un número variable de parámetros

Ejemplos

Ruby: Ejemplo con un constructor

```
1 class RestrictedPoint3D
2
3   # Atributos de clase
4   @@LIMIT_MAX = 100
5   @@LIMIT_MIN = 0
6
7   private
8   def restric_to_range (a) # método de instancia
9     result = [@@LIMIT_MIN, a].max
10    result = [@@LIMIT_MAX, result].min
11    result
12  end
13
14  def initialize (x, y, z) # creación e inicialización de atributos de instancia
15    @x = restric_to_range (x)
16    @y = restric_to_range (y)
17    @z = restric_to_range (z)
18  end
19 end
20
21 puts RestrictedPoint3D.new(-1,1,1).inspect
```

★ ¿Hay algún conflicto de nombres en las líneas 15, 16, ó 17?

Hay que mirar x,y

Ejemplos

Ruby: Ejemplo con dos constructores

```
1 class RestrictedPoint3D
2
3   # Añadimos al código anterior
4
5   def self.new_3D(x,y,z)      # método de clase
6     new(x,y,z)
7   end
8
9   def self.new_2D(x,y)        # método de clase
10    new(x,y,0)
11  end
12
13 private_class_method :new # pasa a ser privado
14 end
15
16 puts RestrictedPoint3D.new_3D(-1,101,-2000).inspect
17 puts RestrictedPoint3D.new_2D(1,99).inspect
18 puts RestrictedPoint3D.new_3D(50,51,52).inspect
19 puts RestrictedPoint3D.new_3D(-2000,50,2000).inspect
20 # puts RestrictedPoint3D.new(-1,1,1).inspect # Error, new es ahora privado
```

¡Mal ejemplo!

Ruby: Error frecuentemente cometido por los estudiantes

```
1 class RestrictedPoint3D
2
3   # Forma ERRÓNEA de implementar estos constructores
4
5   def self.new_3D(x,y,z)  # método de clase
6     @x = restric_to_range (x)
7     @y = restric_to_range (y)
8     @z = restric_to_range (z)
9   end
10
11  def self.new_2D(x,y)      # método de clase
12    @x = restric_to_range (x)
13    @y = restric_to_range (y)
14    @z = 0
15  end
16
17  private_class_method :new # pasa a ser privado
18 end
```

★ ¿Cuáles son los errores? ¿Por qué son errores?

Estos errores, en los exámenes, serán **penalizados**

Errores en el código de Ruby

1. **Métodos de clase (`self.new_3D` y `self.new_2D`) no crean instancias de la clase:** En Ruby, los métodos `self.new_3D` y `self.new_2D` en la clase `RestrictedPoint3D` están definidos como métodos de clase y no están devolviendo instancias de la clase `RestrictedPoint3D`. En su lugar, simplemente asignan valores a variables de instancia de clase, lo que no es el propósito de los constructores.
2. **Uso incorrecto del método privado `new`:** El método `new` ha sido marcado como privado, lo que significa que no puede ser llamado directamente para crear nuevas instancias de la clase. En Ruby, el método `new` es el constructor por defecto y debe mantenerse público o debe ser invocado de manera correcta desde métodos de clase como `self.new_3D` y `self.new_2D`.

Solución correcta

Para corregir estos errores, podemos modificar los métodos de clase para que creen y devuelvan nuevas instancias de `RestrictedPoint3D` utilizando el método `new`:

```
class RestrictedPoint3D
  attr_accessor :x, :y, :z

  def initialize(x, y, z)
    @x = restrict_to_range(x)
    @y = restrict_to_range(y)
    @z = restrict_to_range(z)
  end

  def self.new_3D(x, y, z)
    new(x, y, z)
  end

  def self.new_2D(x, y)
    new(x, y, 0)
  end

  private

  def restrict_to_range(value)
    # Implementar lógica para restringir el valor a un rango específico
  end
end
```

En este código:

- El método `initialize` es el constructor de la clase y se asegura de que las variables de instancia `@x`, `@y` y `@z` sean asignadas correctamente. - Los métodos `self.new_3D` y `self.new_2D` llaman al método `new` para crear y devolver nuevas instancias de la clase `RestrictedPoint3D`. - Se utiliza el método privado `restrict_to_range` para restringir los valores a un rango específico.

Ejemplos

Ruby: initialize con un número variable de parámetros

```
1 def initialize (x, y, *z)
2   # *z es un array con el resto de parámetros que se pasen
3
4   @x = restric_to_range (x)
5   @y = restric_to_range (y)
6   if (z.size != 0) then
7     z_param = z[0]
8   else
9     z_param = 0
10  end
11  @z = restric_to_range (z_param)
12 end
13
14 # En algún lugar fuera de la clase ...
15
16 puts RestrictedPoint3D.new(1,2,3,4,5,6).inspect
17
18 # los parámetros extra son ignorados
```

Ejemplos

Ruby: initialize con valores por defecto

```
1 def initialize (x, y, z=0)
2   # el parámetro z tiene un valor por defecto
3
4   @x=restric_to_range(x)
5   @y=restric_to_range(y)
6   @z=restric_to_range(z)
7 end
8
9 # En algún lugar fuera de la clase ...
10
11 puts RestrictedPoint3D.new(1,2).inspect
12 puts RestrictedPoint3D.new(1,2,3).inspect
```

desvelar los
datos

Ejemplos

Ruby: Parámetros nombrados con valores por defecto

```
1 # Parámetros nombrados con valores por defecto
2
3 def initialize (x:, y:, z:0)
4   @x = restric_to_range (x)
5   @y = restric_to_range (y)
6   @z = restric_to_range (z)
7 end
8
9 # En algún lugar fuera de la clase ...
10
11 puts RestrictedPoint3D.new(x:-1, y:101, z:-2000).inspect
12
13 # Puedo cambiar el orden
14 puts RestrictedPoint3D.new(y:2, z:3, x:1).inspect
15
16 puts RestrictedPoint3D.new(x:1, y:99).inspect
```

Código en Ruby

```
# Par metros nombrados con valores por defecto
def initialize (x:, y:, z:0)
  @x = restrict_to_range(x)
  @y = restrict_to_range(y)
  @z = restrict_to_range(z)
end

# En algún lugar fuera de la clase ...
puts RestrictedPoint3D.new(x:-1, y:101, z:-2000).inspect

# Puedo cambiar el orden
puts RestrictedPoint3D.new(x:2, z:3, x:1).inspect
puts RestrictedPoint3D.new(x:1, y:99).inspect
```

Código Corregido en Ruby

```
class RestrictedPoint3D
  def initialize(x:, y:, z: 0)
    @x = restrict_to_range(x)
    @y = restrict_to_range(y)
    @z = restrict_to_range(z)
  end

  def restrict_to_range(value)
    [[value, 0].max, 100].min
  end

  def inspect
    "#<RestrictedPoint3D: @x=#{@x}, @y=#{@y}, @z=#{@z}>"
  end
end

puts RestrictedPoint3D.new(x: -1, y: 101, z: -2000).inspect
puts RestrictedPoint3D.new(x: 2, y: 3, z: 3).inspect
puts RestrictedPoint3D.new(x: 1, y: 99).inspect
```

Salida del Código

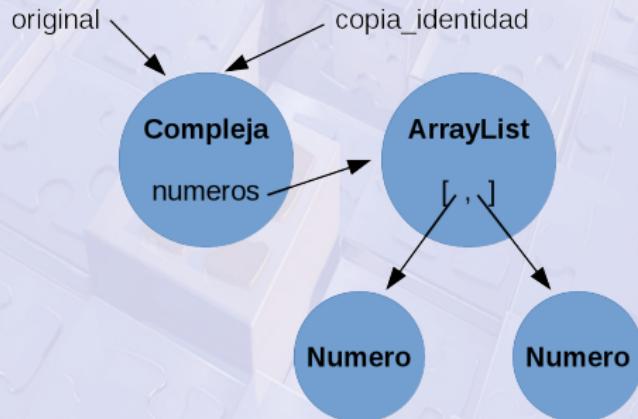
```
#<RestrictedPoint3D: @x=0, @y=100, @z=0>
#<RestrictedPoint3D: @x=2, @y=3, @z=3>
#<RestrictedPoint3D: @x=1, @y=99, @z=0>
```

En este código corregido:

- El método `initialize` es el constructor de la clase y asegura que las variables de instancia `@x`, `@y` y `@z` sean asignadas correctamente.
- El método `restrict_to_range` restringe los valores para que estén en el rango de 0 a 100.
- El método `inspect` proporciona una representación legible para humanos del objeto.

Constructor de copia

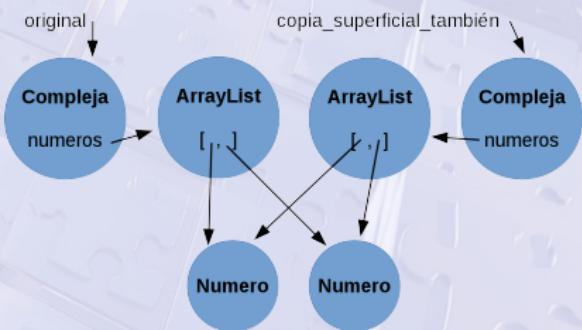
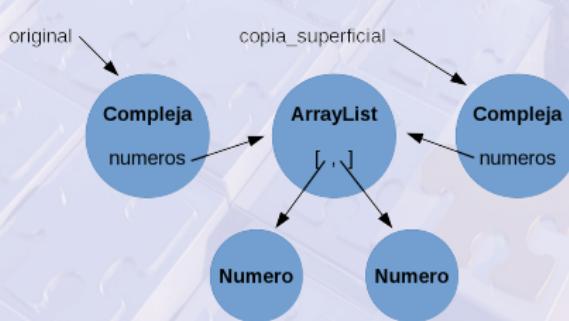
- Un constructor de copia recibe un objeto como parámetro
 - ▶ Construye otro objeto (distinta identidad)
 - ▶ Con el mismo estado (inicialmente) que el objeto recibido
- Puede haber diferentes niveles de copia (superficial y profunda)
 - ▶ Ya conocemos cómo actúa la asignación `copia = original`



