



Ingeniería Informática + ADE

Universidad de Granada (UGR)

Autor: Ismael Sallami Moreno

Asignatura: Programación y Diseño Orientado a Objetos (PDOO)



Índice

1. Teoría	3
1.1. Tema 1	3
1.1.1. Conceptos Básicos	3
1.2. Tema 2	23
1.2.1. Atributos y Métodos	23
1.2.2. Construcción de Objetos	51
1.2.3. Consultores y Modificadores	79
1.2.4. Elementos de Agrupación	95
1.2.5. Diagramas Estructurales	111
1.2.6. Diagramas de Interacción	137
1.3. Tema 3	161
1.3.1. Herencia	161
1.3.2. Visibilidad	198
1.3.3. Polimorfismo	237
1.3.4. Clases Abstractas	259
1.3.5. Clases Parametrizables	281
1.3.6. Herencia en Ámbito de Clases	296
1.3.7. Herencia Múltiple	307
1.4. Tema 4	324
1.4.1. Modelo Vista Controlador	324
1.4.2. Reflexión	338
1.4.3. Copia de Objetos	350
2. Relaciones de Problemas	384
2.1. Relación 1 con soluciones	384
2.2. Relación 2 con soluciones	386
2.3. Relación 3	448
2.3.1. Soluciones	460
3. Exámenes	494
3.1. Examen 1	494

1 Teoría

1.1. Tema 1

1.1.1. Conceptos Básicos

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:

- ▶  <https://pixabay.com/images/id-37254/>
- ▶  Emojis, <https://pixabay.com/images/id-2074153/>
- ▶  <https://pixabay.com/images/id-2495144/>
- ▶  <https://pixabay.com/images/id-3687611/>
- ▶  <https://pixabay.com/images/id-29094/>

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

Objetivos

- Tener un primer contacto con el paradigma de la programación orientada a objetos
- Entender los siguientes conceptos:
 - ▶ Objeto
 - ▶ Clase
 - ▶ Identidad
 - ▶ Estado
 - ▶ Comportamiento
- Entender ejemplos sencillos

Contenidos

1 Conceptos

2 Paradigma de Programación Orientada a Objetos

3 Ejemplos

Concepto de Objeto

- Entidad perfectamente delimitada, que **encapsula estado y funcionamiento** y **posee una identidad** (OMG 2001)
- Elemento, unidad o entidad individual e identifiable, real o abstracta, con un **papel bien definido en el dominio del problema** (Dictionary of Object Technology 1995)
 - ¿Qué significa cada una de estas frases?
 - ▶ Soy un ejemplar de Lápiz
 - ▶ Soy único
 - ▶ Mi color es el verde
 - ▶ Como todos los lápices, puedo dibujar
 - ▶ Yo, además, puedo borrar

Clase

- La clase, entre otras cosas, actúa de **molde o plantilla** para la creación de objetos
 - ▶ En algunos lenguajes las clases son también objetos a todos los efectos
- Los **objetos** creados a partir de una clase se denominan **instancias** de esa clase
 - ▶ Esos objetos *pertenecen* o simplemente *son* de esa clase.
- **Una clase crea un tipo de dato.** Se pueden declarar variables de ese tipo o clase (si el lenguaje dispone de este mecanismo)

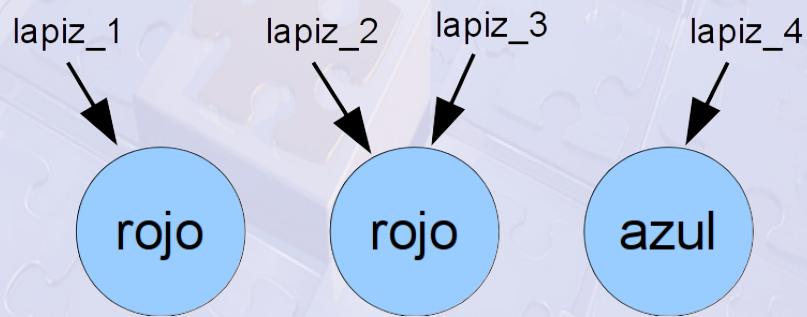
Java: Instanciando clases, creando objetos

```
1 Lapiz miLapiz = new Lapiz (Color.Rojo);  
2 Lapiz tuLapiz = new Lapiz (Color.Verde);
```

Identidad

- Cada instancia tiene su propia identidad
- La identidad la define la **posición de memoria**
- Independientemente de su contenido, objetos distintos residirán en zonas de memoria distintas

★ En el ejemplo, ¿qué lápices son iguales a otros?, ¿se puede considerar más de un criterio de igualdad?, ¿cuáles?



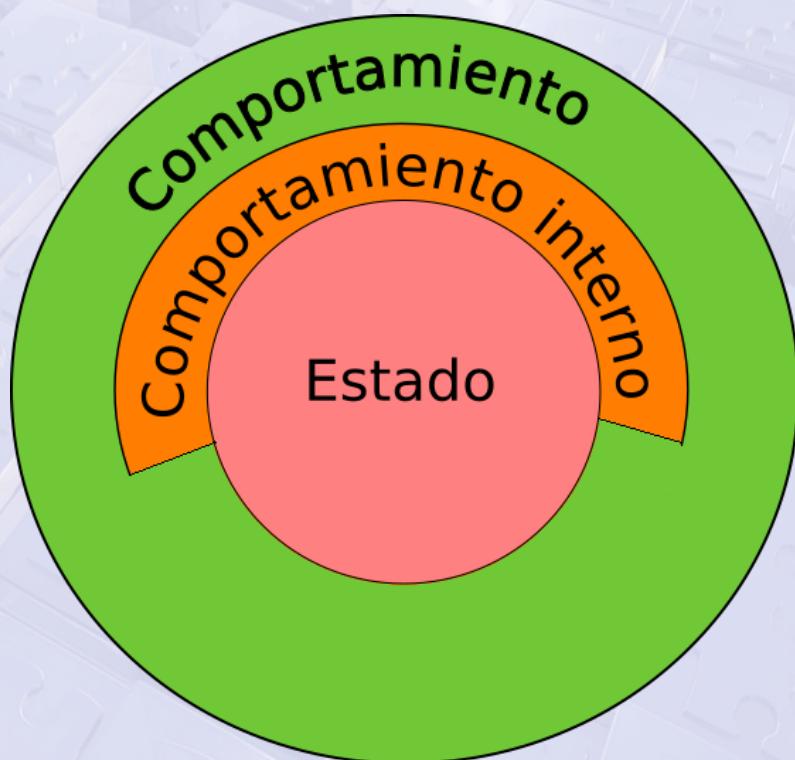
Estado y Comportamiento

- El **estado** de un objeto vendrá definido por los **valores de sus atributos**
 - ▶ Cada objeto tiene una **zona de memoria propia** para el almacenamiento de sus atributos
- Los objetos exhiben **comportamiento**
 - ▶ Disponen de una serie de **métodos** (funciones o procedimientos) que pueden ser llamados/invocados

Ejemplo en C++: Invocando métodos de objetos

```
1 Persona amparo( "Amparo" ), samuel( "Samuel" );
2 Persona * cristina = new Persona ( "Cristina" ); // Puntero C++
3 // ...
4 amparo.saluda() // Se invoca al método: saluda
5 // "Hola, me llamo Amparo"
6 samuel.saluda() // otra instancia distinta
7 // "Hola, me llamo Samuel"
8 cristina->saluda() // C++
9 // "Hola, me llamo Cristina"
```

Estado y Comportamiento



Paradigma de Programación Orientada a Objetos

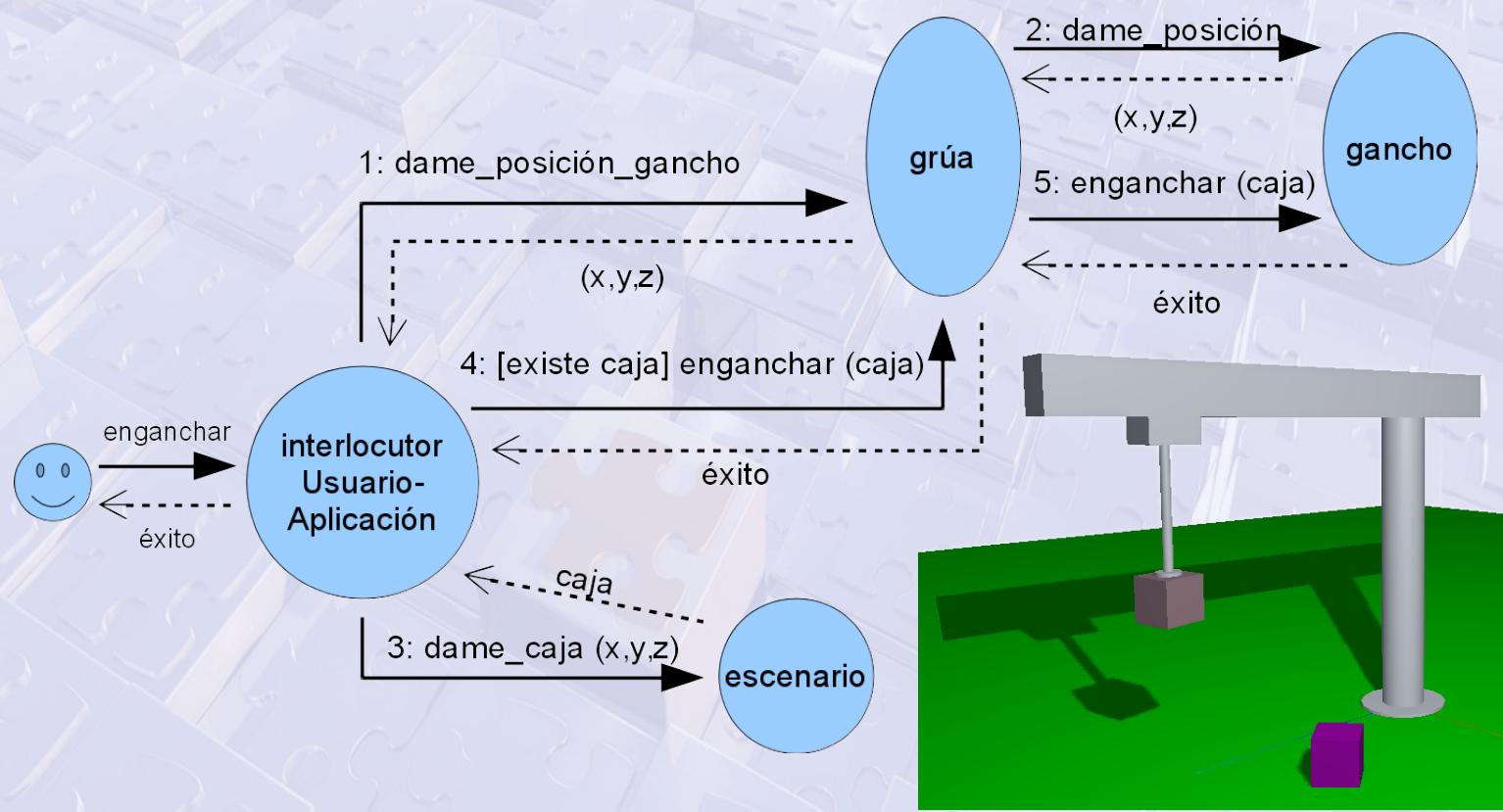
- **Paradigma:** Teoría o conjunto de teorías cuyo núcleo central se acepta sin cuestionar y que suministra la base y modelo para resolver problemas y avanzar en el conocimiento (R.A.E)
- **Paradigma de programación:** Conjunto de reglas que indican como desarrollar software
- Base de la **orientación a objetos:** Se unen los **datos** y el **procesamiento** en entidades denominadas **objetos**

Programación Orientada a Objetos (POO)

- Los objetos son las entidades que se manejan en el software
- Programar *consiste* en modelar el problema mediante un universo dinámico de objetos
- Cada objeto pertenece a una clase y como tal, tiene una responsabilidad dentro de la aplicación
- La funcionalidad del programa se obtiene haciendo que unos objetos *le pidan* a otros objetos (envío de mensajes) que hagan cosas (ejecución de métodos)
 - ▶ Se deben programar los métodos de cada clase de manera que cada objeto se ocupe de lo suyo y no haga el trabajo de otro 😊
- El objetivo es obtener alta cohesión y bajo acoplamiento

Ejemplo

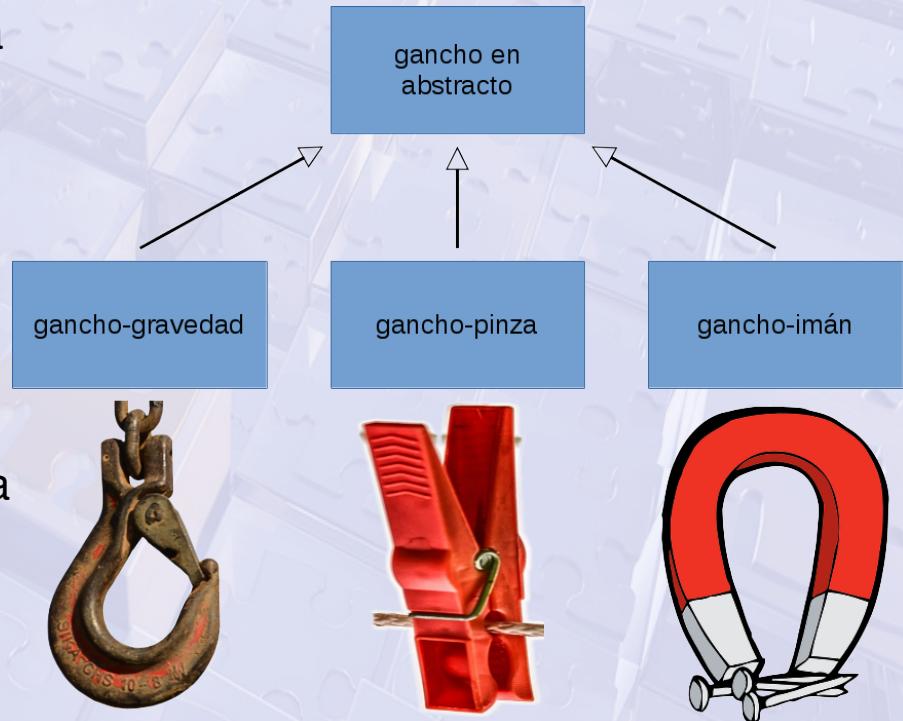
Simulador de entrenamiento de operario de grúa