Constructor de copia

Copia superficial

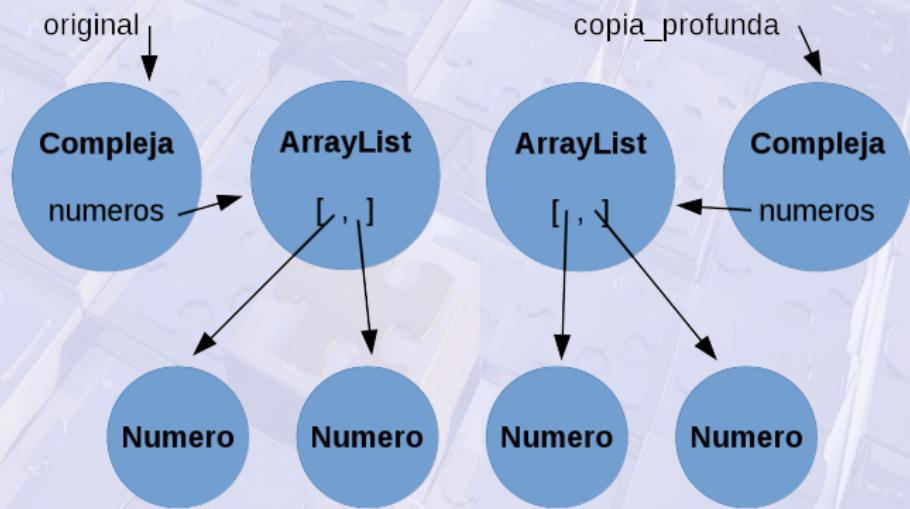
- En una copia superficial, aunque se crea un objeto distinto, en algún nivel se referencia un mismo objeto



Constructor de copia

Copia profunda

- En una copia profunda, se copia toda la jerarquía



Constructor de copia en Java

Java: Constructor de copia (superficial)

```
1 class Persona {  
2     private String nombre;  
3     // otros atributos  
4  
5     public Persona (String unNombre) {  
6         nombre = unNombre;  
7         // se inicializan el resto de atributos  
8     }  
9  
10    public Persona (Persona otraPersona) {  
11        nombre = otraPersona.nombre;  
12        // se asignan el resto de atributos  
13        // tomando los valores de los atributos del parámetro  
14    }  
15 }
```

Constructor de copia en Ruby

Ruby: Constructor de copia (superficial)

```
1 class Persona
2   attr_reader :nombre
3
4   # A los atributos que no tengan consultor básico hay que añadírselo
5
6   attr_reader :otro_atributo
7
8   def initialize (nombre, otro_atributo)
9     @nombre = nombre
10    @otro_atributo = otro_atributo
11  end
12
13  def self.constructor_copia (persona)
14    new(persona.nombre, persona.otro_atributo)
15  end
16 end
```

Memoria dinámica y pila

- En Java y Ruby todos los objetos **se crean** en memoria dinámica (*heap*)
- En ambos lenguajes las variables contienen referencias a objetos (punteros)
 - ▶ Hay algunas excepciones como los tipos primitivos de Java (*int, float, etc.*)
 - ▶ Los *String* también tienen un **tratamiento distinto**
- Cuando se devuelve el valor de una variable, **se está devolviendo una referencia a un objeto**
- ¿**Cómo se libera** la memoria?
 - ▶ Java y Ruby disponen de un **recolector de basura** que libera automáticamente la memoria utilizada por objetos no referenciados

Memoria dinámica y pila: El lenguaje C++

- En C++, el programador puede decidir si crea los objetos en la pila o en el *heap*
- También es el responsable de la liberación de la memoria reservada en el *heap* para un objeto

C++: Destructor

```

1 class A {
2 };
3
4 class B {
5     private:
6     A *atributo;
7
8     public:
9     B() {
10         atributo = new A();
11     }
12
13     ~B() {
14         // destructor
15         delete (atributo);
16     }
17 }
```

C++: Pila y Heap

```

1 void unaFuncion () {
2     A a;           // En la pila
3     B *b = new B(); // En el heap
4
5     . . .
6
7     delete (b);   // se libera del heap
8     // al salir se libera la pila
9 }
10
11 int main() {
12     unaFuncion();
13 }
```



- Los constructores no cuestan dinero
- El tiempo perdido entendiendo un código enrevesado, sí
- Con los constructores, y en general, con cualquier método,
 - ▶ Sobrecargarlos (si el lenguaje lo permite) de manera que cada constructor/método haga una cosa muy concreta
 - ▶ Si el lenguaje no admite sobrecarga, añadir constructores/métodos con distintos nombres
 - ▶ Si en un constructor/método, se debe hacer un procesamiento diferente según el número y tipo de los parámetros recibidos, tal vez haya que sobrecargarlo (o crear más constructores/métodos)
- Los diseños e implementaciones simples son fáciles de mantener



Construcción de objetos

Prof. Francisco Velasco Anguita

Dpto. Lenguajes y Sistemas Informáticos
Universidad de Granada

Programación y Diseño Orientado a Objetos

Doble Grado en Ingeniería Informática
y Administración y Dirección de Empresas
(Curso 2024-2025)

Consultores y Modificadores

Prof. Francisco Velasco Anguita

Dpto. Lenguajes y Sistemas Informáticos
Universidad de Granada

Programación y Diseño Orientado a Objetos

Doble Grado en Ingeniería Informática
y Administración y Dirección de Empresas
(Curso 2024-2025)

Créditos

- Las siguientes imágenes e ilustraciones son libres y se han obtenido de:
 - ▶ Emojis, <https://pixabay.com/images/id-2074153/>
- El resto de imágenes e ilustraciones son de creación propia, al igual que los ejemplos de código

Objetivos

- Saber crear y usar consultores y modificadores, tanto en Java como en Ruby
- Ser conscientes de la problemática de devolver o asignar referencias a objetos

Contenidos

- 1 Consultores
- 2 Modificadores
- 3 Ejemplos
 - Java
 - Ruby
- 4 Problemática de devolver (o asignar) referencias

Consultores

- Métodos encargados de devolver el valor de un atributo
 - No tienen necesariamente que limitarse a devolver ese valor.
Pueden devolverlo modificado, o una copia del mismo, etc.
 - Pueden ser de clase o de instancia
 - Habitualmente se nombran: `getAtributo()` en Java
 - Habitualmente se nombran: `atributo` en Ruby
 - Solo deben crearse los consultores que realmente sean necesarios
 - ▶ Se expone el estado interno al exterior
- ★ ¿Se pueden usar dentro de los constructores?

Modificadores

- Métodos encargados de **modificar el valor de un atributo**
 - No tienen necesariamente que limitarse a fijar ese valor.
Pueden y deben controlar las restricciones sobre ese atributo
 - Pueden ser de clase o de instancia
 - Habitualmente se nombran: **setAtributo(...)** en Java
 - Habitualmente se nombran: **atributo=** en Ruby
 - Solo deben crearse los modificadores que realmente sean necesarios
 - ▶ **Se expone el estado interno al exterior**
- ★ ¿Se pueden usar dentro de los constructores?

Ejemplos

Java: Consultores y modificadores

```
1 public class Persona {  
2  
3     private static final int MAYORIAEDAD = 18; // Atributo de clase  
4     private LocalDateTime fechaNacimiento; // Atributo de instancia  
5  
6     Persona (LocalDateTime fecha) {  
7         fechaNacimiento = fecha;  
8     }  
9  
10    public static int getMayoriaEdad() {  
11        return MAYORIAEDAD;  
12    }  
13  
14    public LocalDateTime getFechaNacimiento() {  
15        // Se devuelve al exterior una referencia a la fecha de nacimiento  
16        // Podría ser modificada desde fuera  
17        return fechaNacimiento;  
18    }  
19  
20    public void setFechaNacimiento(LocalDateTime fecha) {  
21        // Añadir comprobaciones relativas a las restricciones sobre la edad  
22        // Se está asignando una referencia a un objeto que ya está siendo referenciado  
23        // desde fuera de la clase  
24        fechaNacimiento = fecha;  
25    }  
26 }
```

Ejemplos

Java: Usando la clase anterior

```
1 Persona p=new Persona(LocalDateTime.of(2000,7,5,0,0));
2
3 // utilizamos el modificador
4 p.setFechaNacimiento(LocalDateTime.of(1950,7,5,0,0));
5
6 // utilizamos el consultor
7 System.out.println(p.getFechaNacimiento());
8
9 // utilizamos el consultor de clase
10 System.out.println(Persona.getMayoriaEdad());
```

Ejemplos

Ruby: Consultores y modificadores

```
1 require 'date'
2
3 class Persona
4
5   @@MAYORIA_EDAD = 18 # Atributo de clase
6
7   def initialize (fecha)
8     @fecha_nacimiento = fecha
9   end
10
11
12  def fecha_nacimiento      # consultor
13    # Se devuelve al exterior del objeto una referencia a la fecha
14    @fecha_nacimiento
15  end
16
17  def fecha_nacimiento=fecha  # modificador
18  # ¿ Restricciones ?
19  # Se asigna una referencia a un objeto que ya es referenciado desde fuera
20  @fecha_nacimiento = fecha
21 end
22
23  def self.MAYORIA_EDAD=e # modificador de clase
24  @@MAYORIA_EDAD = e
25 end
26 end
```

Ejemplos

Ruby: Usando la clase anterior

```
1 p=Persona.new(Date.new(2000,7,3))
2
3 # utilizamos el modificador
4 p.fecha_nacimiento=Date.new(2000,8,3)
5
6 # utilizamos el consultor
7 puts p.fecha_nacimiento
8
9 # utilizamos el modificador de clase
10 Persona.MAYORIA_EDAD=21
```

Ejemplos

Ruby: Consultores y modificadores implícitos

```
1 class UnaClase
2
3   def initialize (un, dos, tres)
4     @atr1 = un
5     @atr2 = dos
6     @atr3 = tres
7   end
8
9   attr_reader :atr1
10  attr_accessor :atr2
11  attr_writer :atr3
12
13 end
14
15 obj = UnaClase.new(1,2,3)
16 obj.atr2 = 8
17 puts obj.inspect
18 obj.atr2 = 9
19 puts obj.inspect
20 obj.atr3 = 7
21 puts obj.inspect
22 puts obj.atr1
23 puts obj.atr2
24 #puts obj.atr3 # no existe consultor
25 #obj.atr1 = 23 # no existe modificador
```

Ejemplos

Ruby: Consultores y modificadores implícitos

```
1 require 'date'
2
3 class Persona
4
5   @@MAYORIA_EDAD = 18 # Atributo de clase
6
7   def initialize (fecha)
8     @fecha_nacimiento = fecha
9   end
10
11 attr_accessor :fecha_nacimiento # consultor + modificador
12
13 def self.MAYORIA_EDAD=e    # modificador de clase
14   @@MAYORIA_EDAD = e
15 end
16 end
```

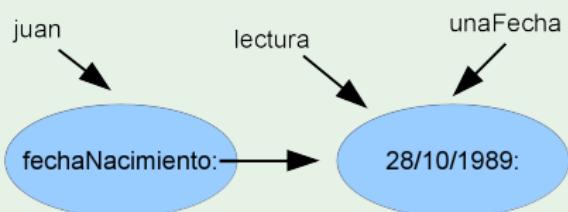
Problemática de devolver (o asignar) referencias

Java: Asignación y devolución de referencias

```

1 class Persona {
2     private GregorianCalendar fechaNacimiento;
3
4     Persona (GregorianCalendar nace) {
5         fechaNacimiento = nace;
6     }
7
8     GregorianCalendar getFechaNacimiento () {
9         return fechaNacimiento;
10    }
11
12 // ...
13 }
14
15 GregorianCalendar unaFecha = new GregorianCalendar (1989,10,28);
16
17 Persona juan = new Persona (unaFecha);
18 System.out.println(juan.toString()); // Nací el 28/10/1989
19
20 GregorianCalendar lectura = juan.getFechaNacimiento();
21 lectura.set (1985,5,13);
22 System.out.println(juan.toString()); // Nací el 13/5/1985
23 unaFecha.set (2001,1,1);
24 System.out.println(juan.toString()); // Nací el 1/1/2001

```



★ ¿Soluciones?