POO

Conceptos básicos

- El paradigma se basa en los siguientes conceptos:
 - ▶ Clase
 - ▶ Objeto o instancia
 - ▶ Estado
 - ▶ Identidad
 - ▶ Mensaje
 - ▶ Herencia
 - ▶ Polimorfismo
 - ▶ Ligadura dinámica



Primeros Ejemplos

- Este es el aspecto de dos ejemplos de programas orientados a objetos escritos en los lenguajes de programación Java y Ruby
- Estos ejemplos servirán de ayuda para comenzar el trabajo de las prácticas de la asignatura. En ellos aparecen elementos que veremos con detalle más adelante

Java: Ejemplo básico

```
1 package basico;
2
3 // Enumerado con visibilidad de paquete
4 /* public */ enum ColorPelo { MORENO, CASTAÑO, RUBIO, PELIROJO }
5
6 public class Persona { // Clase con visibilidad pública
7     private String nombre; // Atributos de instancia privados
8     private int edad;
9     private ColorPelo pelo;
10
11    public Persona (String n,int e,ColorPelo p) { // Constructor público
12        nombre=n;
13        edad=e;
14        pelo=p;
15    }
16
17    void saluda() { // Visibilidad de paquete. Método de instancia
18        System.out.println("Hola, soy "+nombre);
19    }
20 }
21
22 public class Basico { // Clase con programa principal
23     public static void main(String[] args) {
24         Persona p=new Persona("Pepe",10,ColorPelo.RUBIO);
25         p.saluda();
26     }
27 }
```

Ruby: Ejemplo básico

```
1 #encoding: UTF-8
2 module Basico
3   module ColorPelo
4     MORENO= :moreno
5     CASTAÑO= :castaño
6     RUBIO= :rubio
7     PELIROJO= :pelirojo
8   end
9
10  class Persona
11    def initialize(n,e,p) # "constructor"
12      # Atributos de instancia (son privados)
13      @nombre=n
14      @edad=e
15      @pelo=p
16    end
17
18    public # aunque los métodos son públicos por defecto
19    def saluda # Método público de instancia
20      puts "Hola, soy "+@nombre
21    end
22  end
23
24 p=Persona.new("Pepe",10,ColorPelo::RUBIO)
25 p.saluda
26 end
```

Java: Uso desde otro paquete

```
1 // En otro paquete
2 package otroPaquete;
3
4 import basico.Persona;
5 import basico.ColorPelo;    // Error: ColorPelo no tiene visibilidad pública
6
7 // ...
8
9 Persona manolo = new Persona ("Manolo", 20, ColorPelo.MORENO);
10 // Error: No se reconoce el símbolo ColorPelo
11
12 System.out.println (manolo.toString ());
13 // basico.Persona@33909752
14 // Para que muestre información útil hay que redefinir toString ()
```

Ruby: Uso desde otro módulo

```
1 # Fuera del módulo que hemos llamado "Basico"
2
3 manolo = Basico::Persona.new("Manolo", 20, Basico::ColorPelo::MORENO)
4
5 puts manolo.inspect
6 #< Basico::Persona:0x5571 @nombre="Manolo",@edad=20,@pelo=:moreno >
```

Objetos y clases

→ **Diseño** ←

- Determinar qué objetos (y por extensión, qué clases) van a modelar mejor el sistema no es una tarea fácil
 - ▶ Hablamos de “Diseñar Software”, algo que se empieza a aprender en la titulación, y no se termina de aprender nunca
- No obstante, las clases:
 - ▶ Deben tener una responsabilidad muy concreta
 - ★ Si una clase se ocupa de muchas cosas, tal vez haya que crear varias clases y distribuir responsabilidades
 - ▶ Deben ser, en cierta medida, “autónomas”
 - ★ Si una clase se ve muy afectada por cambios realizados en otras clases, tal vez esa clase tiene responsabilidades que no le corresponden
 - ▶ Deben ser “introvertidas” y “no altruistas”
 - ★ El estado de una clase solo debe modificarse desde la propia clase
 - ★ Ninguna clase debe hacer el trabajo que le corresponde a otra clase

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)

Ejemplo en ruby de buen y mal diseño

1.2. Tema 2

1.2.1. Atributos y Métodos

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

Atributos de instancia

Ejemplo: La clase Persona

```

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

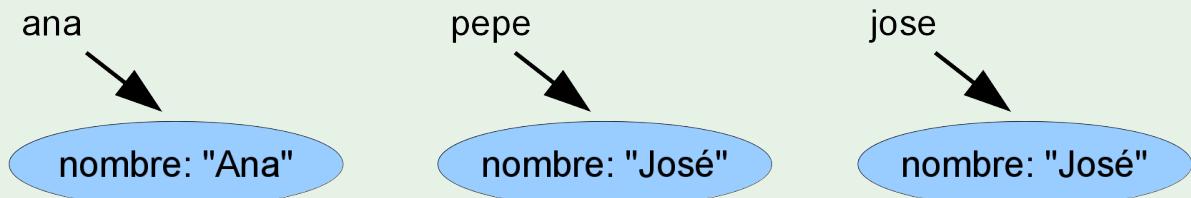
```

- 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
5     String saludar () {
6         return "Hola, me llamo " + nombre;
7     }
8
9     void cambiaNombre (String otroNombre) {
10        nombre = otroNombre;
11    }
12 }
```

Ejemplo: La clase Persona

```

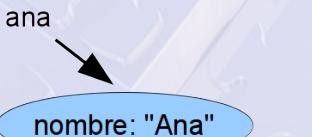
1 Persona pepe = new Persona ("José")
2 Persona ana = new Persona ("Ana");
3
4 System.out.println (ana.saludar ());
5 // Hola, me llamo Ana
6
7 ana.cambiaNombre ("Ana Belén");
8
9 System.out.println (ana.saludar ());
10 // Hola, me llamo Ana Belén
11
12 System.out.println (pepe.saludar ());
13 // Hola,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

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
```



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

- ¿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?

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

Ejemplo

Java: Constante vs. número mágico

```
1 class Factura {  
2     . . .  
3     float calculaIVA (float baselponible) {  
4         return baselponible * 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 }  
14  
15 class Factura {  
16     . . .  
17     float calculaIVA (float baselponible) {  
18         return baselponible * GestionTributaria.IVA;    // Uso de la variable de clase  
19     }  
20 }
```

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?

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

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
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
```

→ *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

★ ¿Cuál es el resultado de ejecutar este programa?

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`?

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

```

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

Ejemplos

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

```
1 class Producto
2   @@iva = 21
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)
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
```

★ ¿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?

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

```

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

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

Ruby: self

```

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
```

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

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)

1.2.2. Construcción de Objetos

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?

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**

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
```

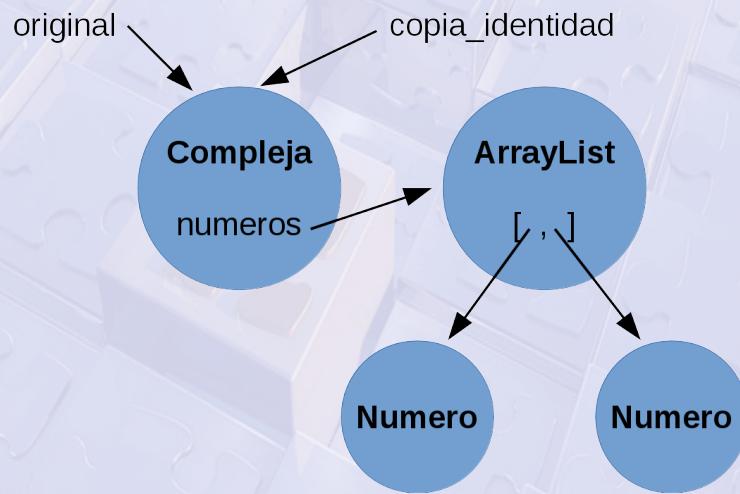
Ejemplos

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
```

Constructor de copia

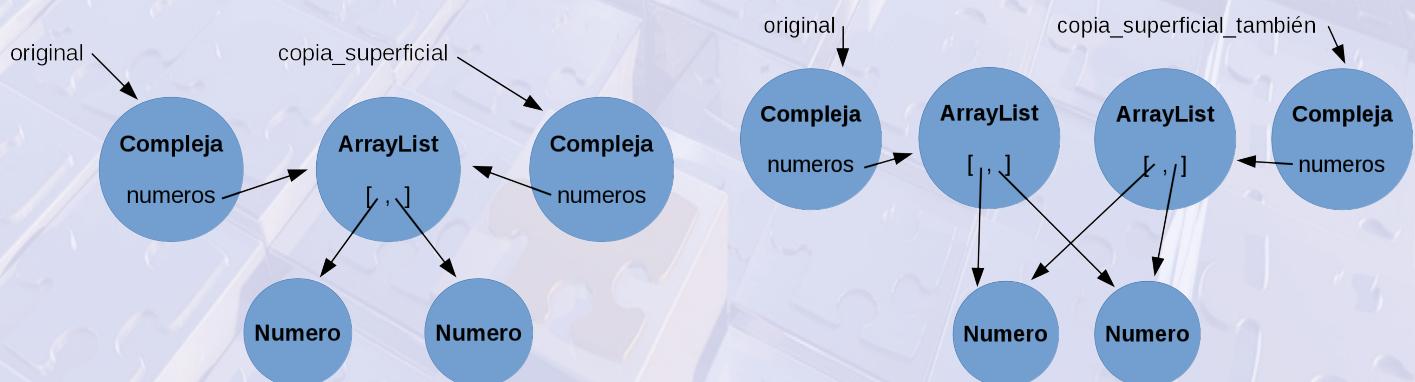
- Un constructor de copia recibe un objeto como parámetro
 - ▶ Construye otro objeto (distinta identidad)
 - ▶ Con el mismo estado (inicialmente) que el objeto recibido
- Puede haber diferentes niveles de copia (superficial y profunda)
 - ▶ Ya conocemos cómo actúa la asignación `copia = original`



Constructor de copia

Copia superficial

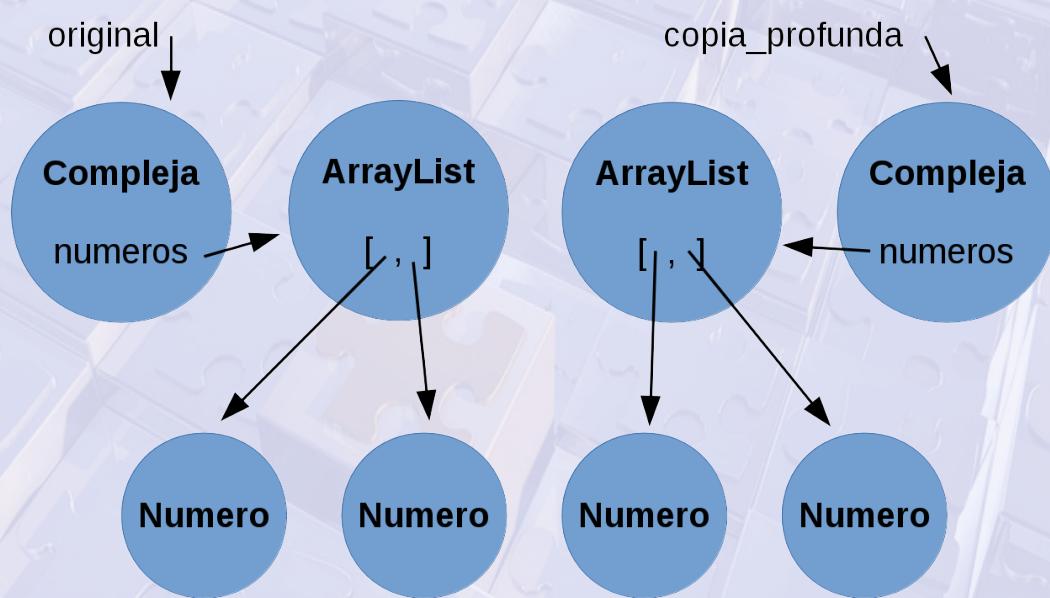
- En una copia superficial, aunque se crea un objeto distinto, en algún nivel se referencia un mismo objeto



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 }
```

Constructores

→ *Diseño* ←



- 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)

1.2.3. Consultores y Modificadores

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
```