Consultores y Modificadores

→ **Diseño** ←

- Crear solo los que sean realmente necesarios
- Tener en cuenta si se devuelven (o se asignan) referencias
 - ▶ En lenguajes como Java o Ruby todas las variables son referencias (punteros)
 - ★ Salvo los tipos primitivos: int, float, ...
 - Los String tampoco deben preocuparnos
- No hay una regla a aplicar en todos los casos
 - ▶ A veces interesa devolver (o asignar) una referencia
 - ▶ Otras veces interesa devolver (o asignar) una copia
 - ▶ Puede depender de *a quién* se le dé (o *de dónde*) venga
- Hay que decidirlo evaluando cada situación



Java: Asignación de referencias

```
1 objeto = new Clase();
2 otroObjeto = objeto;
3 // Un ÚNICO objeto con dos referencias
```

Ruby: Asignación de referencias

```
1 objeto = Clase.new;
2 otroObjeto = objeto;
3 # Un ÚNICO objeto con dos referencias
```

Consultores y Modificadores

Prof. Francisco Velasco Anguita

Dpto. Lenguajes y Sistemas Informáticos
Universidad de Granada

Programación y Diseño Orientado a Objetos

Doble Grado en Ingeniería Informática
y Administración y Dirección de Empresas
(Curso 2024-2025)

Elementos de Agrupación

Prof. Francisco Velasco Anguita

Dpto. Lenguajes y Sistemas Informáticos
Universidad de Granada

Programación y Diseño Orientado a Objetos

Doble Grado en Ingeniería Informática
y Administración y Dirección de Empresas
(Curso 2024-2025)

Créditos

- Las siguientes imágenes e ilustraciones son libres y se han obtenido de:
 - ▶ Emojis, <https://pixabay.com/images/id-2074153/>

 - ▶ <https://pixabay.com/images/id-4541278/>
- El resto de imágenes e ilustraciones son de creación propia, al igual que los ejemplos de código

Objetivos

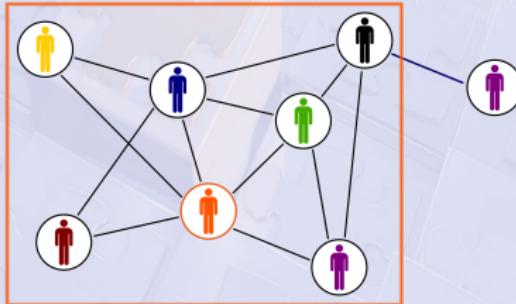
- Saber crear y usar paquetes en Java
- Saber crear y usar módulos en Ruby
- Gestionar correctamente las líneas `require_relative` en Ruby

Contenidos

- 1 Introducción
- 2 Paquetes Java
- 3 Módulos Ruby
- 4 Un proyecto Ruby definido en varios archivos

Introducción

- En el mundo real hay entidades que cooperan para algún fin
 - ▶ **Ejemplo:** En una empresa sus trabajadores tienen distintas responsabilidades pero trabajan para un mismo fin
- Esa circunstancia se puede ver reflejada en los lenguajes de programación implementando algún tipo de agrupación de clases
 - ▶ Puede haber clases que solo se relacionan con clases del grupo
 - ▶ Otras clases también se relacionan con el *exterior*
 - ▶ Las **propiedades** añadidas a una clase por pertenecer a un grupo **pueden ser diferentes** según el lenguaje concreto



Paquetes Java

- Permiten agrupar clases
- Constituyen un espacio de nombres
 - ▶ Es posible tener varias clases que se llamen igual, pero en paquetes distintos
- Existe una visibilidad (de paquete) que otros lenguajes no tienen
- Uso:
 - ▶ Para indicar que los elementos definidos en un archivo pertenecen a un paquete, en dicho archivo se añadirá el nombre del paquete (comenzando con minúscula)
 - ▶ Para usar elementos de un paquete distinto al actual, hay que indicarlo
 - ▶ En disco, un paquete aparece como una carpeta del sistema de ficheros
Los archivos de los elementos que pertenecen a un paquete estarán en su carpeta

Ejemplo: Paquetes Java

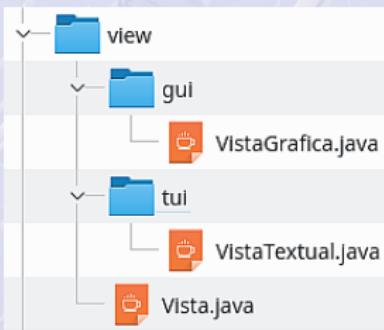
```
1 package miPrograma;  
2 // Los elementos que se definan en este archivo pertenecerán al paquete miPrograma  
3 import modelo.Fachada; // En este fichero se va a usar la clase Fachada del paquete modelo
```

Paquetes Java

- Cada paquete Java es independiente del resto
aunque a nivel de nombrado (y de almacenamiento en disco) parezca que uno es subpaquete de otro
 - En Java NO existen subpaquetes

Paquete: view

```
1 package view;
2 interface Vista {
3     ...
4 }
```



Paquete: view.gui

```
1 package view.gui;
2 import view.Vista;
3 class VistaGrafica
4     implements Vista {
5     ...
6 }
```

Paquete: view.tui

```
1 package view.tui;
2 import view.Vista;
3 class VistaTextual
4     implements Vista {
5     ...
6 }
```

← Estructura de carpetas y archivos

★ ¿Se pueden quitar las líneas import?

No, porque se usa Vista en las clases

Módulos Ruby

- Permiten agrupar una gran variedad de elementos: clases, constantes, funciones, otros módulos, etc.
- Constituyen un espacio de nombres
- Uso:
 - ▶ Para incluir un elemento en un módulo:
se abre el módulo, se realiza la definición, y se cierra el módulo
 - ▶ Para utilizar un elemento de un módulo distinto al actual
hay que anteponer <NombreModulo>::
 - ▶ Se puede copiar todo el contenido de un módulo dentro de una clase (`include`)
- En Ruby sí puede haber módulos dentro de módulos
 - ▶ Se accede a los símbolos encadenando nombres de módulos y ::
Ejemplo: `objeto = ModuloExterno::ModuloInterno::Clase.new`

Módulos Ruby

Ejemplo: Ruby

```
1 module Externo
2   class A
3   end
4
5   module Interno
6     class B
7     end
8   end
9 end
10
11 module Test
12   def test
13     puts "Testeando"
14   end
15 end
16
17 class C
18   include Test  # Literalmente, se copia el contenido del módulo Test
19 end
20
21 a = Externo::A.new
22 b = Externo::Interno::B.new
23 c = C.new
24 c.test
```

Un proyecto Ruby definido en varios archivos

- Normalmente, en cualquier lenguaje, cada clase que forma parte de un proyecto se define en un archivo distinto
 - ★ Buenas prácticas de programación
- En los lenguajes compilados, se procesan todos los archivos fuentes (se compilan) antes de ejecutar el programa principal
 - ★ ¿Habéis usado Makefile en C++?
 - ★ ¿Sabéis lo que es una tabla de símbolos?
- Ruby es interpretado, Ruby:
 - No sabe que un proyecto está formado por varios archivos
 - No realiza un procesamiento previo que identifique las clases
 - ▶ Si en un archivo mencionamos una clase definida en otro archivo, nos dará un error si no nos hemos preocupado nosotros de que procese las clases antes de usarlas

Referenciando archivos Ruby

- Cuando se ejecuta `ruby archivo_principal.rb` se va procesando este archivo línea a línea
- Si en este archivo se menciona la clase `A`, debemos haberle indicado a Ruby que previamente haya procesado el archivo que contiene la definición de la clase `A`
- Lo hacemos con las instrucciones:
 - ▶ `require` se suele usar para archivos del lenguaje
 - ▶ `require_relative` se suele usar para archivos propios
- Criterio:
 - ▶ Cuando en un archivo aparece el nombre de una clase, se añade un `require_relative` al archivo que define esa clase
- Ruby anota qué archivos ha cargado y no los carga dos veces

Ejemplo

: cosa.rb

```

1 class Cosa
2   @@Maximo = 3
3
4   attr_reader :nombre
5
6   def initialize (unNombre)
7     @nombre = unNombre
8   end
9
10  def self.Maximo
11    @@Maximo
12  end
13 end

```

: persona.rb

```

1 require_relative 'cosa' # por línea 4
2
3 class Persona
4   @@MaximoPermitido = Cosa.Maximo
5
6   def initialize (unNombre)
7     @nombre = unNombre
8     @cosas = []
9   end
10
11  def otraCosaMas (unaCosa)
12    if @cosas.size < @@MaximoPermitido
13      @cosas << unaCosa
14    end
15  end
16
17  def to_s
18    salida = "Me llamo #{@nombre} y
19              tengo :\n"
20    for unaCosa in @cosas do
21      salida += "- #{unaCosa.nombre}\n"
22    end
23    salida
24  end

```

: principal.rb

```

1 require_relative 'cosa' # por línea 4
2 require_relative 'persona' # por línea 5
3
4 mochila = Cosa.new("Mochila")
5 juan = Persona.new("Juan")
6 juan.otraCosaMas (mochila)
7 puts juan.to_s

```

Mal ejemplo

- Algunos estudiantes añaden `require_relative` de todos los archivos en todos los archivos
- Esa mala práctica, más pronto que tarde, produce errores



:cosa.rb

```
1 # Se añade un require_relative innecesario
2 # No se menciona la clase Persona en este archivo
3
4 require_relative 'persona'
5
6 class Cosa
7   # La clase se define igual que en el ejemplo anterior
8 end
9 # No cambia nada más en ningún otro archivo
```

Ejecución: Mensaje de error obtenido

```
persona.rb:4:in `<class:Persona>': uninitialized constant Persona::Cosa (NameError)
```

★ Trazémoslo y averigüemos el porqué

- ¿Cuándo debo agrupar clases en paquetes o módulos?
 - ▶ No hay una respuesta única
 - ▶ Normalmente se agrupan clases que tienen una relación entre ellas
 - ▶ En la asignatura *Fundamentos de Ingeniería del Software* profundizaréis más en cuestiones de diseño como esta

Elementos de Agrupación

Prof. Francisco Velasco Anguita

Dpto. Lenguajes y Sistemas Informáticos
Universidad de Granada

Programación y Diseño Orientado a Objetos

Doble Grado en Ingeniería Informática
y Administración y Dirección de Empresas
(Curso 2024-2025)

UML: Diagramas Estructurales

Prof. Francisco Velasco Anguita

Dpto. Lenguajes y Sistemas Informáticos
Universidad de Granada

Programación y Diseño Orientado a Objetos

Doble Grado en Ingeniería Informática
y Administración y Dirección de Empresas
(Curso 2024-2025)

Créditos I

- Las siguientes imágenes e ilustraciones son libres y se han obtenido de:

- ▶ Emojis, <https://pixabay.com/images/id-2074153/>



- ▶ <https://pixabay.com/images/id-1044090/>



- ▶ <https://pixabay.com/images/id-4129246/>



- ▶ <https://pixabay.com/images/id-3480187/>



- ▶ <https://pixabay.com/images/id-36561/>



- ▶ <https://www.uml.org>

Créditos II



▶ <https://pixabay.com/images/id-3846597/>

- El resto de imágenes e ilustraciones son de creación propia, al igual que los ejemplos de código

Objetivos

- Saber interpretar un diagrama de clases
 - ▶ Cada clase individualmente
 - ▶ Y las relaciones entre ellas
- Saber implementarlo
- Entender la semántica de un diagrama de clases
- Aprender diseño analizando los diagramas de clases que se os proporcionen

Contenidos

1 Introducción

- UML

2 Diagrama de clases

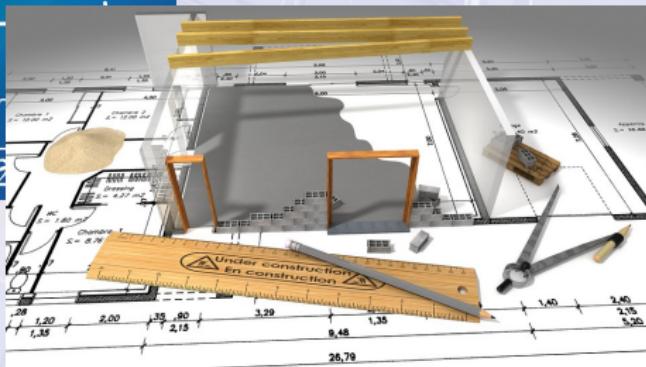
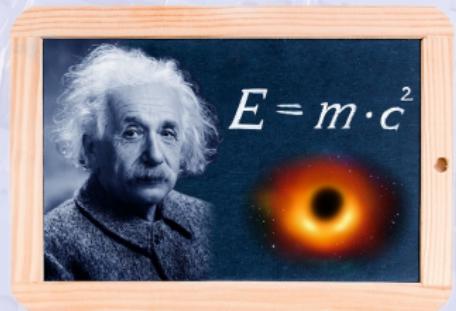
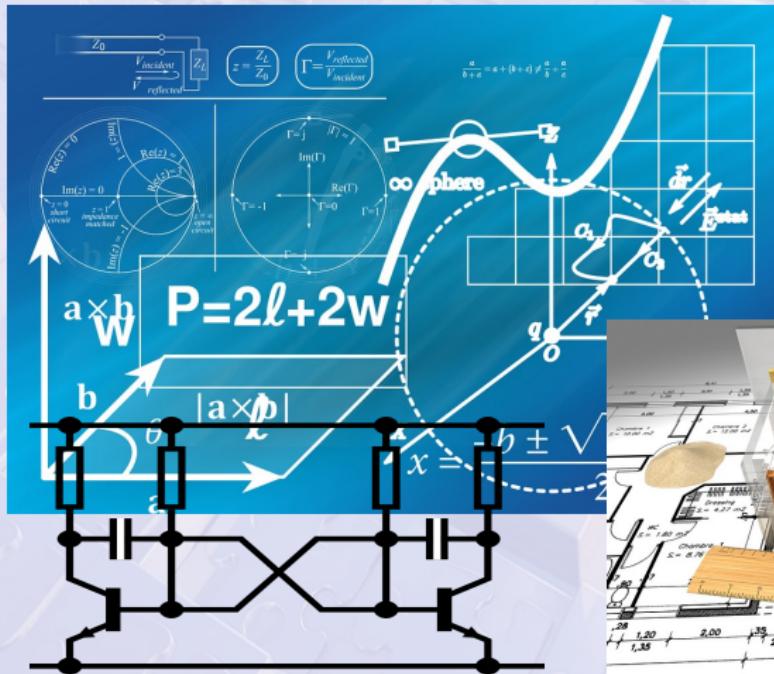
3 Relaciones entre clases

- Asociación
- Dependencia

4 Diagrama de paquetes

Introducción

- Muchas disciplinas usan lenguajes para expresarse eliminando en parte la ambigüedad del lenguaje natural



UML



- UML es un "**Lenguaje Unificado de Modelado**", un lenguaje de diseño y no una metodología
- Es **independiente del lenguaje de programación** con el que posteriormente se implemente el diseño
- Permite:
 - ▶ **Especificar** mediante modelos las características de un sistema antes de su construcción
 - ▶ **Visualizar** gráficamente un sistema software de forma que sea entendible por diversos desarrolladores
 - ▶ **Documentar** un sistema desarrollado para facilitar su mantenimiento, revisión y modificación
- Dispone de una amplia variedad de diagramas. Propósitos:
 - ▶ Modelar la estructura de un sistema, su comportamiento dinámico, los productos resultantes de un proyecto, etc.

Diagrama de clases

- Muestra las clases y sus relaciones

- **Tipos de relaciones:**

(en esta lección)

- ▶ **Asociación**
- ▶ **Dependencia**
- (cuando veamos herencia)
- ▶ **Generalización**
- ▶ **Realización**



Representación de una clase

ClaseEjemplo
-deClase : long
+publico : float = 100
#protegido : float
~paquete : OtraClase [1..*]
-privado : boolean
+metodoClase(a : int) : void
+deInstanciaPublico(a : float, b : int[]) : int
-deInstanciaPrivado()

Representación de una clase

Visual Paradigm Standard (verada/Universidad Granada)

Especificadores de acceso:

- + público
- ~ paquete
- # protegido
- privado

Se pueden indicar valores por defecto para los atributos

ProductoPrimeraNecesidad

```
+tasalVA : float = 4.0
-componentes : String[1..*]
#nombre : String
~perecedero : Boolean
-precioSinIVA : float
+precio() : float
+setTasaIVA(nuevaTasa : float)
```

Es posible indicar extremo inferior y superior en colecciones

Los atributos y métodos de clase se subrayan

Relaciones entre clases

● Asociación

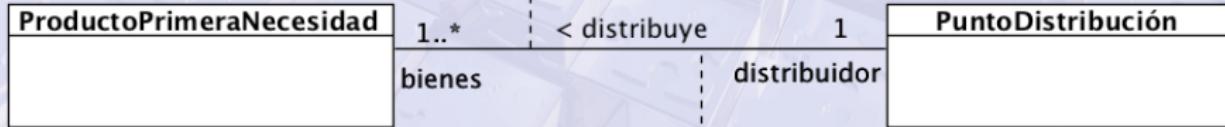


- ▶ Modela una **relación estructural fuerte y duradera en el tiempo**
- ▶ Las asociaciones **generan atributos de referencia**
- **Error MUY común:** No añadir atributos de referencia
- ▶ **Navegabilidad:**
 - ★ Se representa con puntas de flecha
 - ★ Indica si es posible *conocer* la/s instancia/s relacionadas con la instancia de origen
 - ★ Si no se indican flechas, por defecto las relaciones son **bidireccionales**
- ▶ **Cardinalidad / multiplicidad:**
 - ★ Se representa con **números** (pueden definir un rango)
 - ★ Indica cuántas instancias de la clase situada en un extremo están vinculadas a una instancia de la clase situada en el extremo opuesto
 - ★ Si no se indica nada, por defecto su valor es 1

Ejemplo de asociación

Visual Paradigm Standard/Zoraida Callejas/Universidad Granada)

Se puede dar nombre a la asociación e indicar el sentido en que debe leerse (no confundir con la navegabilidad)



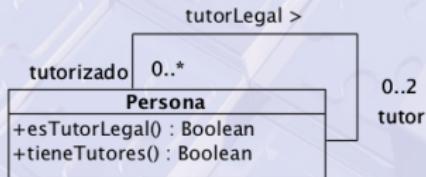
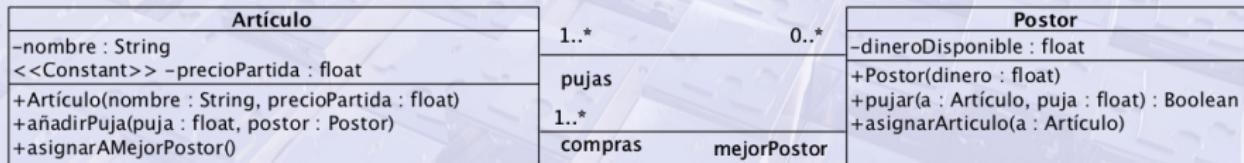
En cada extremo se puede indicar los nombres de los roles de las clases en la asociación

Multiplicidad: indica que un punto de distribución puede distribuir varios productos de primera necesidad y cada producto de primera necesidad sólo puede ser distribuido por un punto de distribución