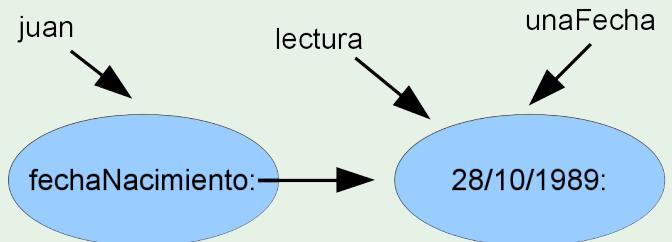
Problemática de devolver (o asignar) referencias

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)

1.2.4. Elementos de Agrupación

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

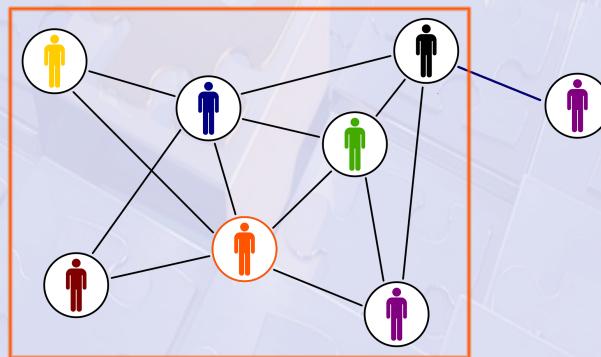
- Saber crear y usar paquetes en Java
- Saber crear y usar módulos en Ruby
- Gestionar correctamente las líneas `require_relative` en Ruby

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

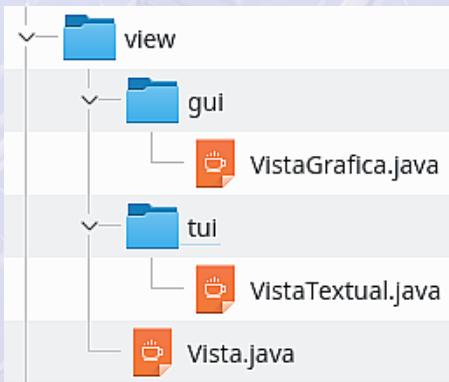
```
1 package miPrograma;  
2 // Los elementos que se definan en este archivo pertenecerán al paquete miPrograma  
3 import modelo.Fachada; // En este fichero se va a usar la clase Fachada del paquete modelo
```

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?

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 <NombreMódulo>::
 - ▶ 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  end
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

Paquetes y módulos

→ **Diseño** ←

- ¿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)

1.2.5. Diagramas Estructurales

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)

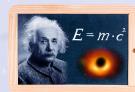
Créditos I

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

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



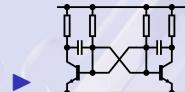
<https://pixabay.com/images/id-1044090/>



<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

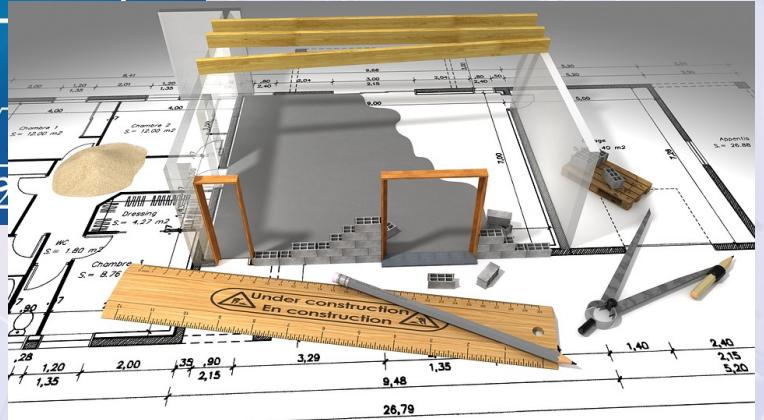
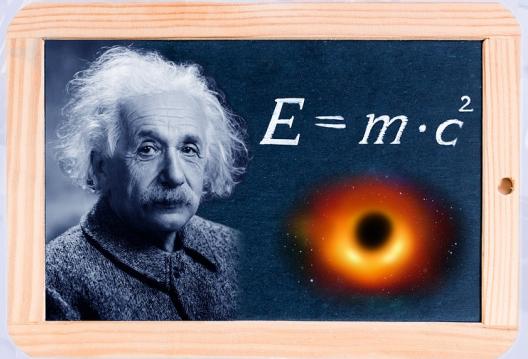
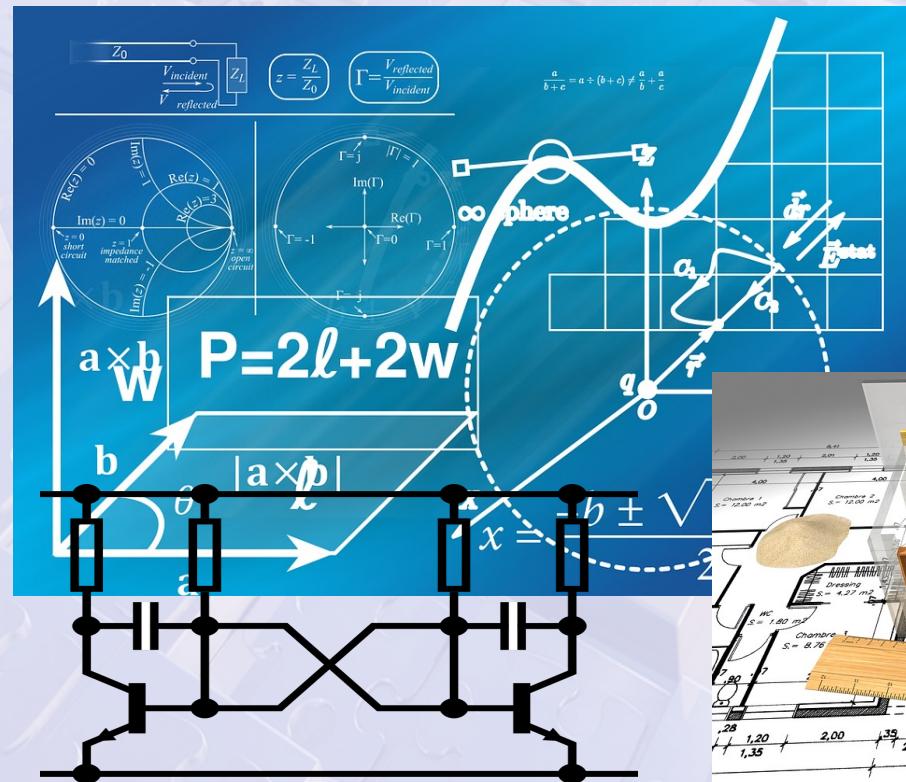
3 Relaciones entre clases

- 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  
#protector : float  
~paquete : OtraClase [1..*]  
-privado : boolean  
  
+metodoClase(a : int) : void  
+deInstanciaPublico(a : float, b : int[]) : int  
-deInstanciaPrivado()
```

Representación de una clase

Visual Paradigm Standard/zoraida(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)



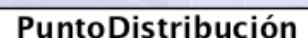
1..*

bienes

< distribuye

1

distribuidor



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

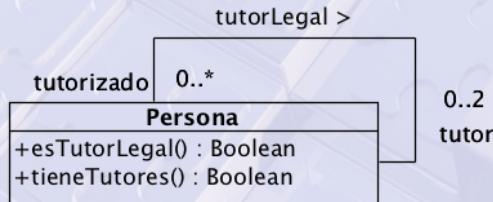
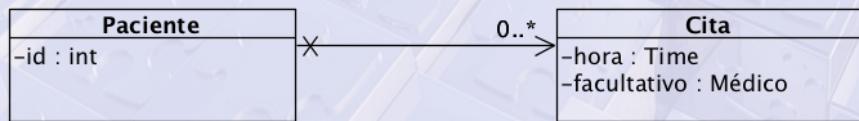
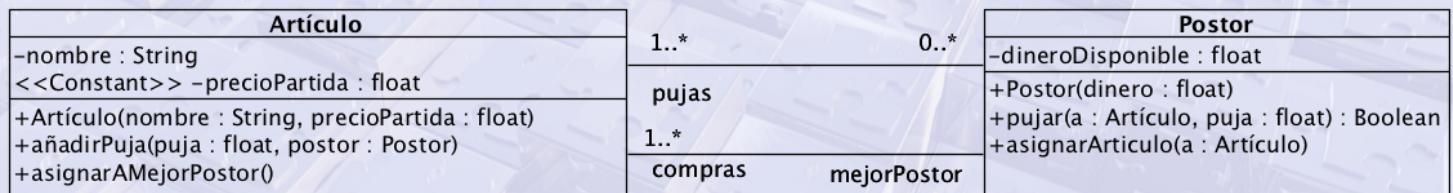
Navegabilidad: La asociación es bidireccional, por lo tanto el punto de distribución puede conocer los productos que distribuye y cada producto conoce también cuál es su punto de distribución

Relaciones entre clases

Asociación

Ejemplos de asociaciones

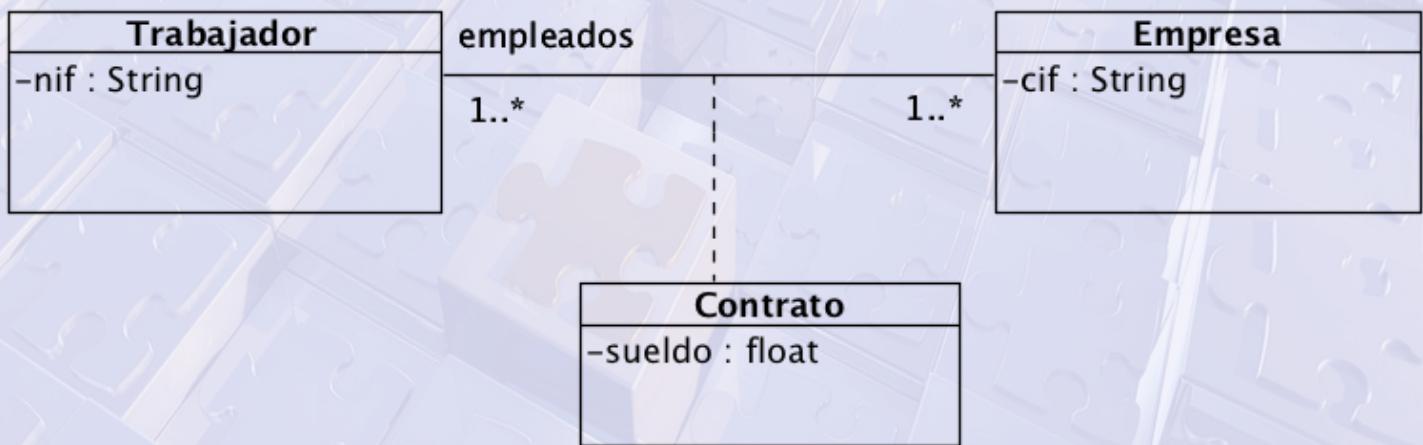
Visual Paradigm Standard (zoraida) (Universidad Granada)



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

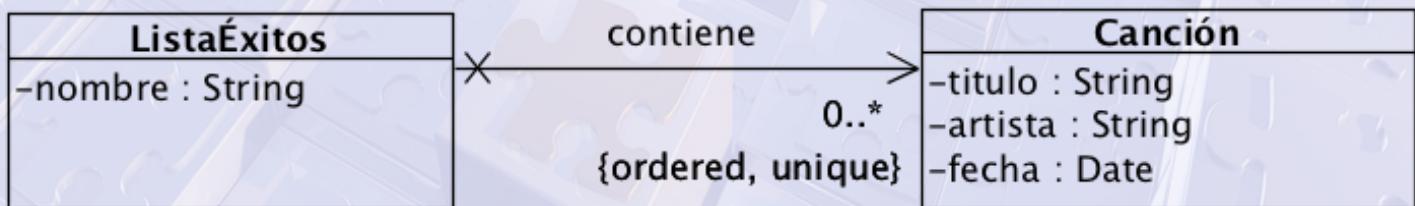
Visual Paradigm Standard(zoraida(Universidad de Granada))



Asociación: Propiedades de los extremos

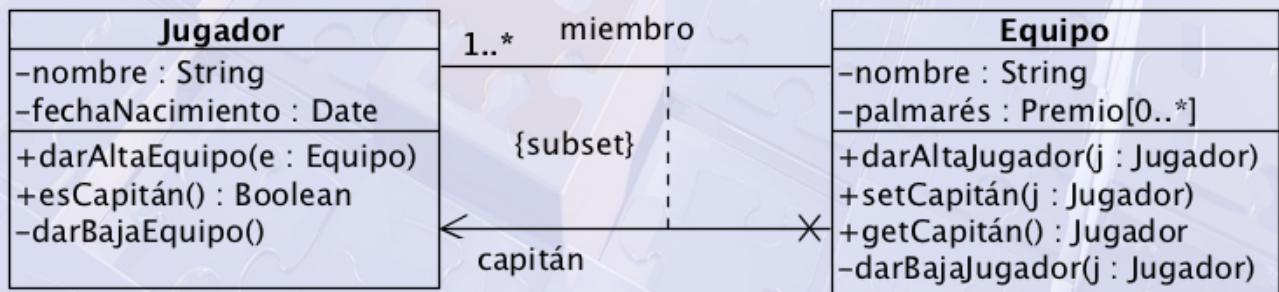
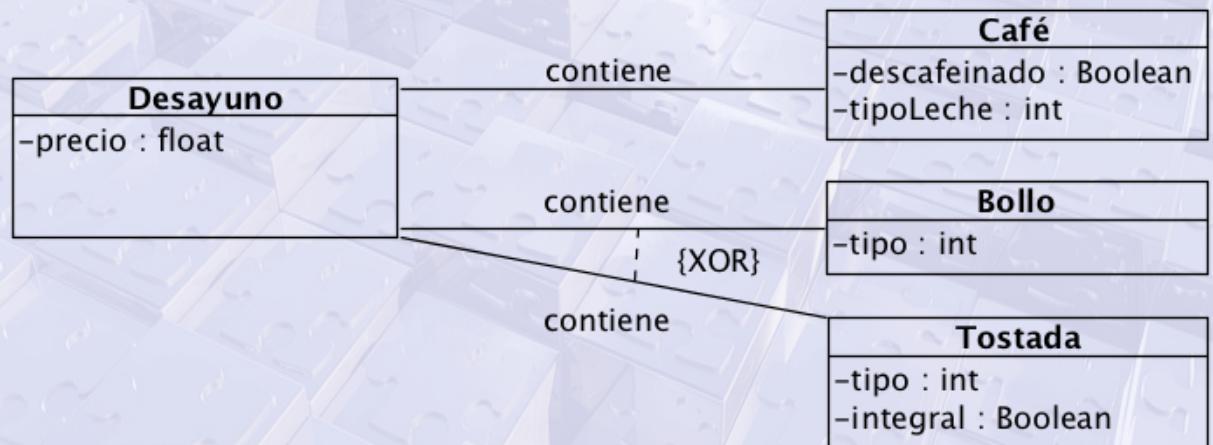
- En los extremos de las asociaciones se pueden indicar **propiedades**
- Las más comunes con multiplicidad mayor a 1 son:
 - ▶ **{ordered}** para indicar que se trata de una secuencia ordenada
 - ▶ **{unique}** para indicar que los elementos no se repiten

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 las 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

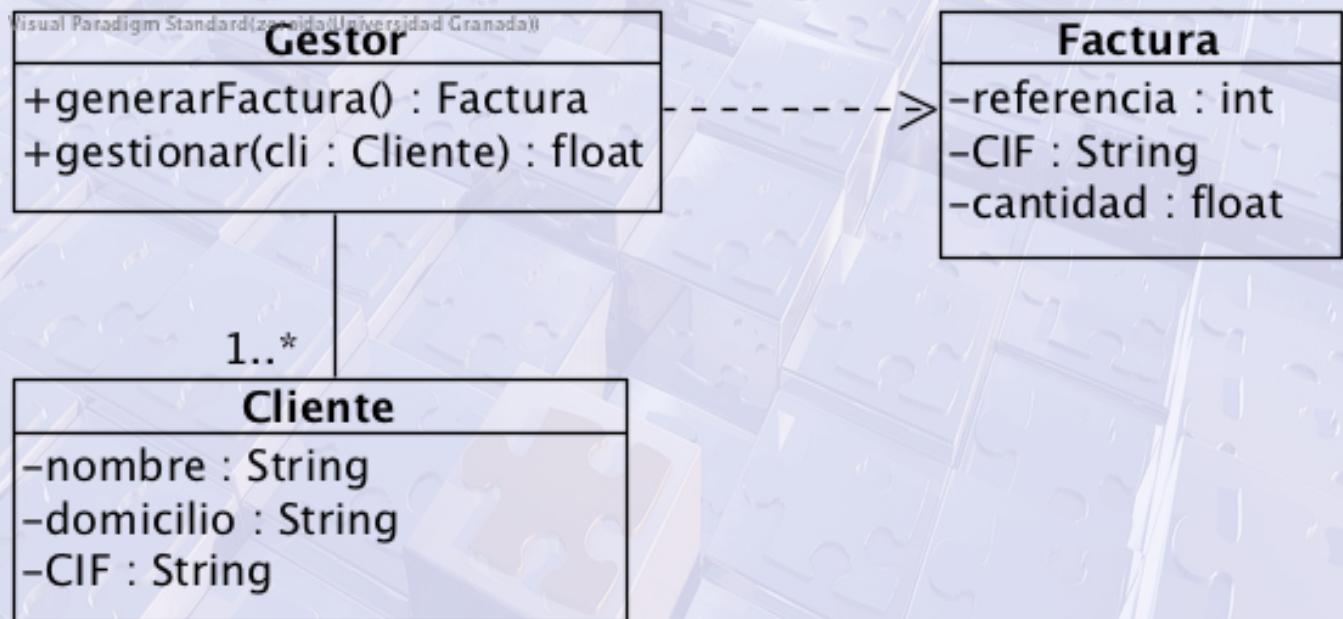


Diagrama de paquetes

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

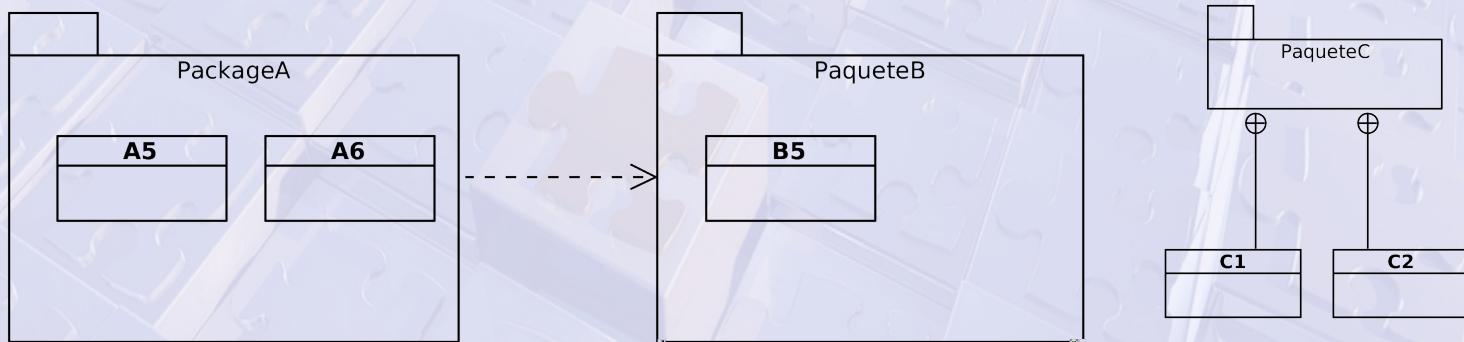


Diagrama de clases

→ **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

Diagrama de clases

→ **Diseño** ←

- 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)

1.2.6. Diagramas de Interacción

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

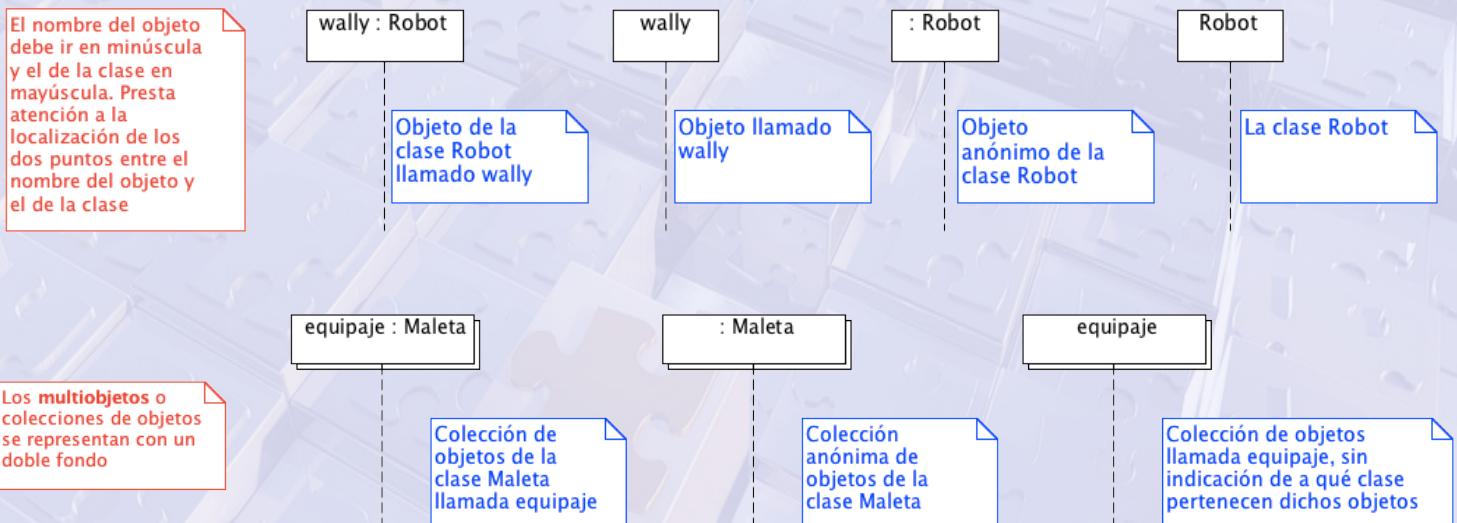
3 Diagramas de comunicación

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

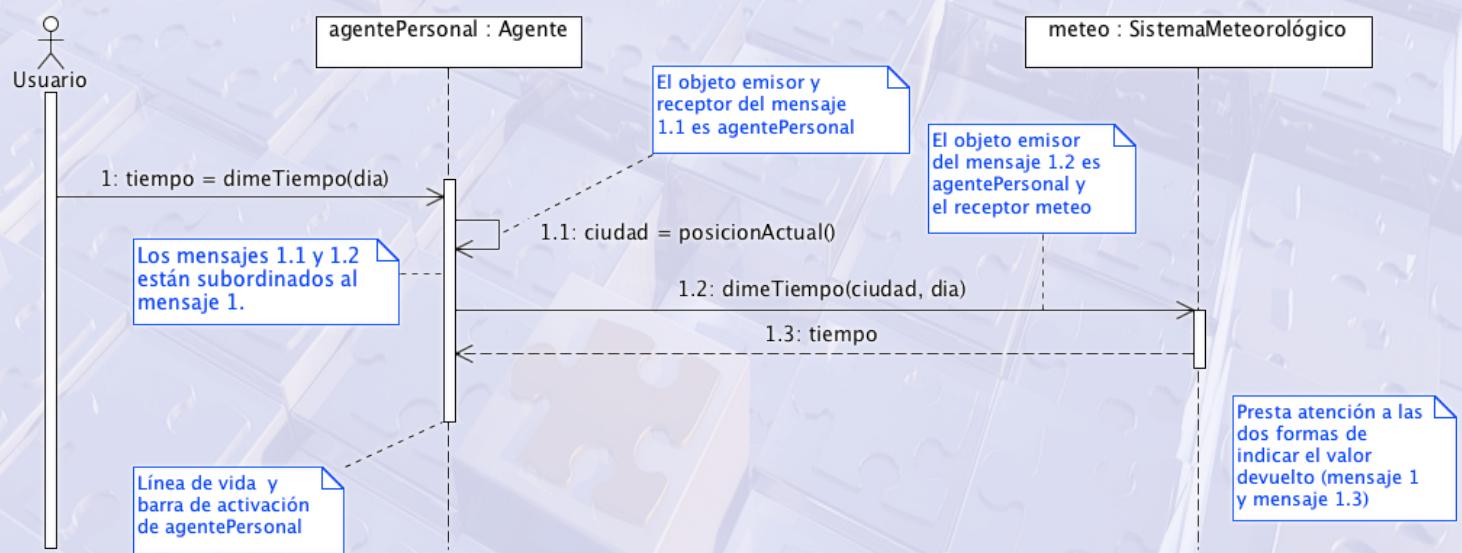
Diagramas de secuencia

- Los participantes se muestran en una caja



Diagramas de secuencia

- **Mensajes:** Emisor y Receptor



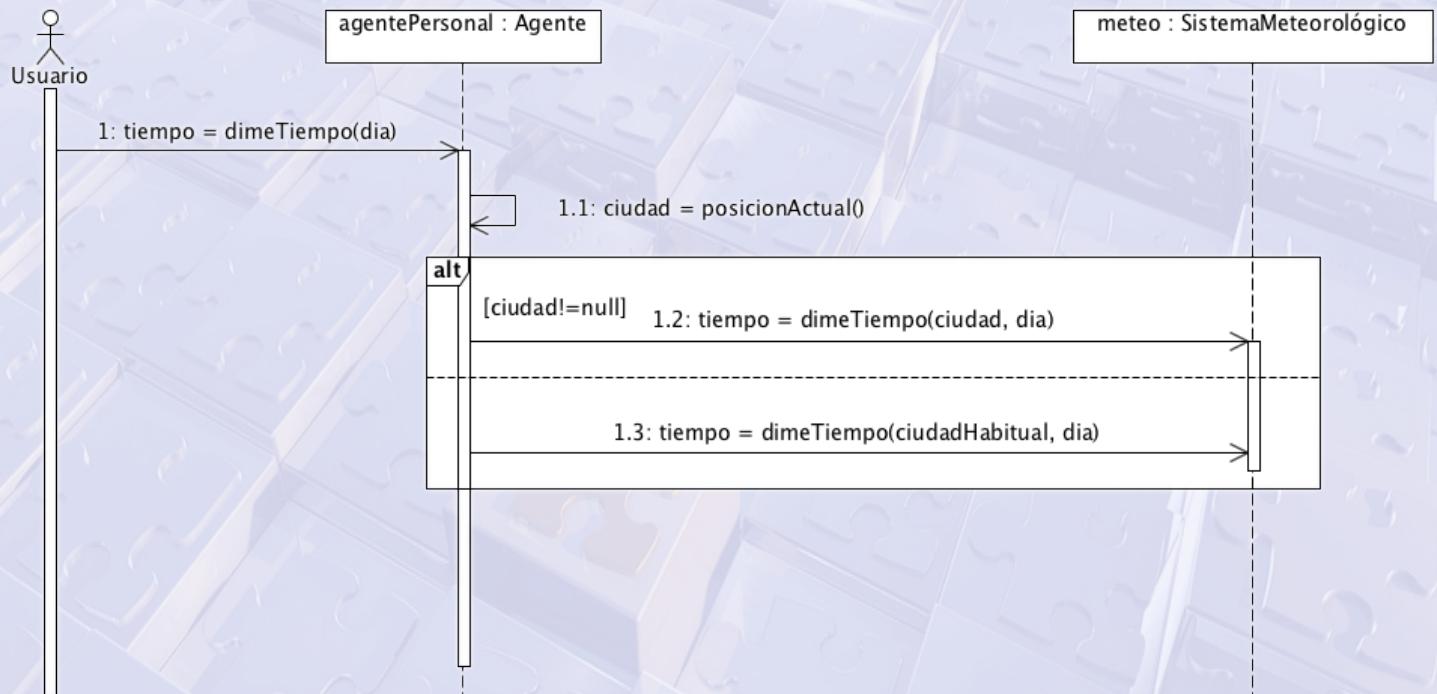
Diagramas de secuencia

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
```