Navegabilidad: La asociación es bidireccional, por lo tanto el punto de distribución puede conocer los productos que distribuye y cada producto conoce también cuál es su punto de distribución

Ejemplos de asociaciones

Visual Paradigm Standardizada (Universidad Granada)



Clases asociación

- Los vínculos entre las instancias pueden llevar información asociada
- Una asociación puede modelarse como una clase, cada enlace se convierte entonces en instancia de dicha clase

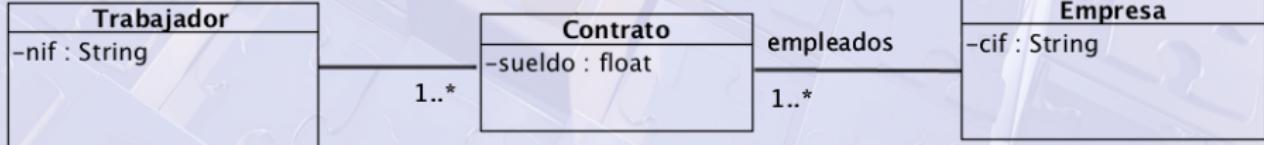
Visual Paradigm Standard (zoraida (Universidad Granada))



Clases asociación

- Ambos diagramas son equivalentes

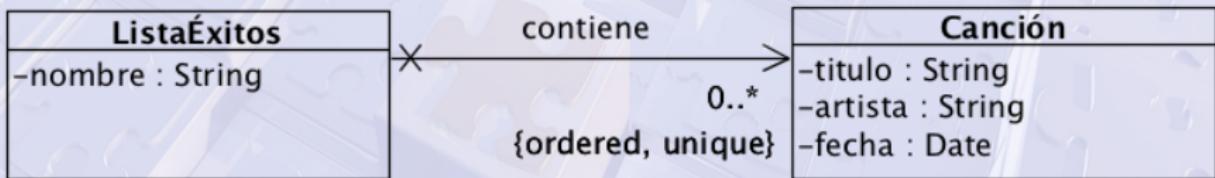
Visual Paradigm Standard (versión 10.0) (Universidad de Granada)



Asociación: Propiedades de los extremos

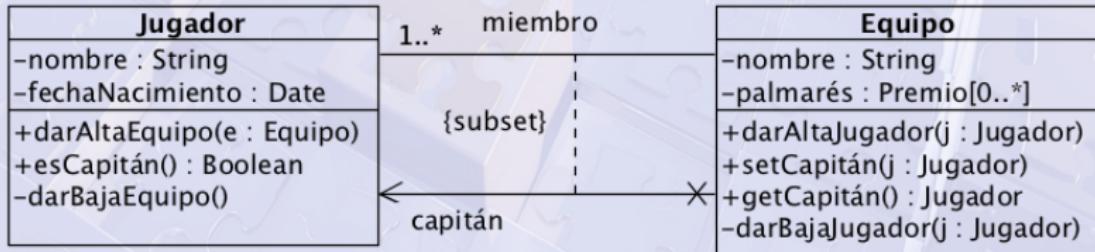
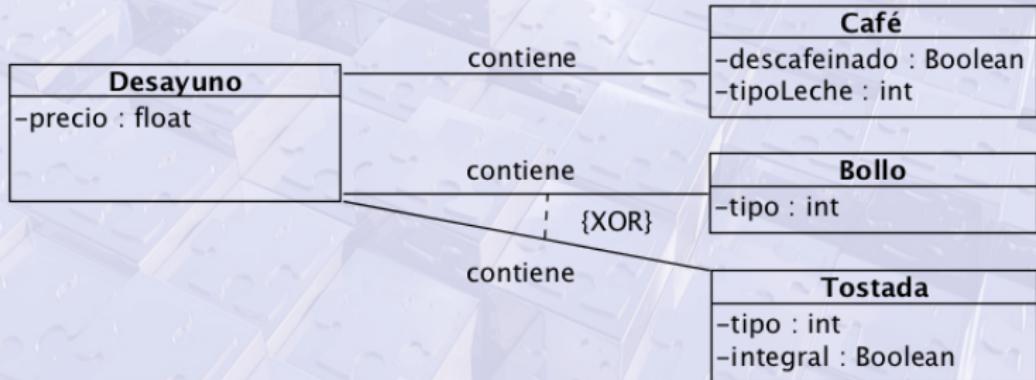
- En los extremos de las asociaciones se pueden indicar propiedades
- Las más comunes con multiplicidad mayor a 1 son:
 - {ordered} para indicar que se trata de una secuencia ordenada
 - {unique} para indicar que los elementos no se repiten

Visual Paradigm Standard(zoraida(Universidad Granada))



Especificaciones de las asociaciones

Visual Paradigm Standard (Zoraida (Universidad Granada))

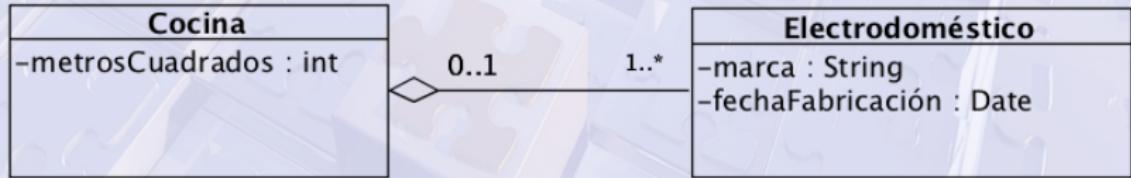


Asociaciones especiales

• Agregación



- ▶ Una de las clases representa el TODO y la otra las PARTES
- ▶ La cardinalidad en el TODO puede ser cualquiera
- ▶ Un objeto PARTE podría estar en varios TODO ...
- ▶ ... o en ninguno



Asociaciones especiales

● Composición



- ▶ Agregación fuerte donde las PARTES no tienen sentido sin el TODO
- ▶ La cardinalidad en el TODO debe ser 1
- ▶ Un objeto PARTE NO puede estar en varios TODO
- ▶ Tampoco puede estar en ningún TODO



Relaciones entre clases

● Dependencia



- ▶ Modela una relación débil y poco duradera en el tiempo
 - ▶ Cuando desde una clase se utilizan instancias de otra clase
 - ▶ Ejemplos
 - ★ Un método de una clase recibe como parámetros instancias de otra clase
 - ★ Un método de una clase devuelve una instancia de otra clase
 - ▶ Si se modifica la interfaz externa de una clase podrían verse afectadas todas las que dependen de ella
 - ▶ No genera atributos
- **Error común:** Añadir *atributos de dependencia*
- ▶ Dirección de la dependencia:
 - ★ Se representa con puntas de flecha
 - ★ Indica que una clase utiliza a la otra

Ejemplo de dependencia

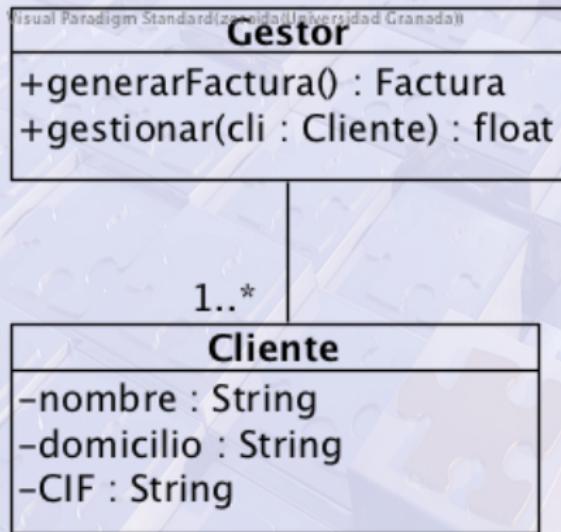


Diagrama de paquetes

- Permiten expresar relaciones de dependencia entre paquetes
- *Recordar:*
 - ▶ Los paquetes son agrupaciones
 - ▶ Pueden agrupar clases y otros paquetes
 - ★ En Java no existen los subpaquetes

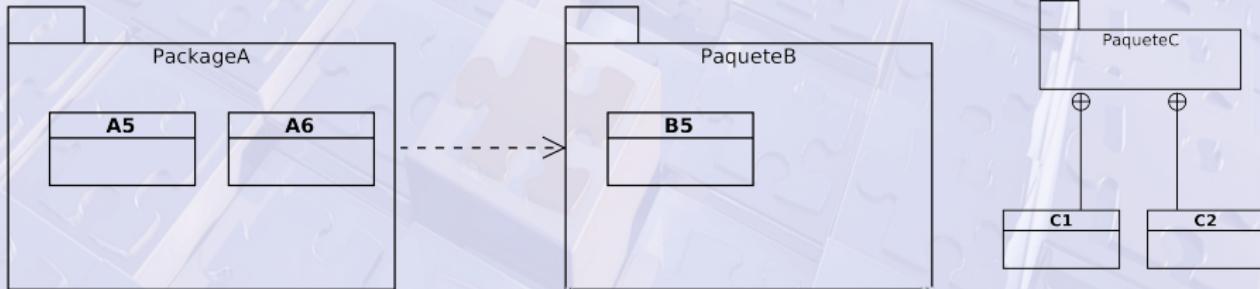


Diagrama de clases

→ **Diseño** ←

- Ya sabemos interpretar un diagrama de clases (DC)
- También sabemos implementarlo
- Pero, ¿cómo realizamos un DC para un problema concreto?
 - ▶ Para ello tenéis que:
 - ★ Entender bien el problema, los requerimientos que plantea
 - ★ Determinar qué clases (responsabilidad, atributos y métodos) van a modelar dicho problema
 - ★ Determinar cómo se relacionan unas clases con otras

Objetivo: Cumplir con los requerimientos planteados

- ▶ En definitiva, hay que realizar INGENIERÍA DEL SOFTWARE

★ Todo esto lo aprenderéis en la asignatura

Fundamentos de Ingeniería del Software

- No obstante, una vez entendido, sí deberíais ser capaces de modificar un DC ante pequeños cambios en el problema
- Cuando implementéis un DC (por ejemplo, en prácticas)
 - ▶ No os limitéis a la parte sintáctica (flechas, cajas, símbolos, etc.)
 - ▶ No os preocupéis solamente por *traducir* el DC a código
 - ▶ Entender el DC desde el punto de vista semántico
 - ★ Observar cómo el DC modela el problema