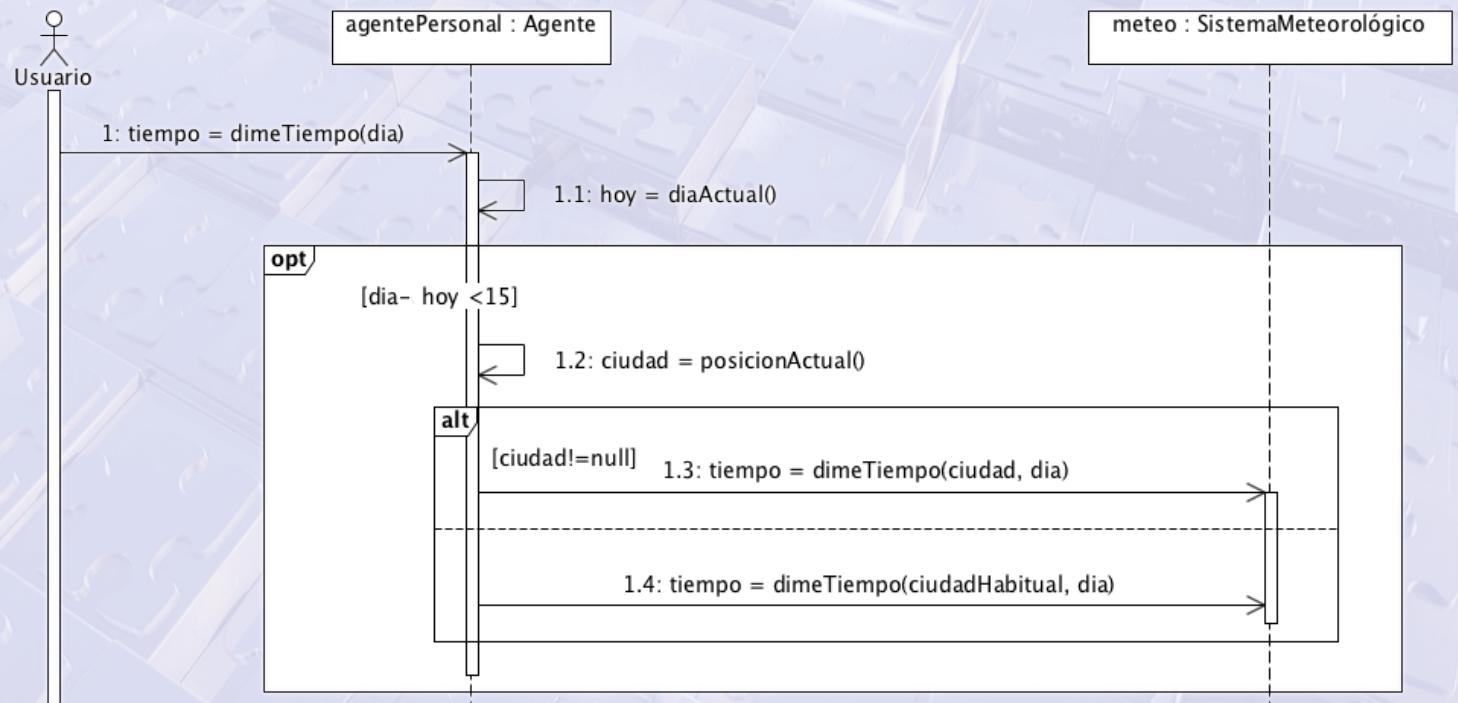
Diagramas de secuencia

- Fragmentos: **Condicionales**



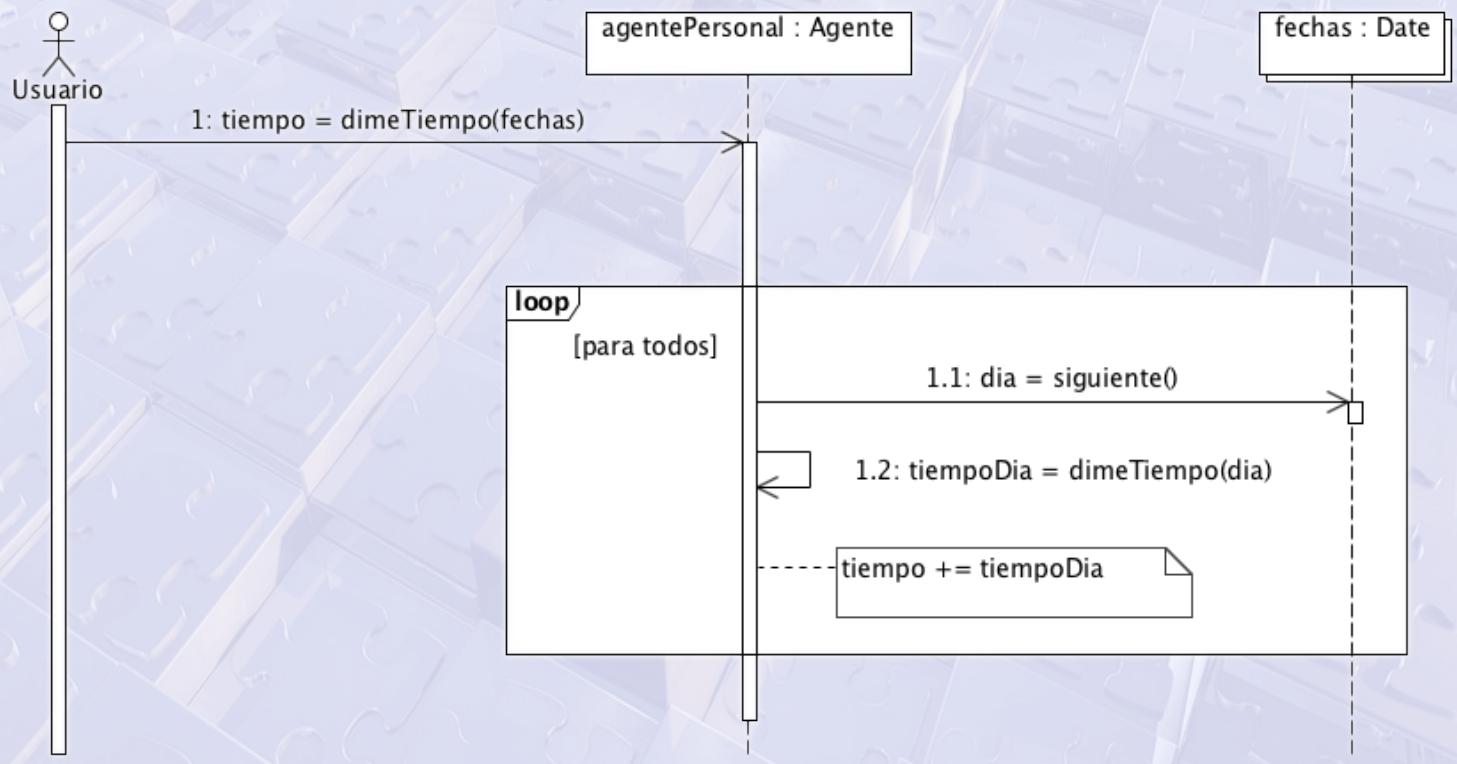
Diagramas de secuencia

- Fragmentos: **Condicionales**



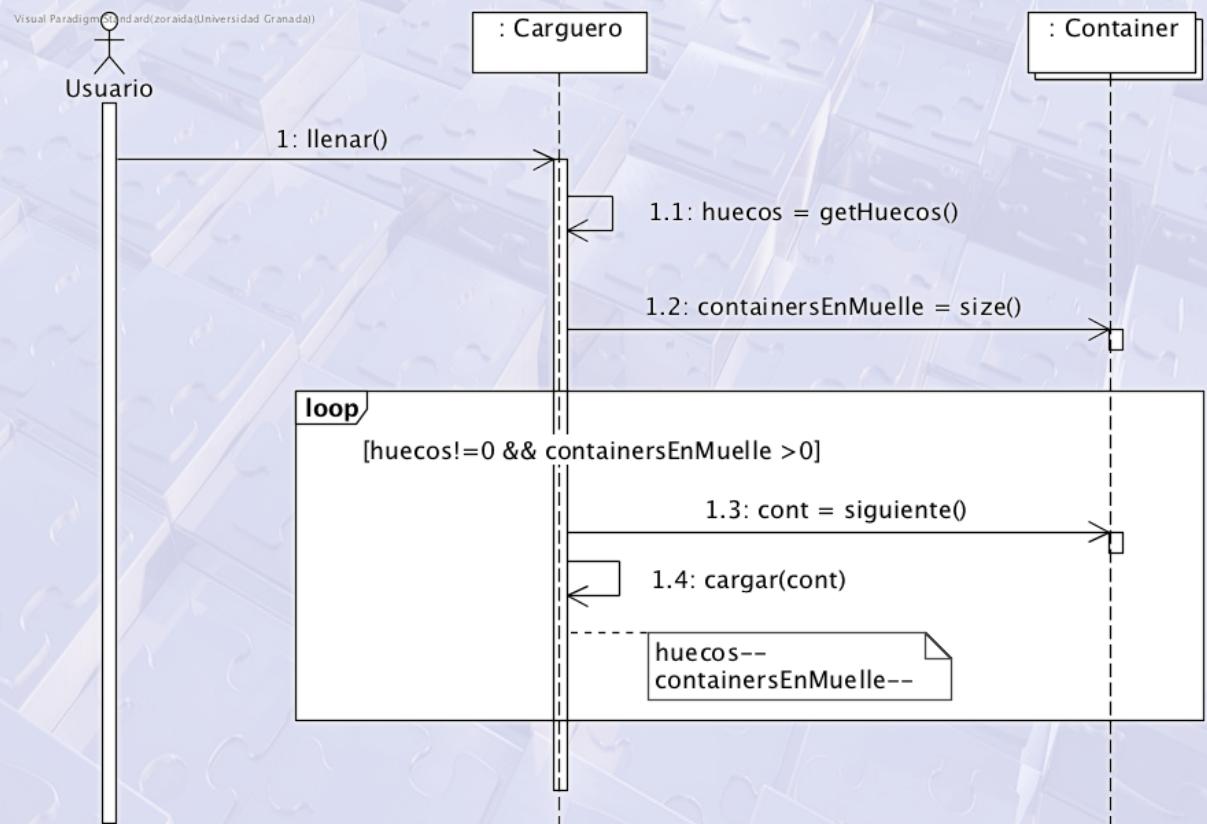
Diagramas de secuencia

- Fragmentos: **Bucles**



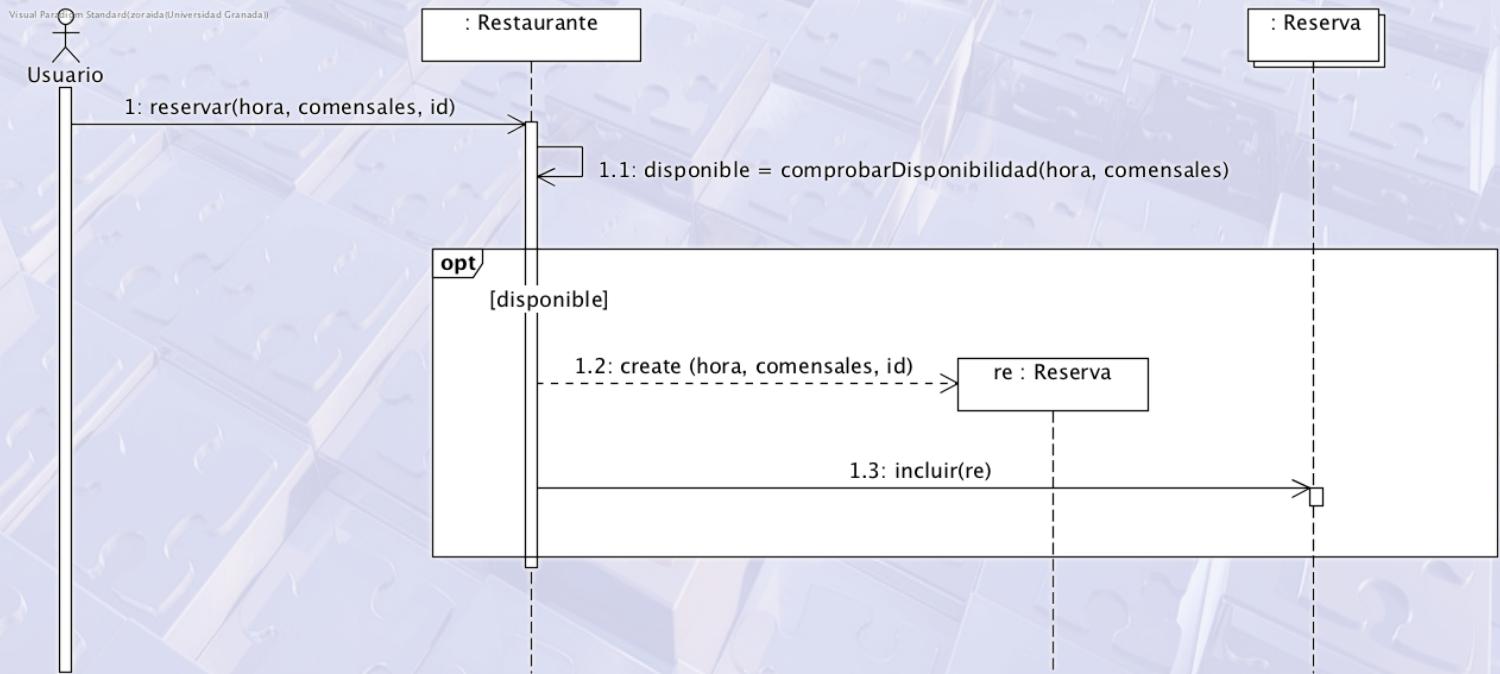
Diagramas de secuencia

- Fragmentos: **Bucles**



Diagramas de secuencia

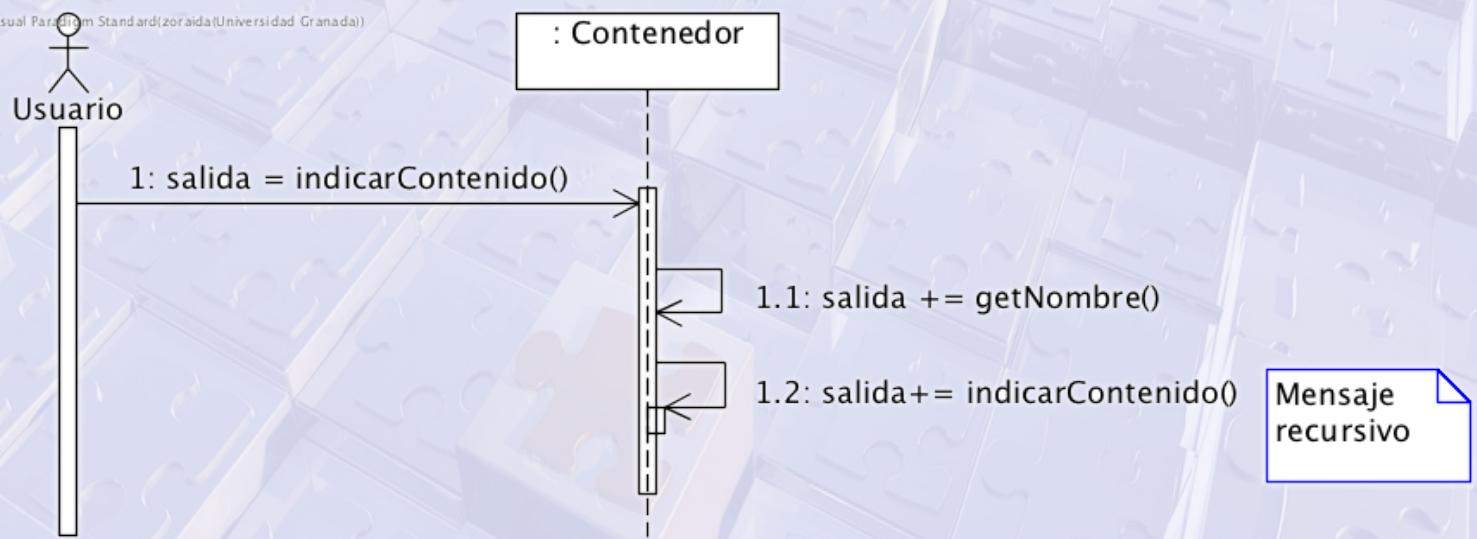
● Creación de instancias



Diagramas de secuencia

● Recursividad

Visual Paradigm Standard(zoraida(Universidad Granada))



Diagramas de comunicación

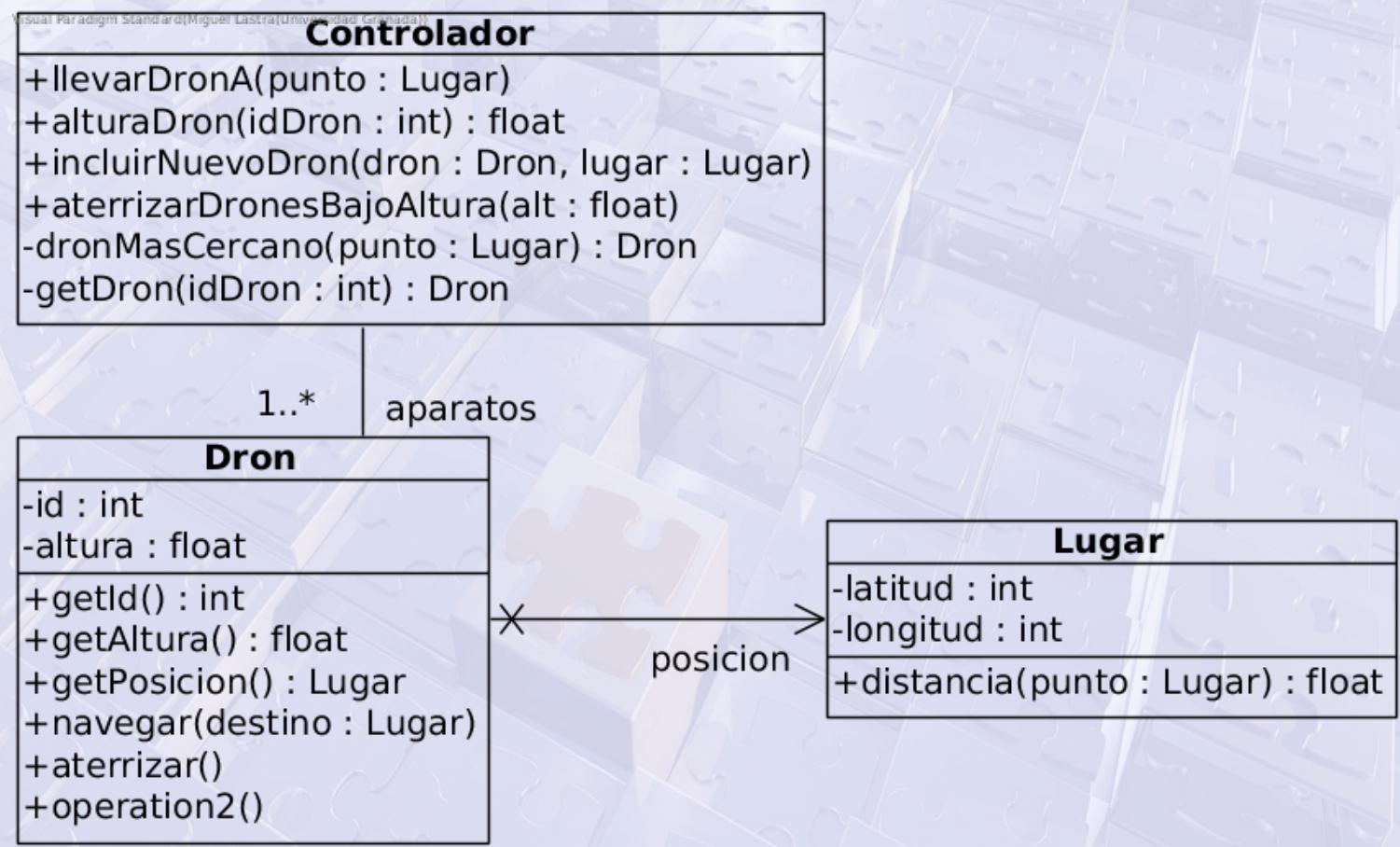
- Muestran de forma visual muy clara las vías de comunicación que deben darse entre los participantes para que pueda llevarse a cabo el envío de mensajes entre ellos
- Las vías de comunicación (enlaces) son el elemento principal y el orden temporal de los mensajes un elemento secundario

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

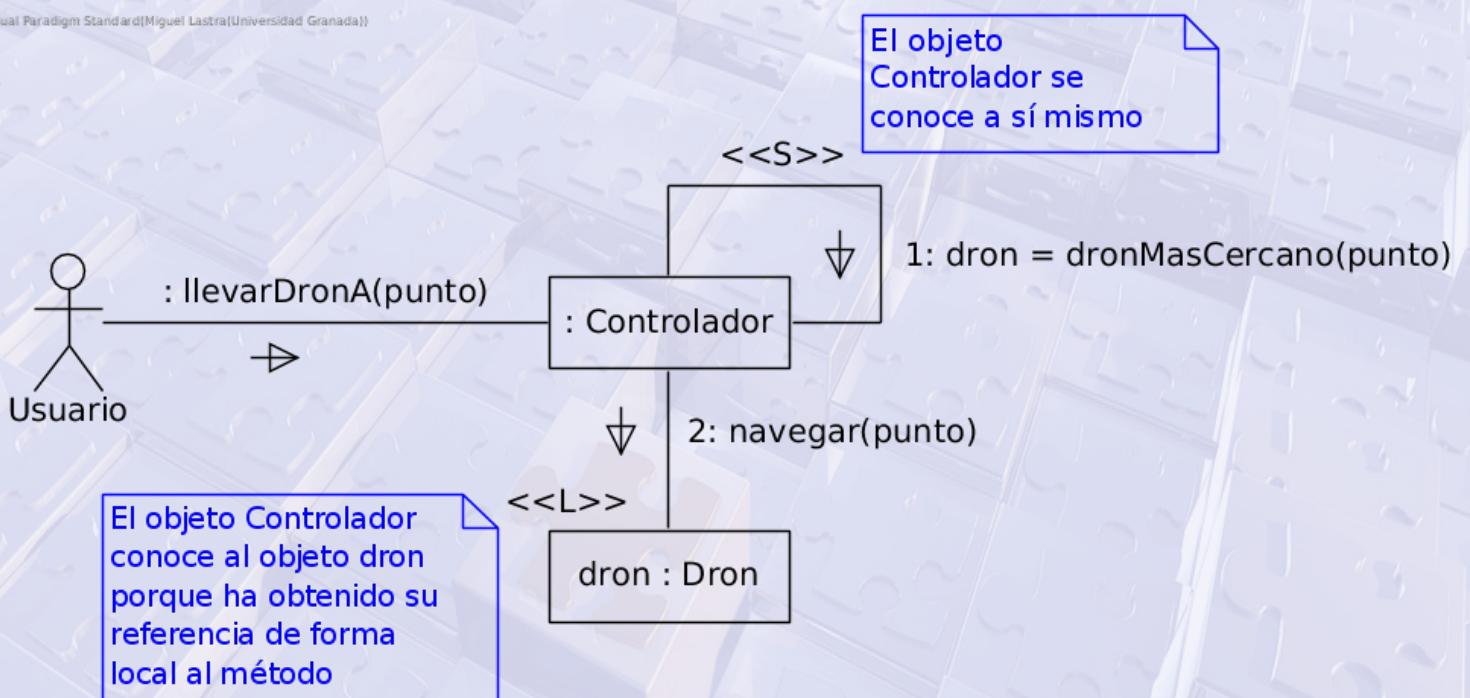
DC para los ejemplos siguientes

Visual Paradigm Standard (Miguel Lastra (Universidad de Granada))