★ Aprender diseño analizando los DC que se os proporcionen

UML: Diagramas Estructurales

Prof. Francisco Velasco Anguita

Dpto. Lenguajes y Sistemas Informáticos
Universidad de Granada

Programación y Diseño Orientado a Objetos

Doble Grado en Ingeniería Informática
y Administración y Dirección de Empresas
(Curso 2024-2025)

UML: Diagramas de Interacción

Prof. Francisco Velasco Anguita

Dpto. Lenguajes y Sistemas Informáticos
Universidad de Granada

Programación y Diseño Orientado a Objetos

Doble Grado en Ingeniería Informática
y Administración y Dirección de Empresas
(Curso 2024-2025)

Créditos

- Las siguientes imágenes e ilustraciones son libres y se han obtenido de:
 - ▶ Emojis, <https://pixabay.com/images/id-2074153/>
- El resto de imágenes e ilustraciones son de creación propia, al igual que los ejemplos de código

Objetivos

- Saber interpretar los diagramas de secuencia y comunicación
- Saber implementarlos

Contenidos

1 Introducción

2 Diagramas de secuencia

3 Diagramas de comunicación

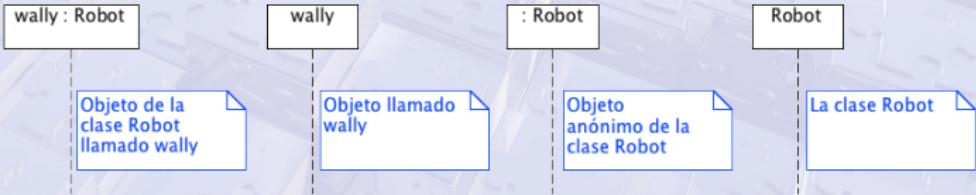
Diagramas de interacción

- Su propósito es mostrar el comportamiento del sistema a través de las interacciones entre los elementos del modelo
- Hay dos tipos básicos:
 - ▶ Diagramas de secuencia: Enfatizan la secuencia temporal de los mensajes enviados entre objetos
 - ▶ Diagramas de comunicación: Enfatizan la relación entre los objetos receptores y emisores de los mensajes
- Elementos:
 - ▶ Participantes: Objetos y clases que forman parte de la interacción
 - ▶ Mensajes: El flujo y su secuencia entre los participantes

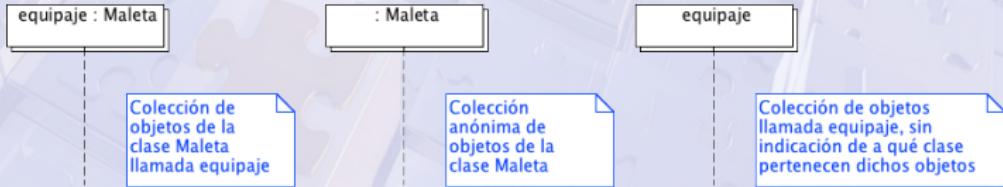
Diagramas de secuencia

- Los **participantes** se muestran en una caja

El nombre del objeto debe ir en minúscula y el de la clase en mayúscula. Presta atención a la localización de los dos puntos entre el nombre del objeto y el de la clase

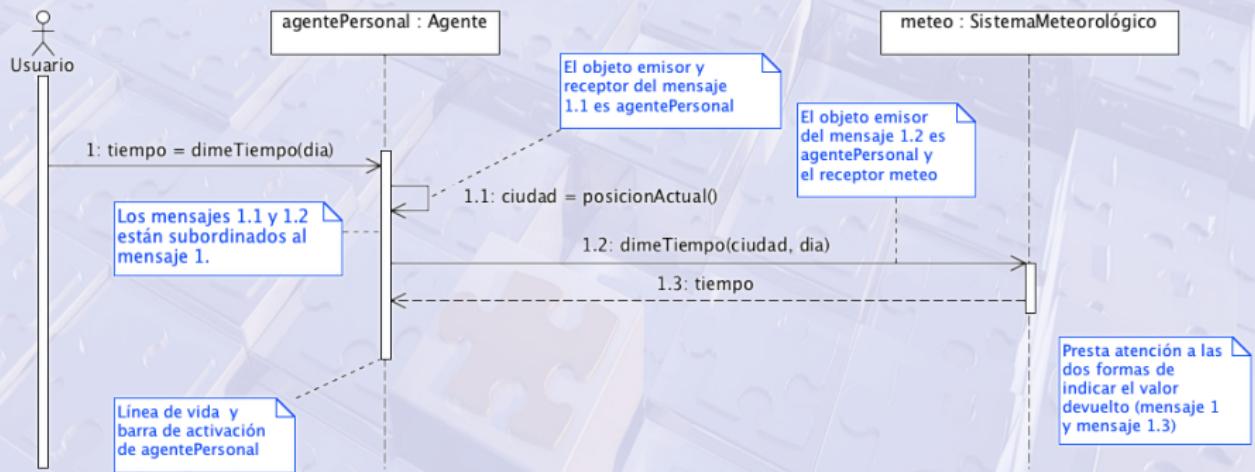


Los **multibjetos** o colecciones de objetos se representan con un doble fondo



Diagramas de secuencia

- **Mensajes: Emisor y Receptor**



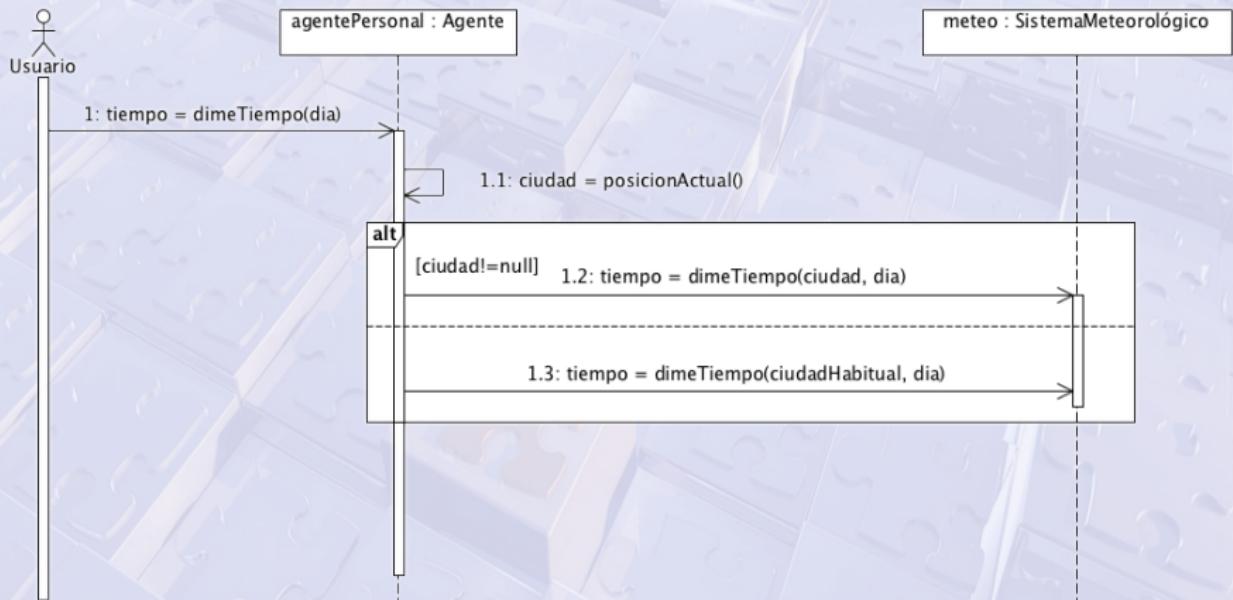
Diagramas de secuencia

Ruby: Implementación del diagrama anterior

```
1 class Agente
2
3 . .
4
5   def dimeTiempo (dia)
6     # No se indica receptor, es el propio objeto
7     ciudad = posicionActual
8
9     # ¿Cómo sabemos que meteo es un atributo?
10    @meteo.dimeTiempo (ciudad, dia)
11
12    # Devuelve el resultado del último paso de mensaje
13  end
14
15 . .
16
17 end
```