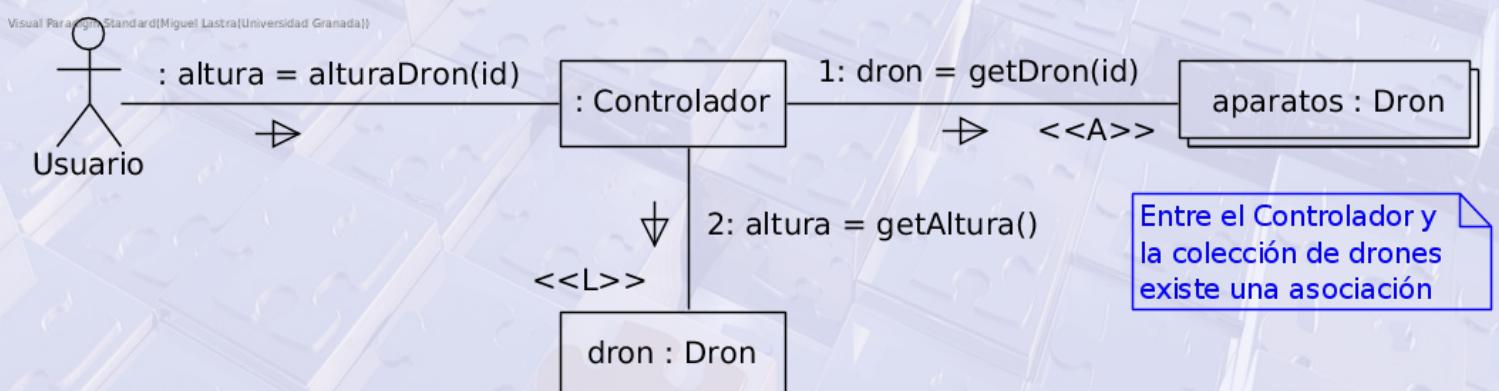


Ejemplo 1

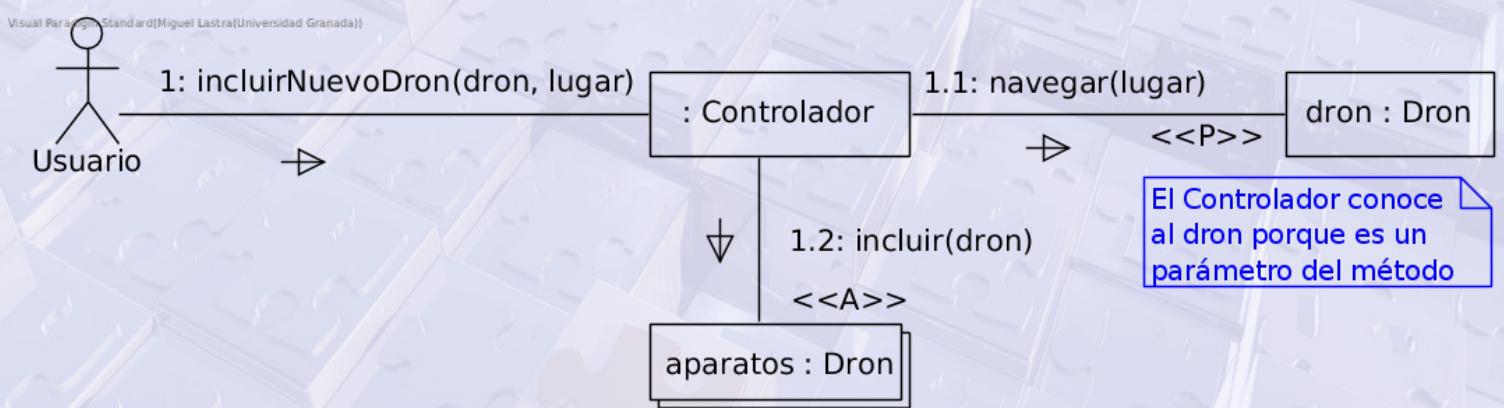
Visual Paradigm Standard (Miguel Lastra (Universidad Granada))



Ejemplo 2

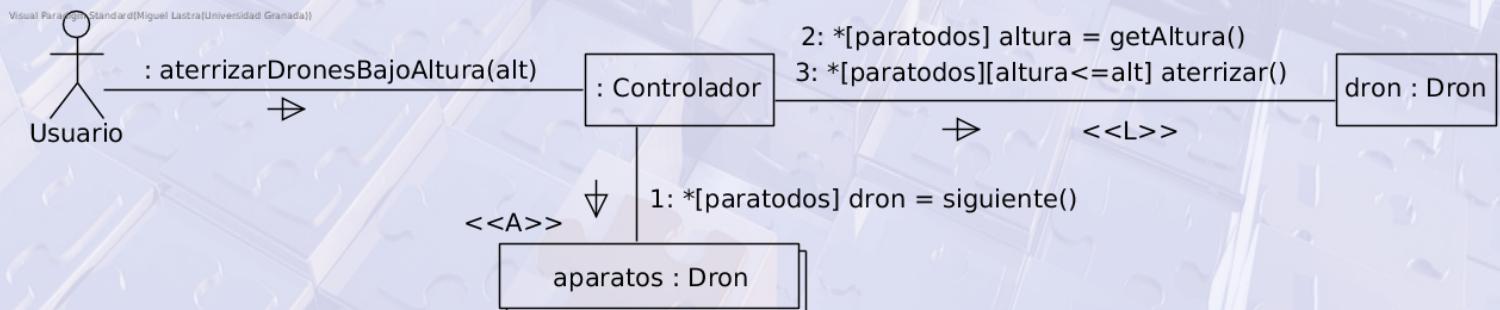


Ejemplo 3



Ejemplo 4

• Condicionales y bucles



Diagramas de interacción

→ **Diseño** ←

- 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)

1.3. Tema 3

1.3.1. Herencia

Herencia

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-147130/>
 - ▶  <https://pixabay.com/images/id-37254/>
- El resto de imágenes e ilustraciones son de creación propia, al igual que los ejemplos de código

Objetivos

- Entender qué significa que una clase *hereda* (o deriva) de otra
 - ▶ Conocer la diferencia entre herencia simple y múltiple
- Comprender la utilidad de la herencia en el diseño de software
- Distinguir cuándo crear clases heredadas (criterios válidos) y cuándo no (criterios no válidos)
- Aprender a crear una clase derivada de otra
 - ▶ Constructor(es)
 - ▶ Redefinición de métodos en las clases derivadas
 - ▶ Uso de la pseudovariable `super`
- Saber las particularidades de Java y Ruby en cuanto a la herencia
- Saber interpretar los diagramas de clases con herencia

Contenidos

1 La relación de herencia entre clases

- Criterios para crear clases derivadas (o superclases)
- Ejemplos

2 Tipos de herencia

3 Redefinición de métodos

4 Particularidades

- Java
- Ruby

5 Diagramas de clases con herencia

6 Anexo: Ejemplos

Introducción al concepto de herencia

- La **herencia** permite **derivar** clases a partir de clases existentes
- ¿Qué significa?
 - ▶ Veamos un pequeño ejemplo:

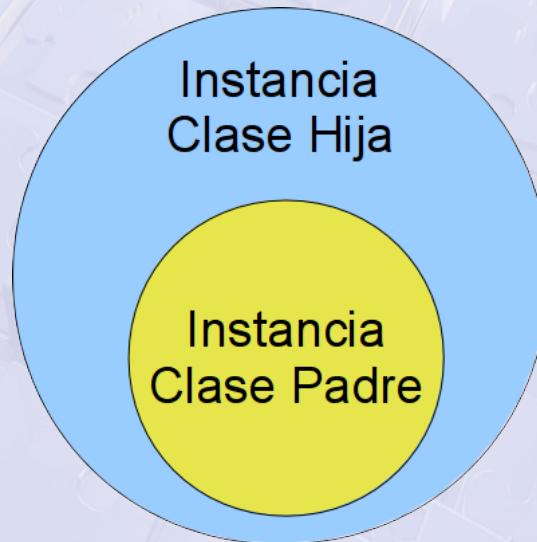


La relación de herencia es una relación **es-un**

- La clase de la que se deriva se denomina:
ancestro, superclase, clase padre, etc.
 - La clase derivada se denomina:
descendiente, subclase, clase hija, etc.
 - La **relación** que se establece es de tipo **es-un**
 - ▶ Un descendiente **es-un** ascendiente
Un lápiz con goma, a todos los efectos, es un lápiz
 - ▶ Donde se espere usar una instancia de una clase, potencialmente, también se podrá emplear una instancia de alguna clase derivada
 - ▶ La relación es-un es **transitiva**
Si C hereda de B y B hereda de A, entonces C hereda de A
- ★ ¿Algún ejemplo de transitividad?

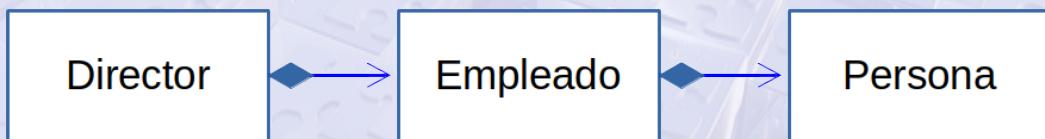
¿Qué se hereda?

- Usualmente la clase hija hereda TODO el código de la clase padre
 - ▶ Cada lenguaje tiene sus particularidades
 - ▶ **No implica** que desde el ámbito de la clase hija se pueda acceder a cualquier elemento del ámbito de la clase padre
 - ▶ El acceso depende de la visibilidad



La herencia como composición (1)

- Consideremos el siguiente diagrama de clases



- ▶ Todas las instancias de Director incluirán una referencia a una instancia de Empleado (similar entre Empleado y Persona)
- ▶ Al construir un Director hay que construir un Empleado
- ▶ Y al construir un Empleado se construirá una Persona

Ruby: Implementación (parcial) del DC anterior

```
class Director
  def initialize (n)
    @empleado = Empleado.new(n)
  end
end
```

```
class Empleado
  def initialize (n)
    @persona = Persona.new(n)
  end
end
```

```
class Persona
  def initialize (n)
    @nombre = n
  end
end
```

La herencia como composición (2)

- (continuación)



- ▶ Si a un Director se le pregunta el nombre, lo normal es que, a su vez, se lo pregunte a la instancia de Empleado que referencia
- ▶ Y lo mismo en el caso de las instancias de Empleado

Ruby: Implementación (parcial) del DC anterior

```
class Director  
  . . .  
  def nombre  
    @empleado.nombre  
  end  
end
```

```
class Empleado  
  . . .  
  def nombre  
    @persona.nombre  
  end  
end
```

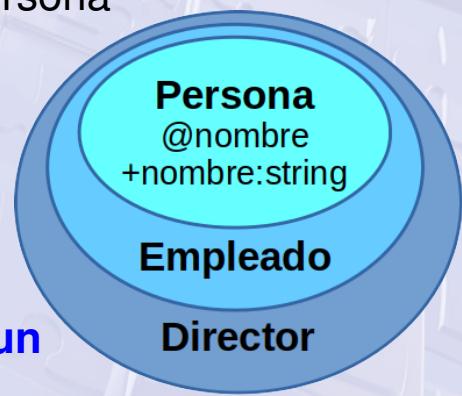
```
class Persona  
  . . .  
  def nombre  
    @nombre  
  end  
end
```

La herencia como composición (3)

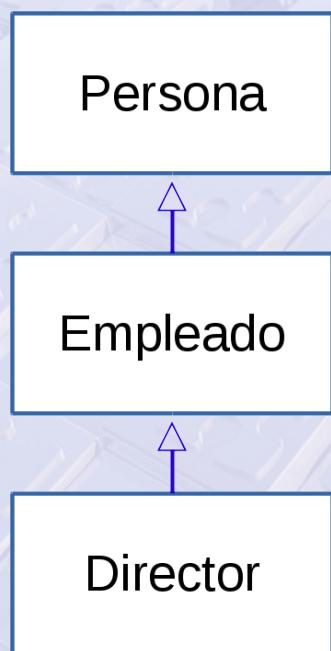
- (continuación)



- ▶ Podría verse como que toda instancia de Director *tiene dentro* una instancia de Empleado y a su vez, toda instancia de Empleado *tiene dentro* una instancia de Persona
- La **herencia** podría verse como
 - ▶ Un **sistema de composición implícita**
 - ▶ **Entre clases que tienen una relación es-un**
 - ▶ Y que es **gestionada de forma automática** por el lenguaje



La herencia como composición (y 4)



Ruby: Implementación mediante herencia

```
1 #encoding : utf -8
2
3 # Todas las clases están COMPLETAS
4 # no se ha omitido ninguna línea
5
6 class Persona
7   def initialize (n)
8     @nombre = n
9   end
10  attr_reader :nombre
11 end
12 class Empleado < Persona
13 end
14 class Director < Empleado
15 end
16
17 el_dire = Director.new("Pedro")
18 puts el_dire.nombre
```

★ ¿Cuál es el resultado de la ejecución?

¿Por qué se usa herencia?

- Reutilización de código, normalmente
 - ▶ La clase derivada añade y/o modifica el comportamiento de la clase padre
- La clase padre puede verse como una generalización de sus descendientes
- Una clase hija puede verse como una especialización de la clase padre



Criterios válidos

- **Especificación**

Las clases hija **implementan** comportamiento declarado (pero no implementado) en el padre

- **Especialización**

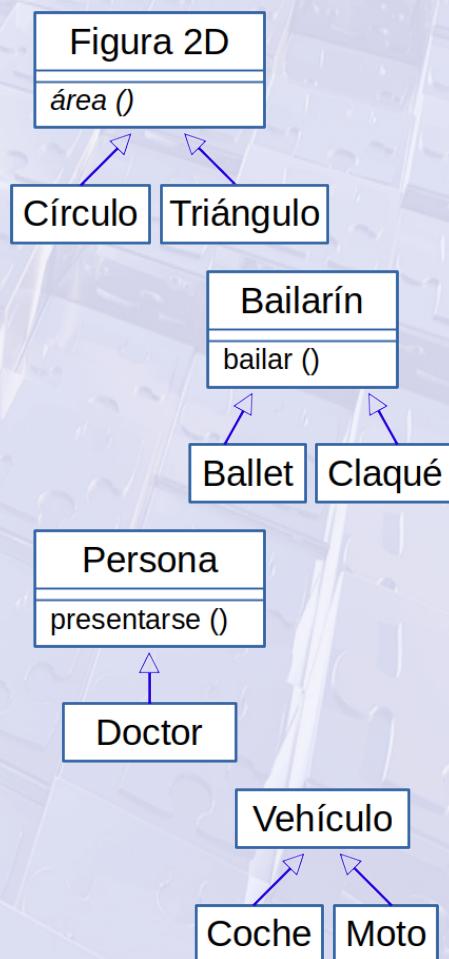
Las clases hijas **modifican** el comportamiento de la clase padre

- **Extensión**

Las clases hijas **amplían** el comportamiento de la clase padre

- **Generalización**

Un ascendiente puede surgir de los **aspectos comunes** entre varias clases



Criterios NO válidos

- Construcción:
Utilizar una clase como base para construir otra
sin que exista una relación es-un entre ellas
- Limitación:
Las clases descendientes **restringen el comportamiento**
especificado por su ascendiente
- **La mera reutilización de código** no puede ser el criterio utilizado
para la utilización de herencia

Ejemplos

Java: Ejemplo de herencia sencillo

```
1 class Persona {  
2     public String andar() {  
3         return ("Ando como una persona");  
4     }  
5  
6     public String hablar() {  
7         return ("Hablo como una persona");  
8     }  
9 }  
10  
11 class Profesor extends Persona {  
12     public String hablar() {  
13         return ("Hablo como un profesor");  
14     }  
15 }  
16  
17 // *****  
18  
19 public static void main(String[] args) {  
20     Profesor profe = new Profesor();  
21     profe.andar(); // Los profesores también andan  
22     profe.hablar();  
23 }
```

Ejemplos

Ruby: Ejemplo de herencia sencillo

```
1 class Persona
2   def andar
3     "Ando como una persona"
4   end
5
6   def hablar
7     "Hablo como una persona"
8   end
9
10 class Profesor < Persona
11   def hablar
12     "Hablo como un profesor"
13   end
14
15   def impartir_clase
16     "Impartiendo clase"
17   end
18
19 puts Persona.new.andar
20 puts Persona.new.hablar
21 puts Persona.new.impartir_clase
22 puts Profesor.new.andar
23 puts Profesor.new.hablar
24 puts Profesor.new.impartir_clase
```

★ ¿Alguna cosa os llama la atención?