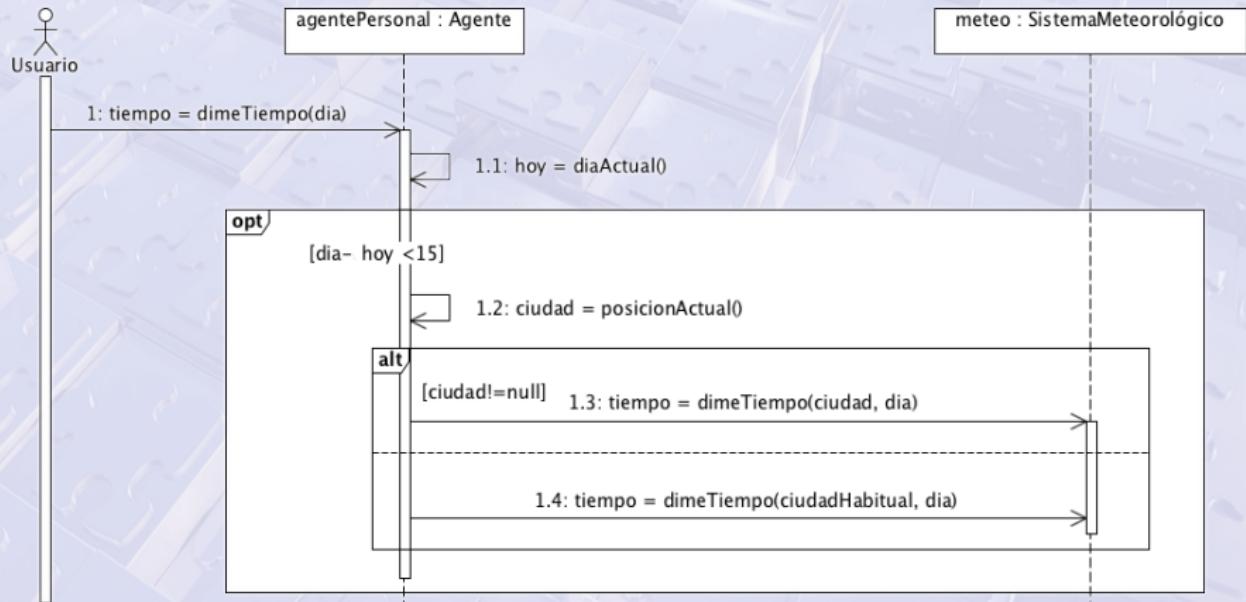
Diagramas de secuencia

- Fragmentos: **Condicionales**



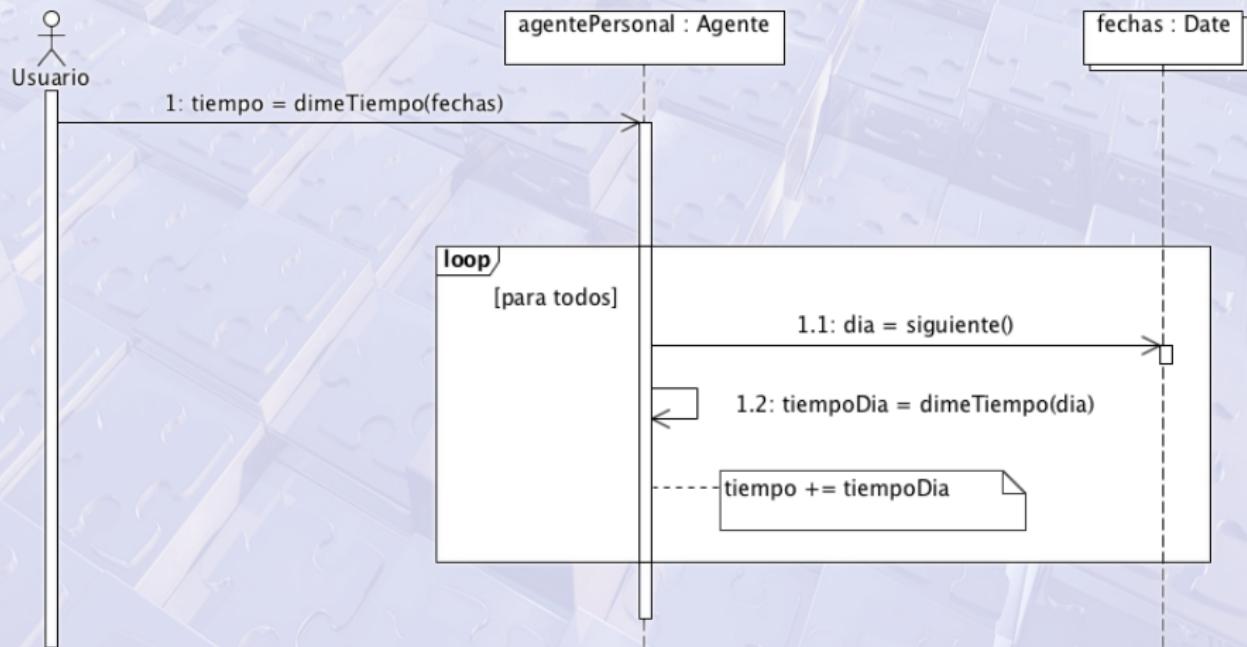
Diagramas de secuencia

- Fragmentos: Condicionales



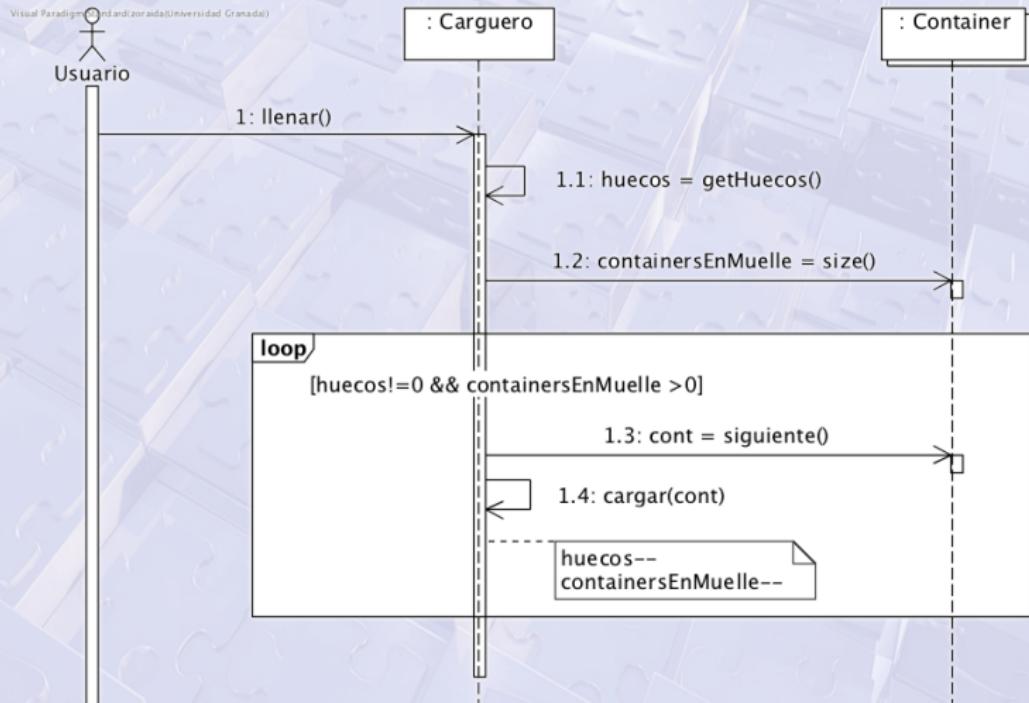
Diagramas de secuencia

• Fragmentos: Bucles



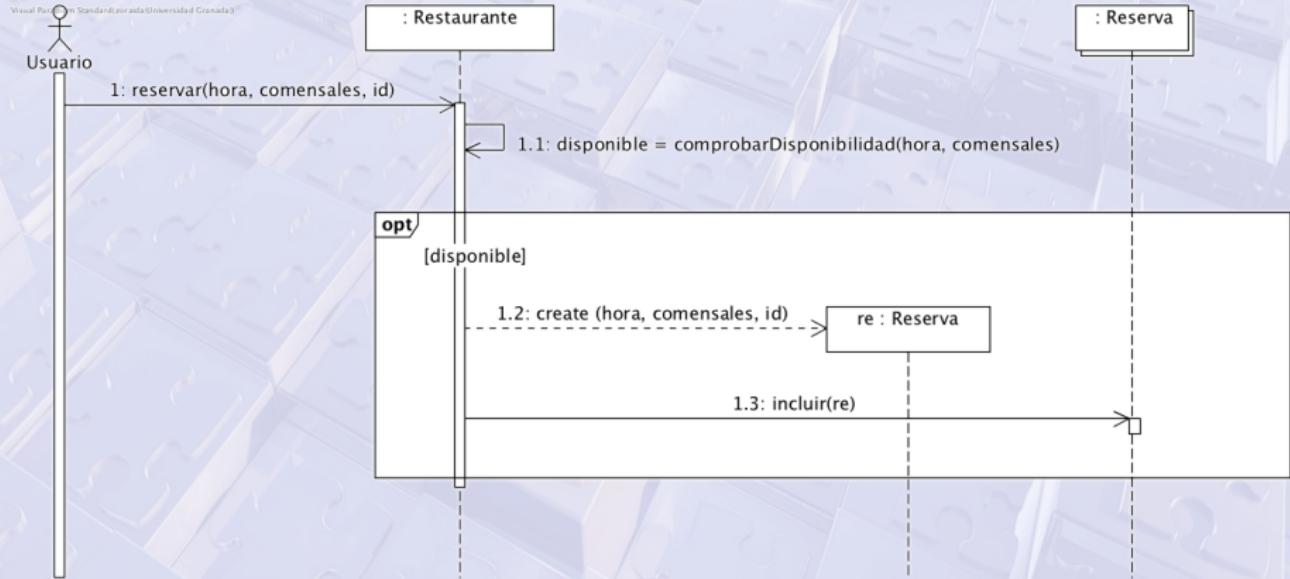
Diagramas de secuencia

- Fragmentos: Bucles



Diagramas de secuencia

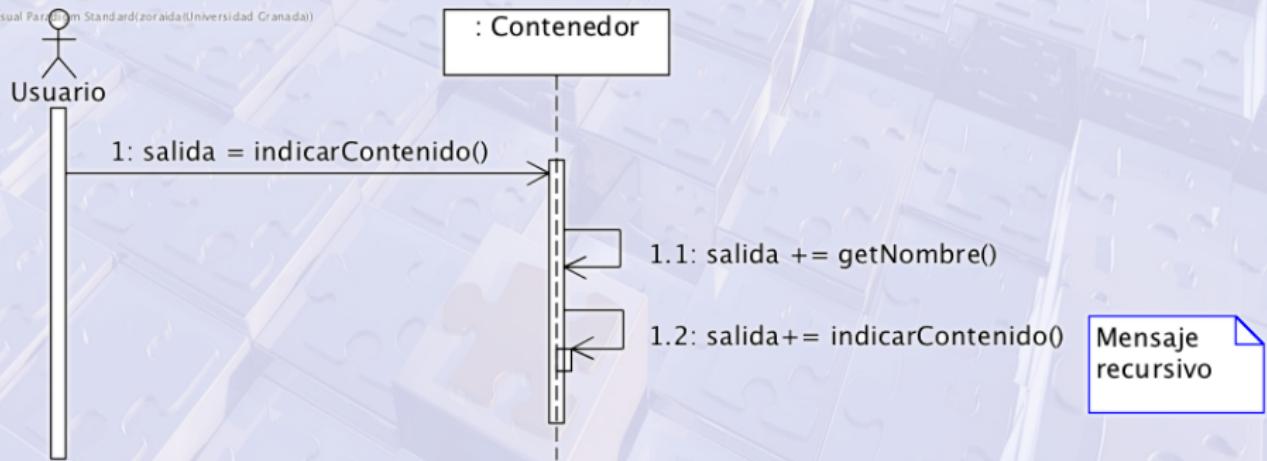
● Creación de instancias



Diagramas de secuencia

• Recursividad

Visual Paragon Standard (zoraida/Universidad_Granada)



Diagramas de comunicación

- Muestran de forma visual muy clara las vías de comunicación que deben darse entre los participantes para que pueda llevarse a cabo el envío de mensajes entre ellos
- Las vías de comunicación (enlaces) son el elemento principal y el orden temporal de los mensajes un elemento secundario

Diagramas de comunicación

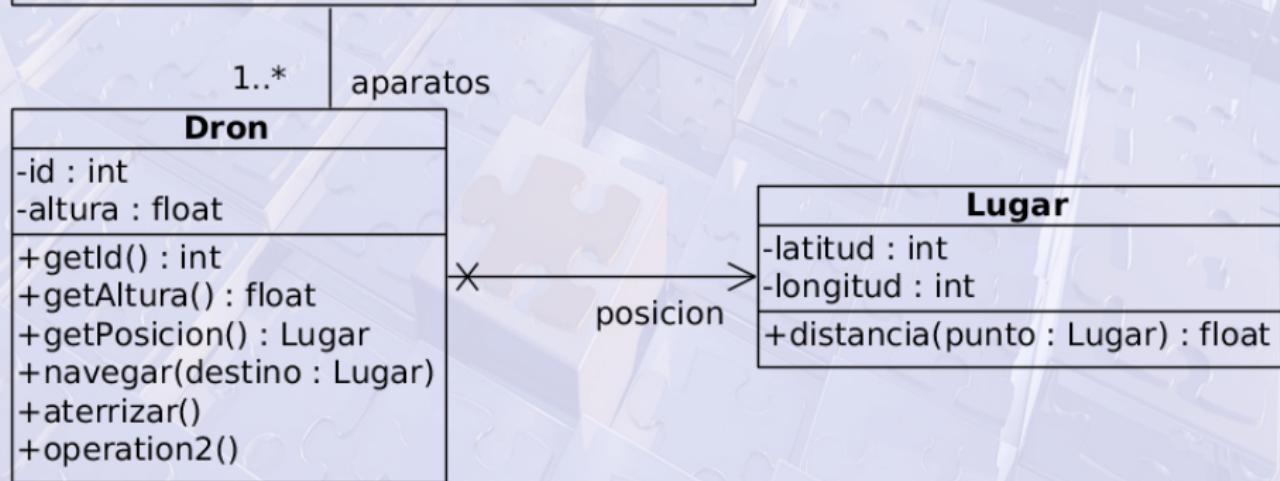
- Las vías de comunicación se representan mediante líneas que unen a los participantes
- Tipos de enlaces:
 - ▶ Global (G): Uno de los participantes pertenece a un ámbito superior. Ej: un atributo de clase
 - ▶ Asociación (A): Entre los participantes existe una asociación
 - ▶ Parámetro (P): Uno de los objetos es pasado como parámetro a un método del otro participante
 - ▶ Local (L): Uno de los participantes es un objeto local a un método del otro participante
 - ▶ Self (S): Un objeto también puede enviarse mensajes a sí mismo

DC para los ejemplos siguientes

UML Paradigm Standard (Miguel Lástra (Universidad de Granada))

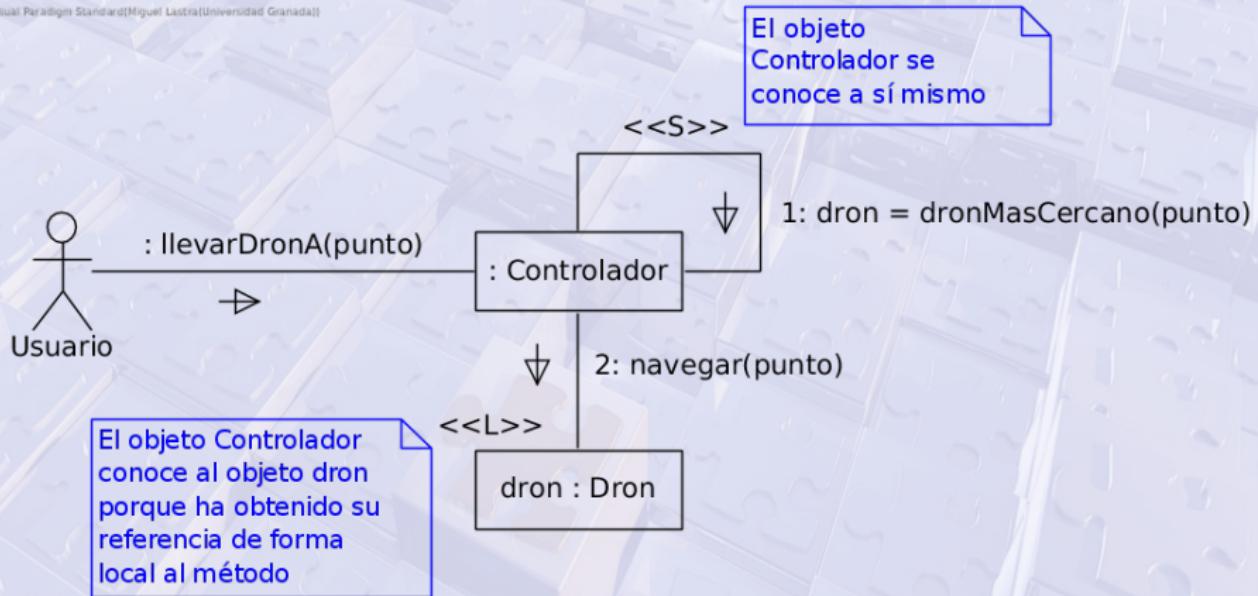
Controlador

```
+llevarDronA(punto : Lugar)
+alturaDron(idDron : int) : float
+incluirNuevoDron(dron : Dron, lugar : Lugar)
+aterrizarDronesBajoAltura(alt : float)
-dronMasCercano(punto : Lugar) : Dron
-getDron(idDron : int) : Dron
```



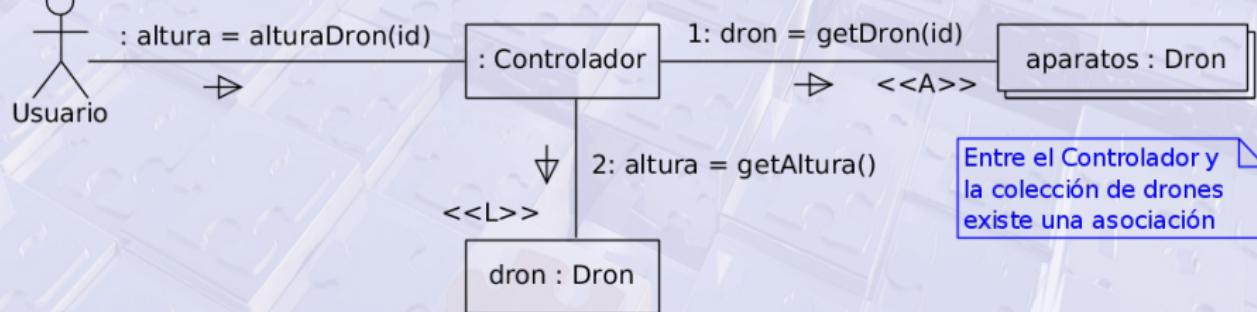
Ejemplo 1

Visual Paradigm Standard (Miguel Lastra (Universidad Granada))



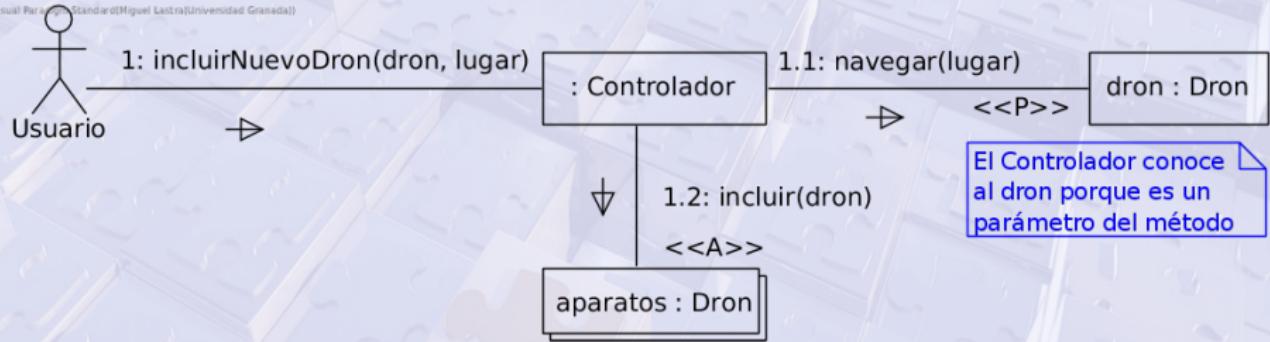
Ejemplo 2

Visual Paragon Standard (Miguel Lastra (Universidad Granada))



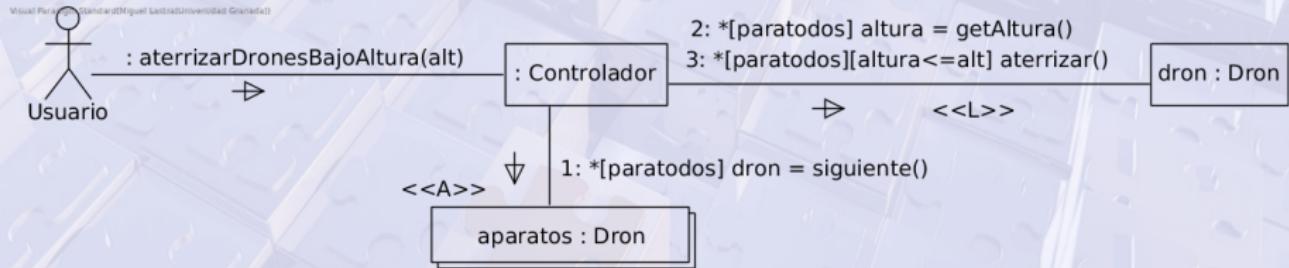
Ejemplo 3

Visual Paradigm Standard (Miguel Lastra / Universidad de Granada)



Ejemplo 4

• Condicionales y bucles



- Recordar que el objetivo de los diagramas UML son:
 - ▶ Especificar las características de un sistema antes de su construcción
 - ▶ Visualizar gráficamente un sistema software de forma que sea entendible
 - ▶ Documentar un sistema para facilitar su mantenimiento, revisión y modificación
- En definitiva, facilitar la tarea del equipo de desarrollo
- Si la especificación de un método (sobre todo los de comunicación) es una maraña de flechas donde es más fácil perderse que aclararse:
 - ① Tal vez ese tipo de diagrama no sea el más adecuado para esa especificación
 - ② Tal vez haya que subdividir un diagrama grande en varios pequeños
 - ③ Tal vez el método deba subdividirse en diversas tareas más pequeñas y más fáciles de especificar de una manera clara y fácilmente entendible (supondrá un desarrollo y mantenimiento más fácil)

UML: Diagramas de Interacción

Prof. Francisco Velasco Anguita

Dpto. Lenguajes y Sistemas Informáticos
Universidad de Granada

Programación y Diseño Orientado a Objetos

Doble Grado en Ingeniería Informática
y Administración y Dirección de Empresas
(Curso 2024-2025)