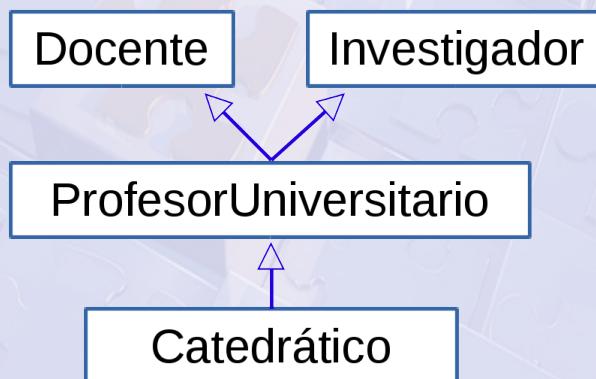
Tipos de herencia

● Simple

- ▶ Cada clase tiene a lo sumo un ascendiente directo
- ▶ Presente en la mayor parte de los lenguajes de programación

● Múltiple

- ▶ Una clase puede tener varios ascendientes directos y heredar de todos ellos
- ▶ La mayor parte de los lenguajes no soportan este tipo de herencia
(se tratará con más detalle en otra lección)



Redefinición de métodos

- Se redefine (sobrescribe) un método cuando una clase proporciona una implementación alternativa a la que ha heredado
 - ▶ La implementación heredada queda anulada
- En general basta con definir un método con la misma signatura (cabecera) que el método que se desea redefinir

Java y Ruby: Repasar los ejemplos anteriores

- ▶ *El método hablar se ha redefinido en Profesor anulando la implementación realizada en Persona*
- ▶ *El método andar no se ha redefinido. Por tanto, en Profesor tiene la misma implementación que en Persona (sin hacer nada explícitamente)*
- Como parte de la redefinición de un método se puede reutilizar el código heredado, extendiendo el mismo

Pseudovariable super

- Cuando se está redefiniendo un método, `super` permite ejecutar la implementación del método proporcionada por la clase padre
- En algunos lenguajes también permite referenciar al constructor de la clase padre

Ruby: Pseudovariable super

```

1 class Persona
2   def hablar
3     "Hablo como una persona"
4   end
5 end
6 class Profesor < Persona
7   def hablar
8     tmp = super
9     tmp += ", y también como un profesor"
10    tmp
11  end
12 end
13 puts Profesor.new.hablar

```

★ ¿Qué produce la línea 13?

super en Java

Acceso a métodos de la clase padre

- Permite acceder a la implementación de cualquier método proporcionado por la clase padre
- **Recomendación:** Usarlo únicamente para acceder al método de la clase padre con el mismo nombre

Java: super en métodos

```
1 int metodo1 (int i, int j) {  
2     int a = super.metodo1 (i, j);      // Uso adecuado  
3     int b = super.metodo2();          // Uso NO recomendado  
4     int c = metodo2();              // Uso recomendado  
5     return (a+b+c);  
6 }
```

- ▶ Si `metodo2` ha sido redefinido en la clase hija, hay que usar esa versión
- ▶ Si no ha sido redefinido, es totalmente innecesario

super en Java

Acceso explícito al constructor de la clase padre

- Permite invocar al constructor de la clase padre. **Debe aparecer en la primera línea del constructor** de la clase derivada
- Si no se invoca expresamente, se invocará implícitamente a un constructor sin parámetros de la clase padre
 - ▶ En ese caso, si no existe dicho constructor, dará error

Java: super en constructor

```

1 class Persona {
2     private String nombre;
3     // Al definir un constructor, deja de existir el constructor por defecto, sin parámetros
4     Persona (String nombre) {
5         this.nombre = nombre;
6     }
7 }
8
9 class Profesor extends Persona {
10    private String asignatura;
11    Profesor (String nomb, String asign) {
12        super (nomb);           // Primera línea. Llamada obligatoria, si no, dará error ¿por qué?
13        asignatura = asign;
14    }
15 }
```

super en Ruby

- Solo permite acceder en la clase padre a la implementación del mismo método que está siendo redefinido
- Si se utiliza sin argumentos se pasan automáticamente los mismos que los recibidos por el método redefinido
- Su uso en el método `initialize` no tiene nada de particular

Ruby: super en métodos

```
1 def metodo1 (i, j)
2   a = super (i, j)
3   # equivalente en este caso a
4   # a = super
5
6   # error salvo que el objeto devuelto por super
7   # tenga un método llamado metodo2
8   # b = super.metodo2
9
10  return 2*a
11 end
```

Llamada a super en initialize

- La responsabilidad de llamar a `super` es nuestra
 - ▶ La llamada no se produce implícitamente
- No hacer la llamada explícita no produce un error por sí mismo
- Aunque puede *inducir* otros errores

Ruby: Ejemplo de initialize sin llamada a super

```

1 class Padre
2   def initialize
3     @padre = "Padre"
4   end
5 end
6
7 class Hija < Padre
8   def initialize
9     @hija = "Hija"
10  end
11
12  def mostrar
13    puts "#{@padre} #{@hija}"
14  end
15 end
16
17 Hija.new.mostrar # ¿Qué ocurre aquí?

```

Llamada a super en initialize

Explicación del ejemplo anterior

- Los atributos no se heredan por se
- Se definen al darles valor en el initialize del padre y ...
- ... el initialize del padre no se ejecuta si no se llama a super

Ruby: Ejemplo de initialize CON llamada a super

```

1 class Padre
2   def initialize
3     @padre = "Padre"
4   end
5 end
6
7 class Hija < Padre
8   def initialize
9     super
10    @hija = "Hija"
11  end
12
13 def mostrar
14   puts "#{@padre} #{@hija}"
15 end
16 end
17
18 Hija.new.mostrar # "Padre Hija"

```

Particularidades de Java

- Todas las clases heredan implícitamente de **Object**
- **No pueden ser redefinidos:**
 - ▶ Los métodos declarados como `final`
 - ▶ Los métodos privados
- Al redefinir un método,
 - ▶ **Es aconsejable** utilizar la anotación `@Override`
 - ★ El compilador avisa si se está sobrecargando en vez de redefiniendo
 - ▶ **Se permiten los siguientes cambios en la cabecera:**
 - ★ Mayor accesibilidad en cuanto al especificador de acceso.
Por ejemplo: cambiar algo de `protected` a `public`
 - ★ Tipo covariante en el tipo del valor retornado.
Puede ser una subclase del indicado en el método del ancestro

Java: Anotación `@Override`

```
1  @Override      // Cada método redefinido debe llevarla
2  int metodo (int parametro) { . . . }
```

Particularidades de Ruby

- Todas las clases heredan implícitamente de `Object`
- Al crear un método con el mismo nombre que en la superclase se produce la redefinición (independientemente de los parámetros)
 - ▶ Debido a que Ruby no admite sobrecarga

Ruby: Redefinición de métodos

```
1 class Persona
2   def andar
3     "Ando como una persona"
4   end
5
6   def hablar
7     "Hablo como una persona"
8   end
9
10 class Profesor < Persona
11   def andar          # Redefinición
12     "Ando como un profesor"
13   end
14
15   def hablar (txt)  # También redefinición, no sobrecarga
16     "Estoy diciendo: " + txt
17   end
18
19 end
```

Diagrama de clases con herencia

- En la clase derivada se incluyen:
 - ▶ Los atributos y métodos añadidos
 - ▶ Los métodos redefinidos

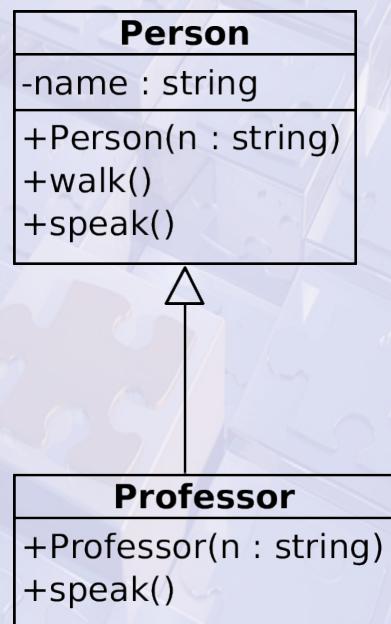
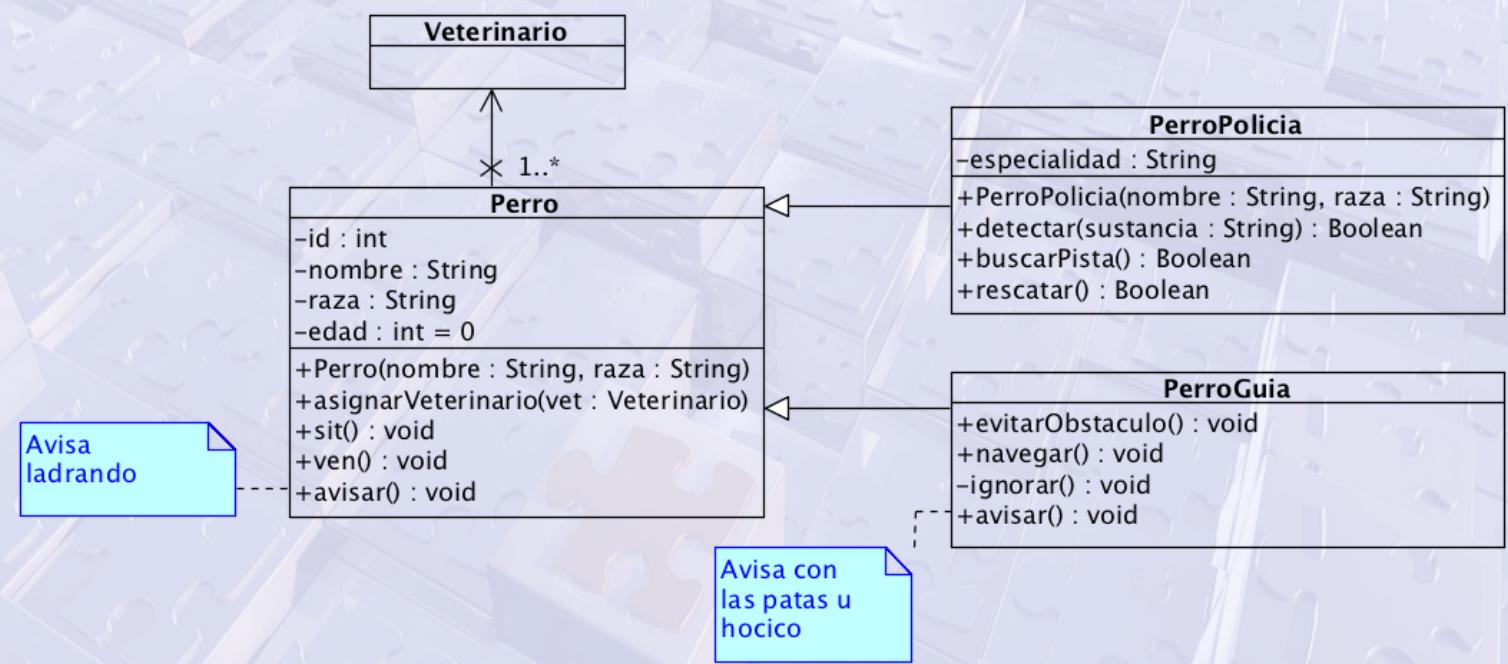


Diagrama de clases con herencia



- *Este módulo al completo podía llevar la etiqueta → **Diseño** ←*
- *En particular, prestar especial atención a los conceptos explicados en las páginas:*
 - ▶ Relación es-un
 - ▶ Herencia como composición
 - ▶ Criterios válidos para crear clases derivadas (o superclases)
 - ▶ Redefinición de métodos
 - ▶ Pseudovariable super

Anexo: Ejemplos

- Diversos ejemplos
- Planteároslos como ejercicios

Ejemplo en Ruby

Ruby: Uso de `super`, ausencia de `initialize`

```
1 class Persona
2   def initialize(n)
3     @nombre = n
4   end
5
6   def andar
7     "Ando como una persona"
8   end
9
10  def hablar
11    "Hablo como una persona"
12  end
13 end
14
15 class Profesor < Persona
16   def hablar
17     tmp = "Estimados alumnos: \n"
18     tmp += "Me llamo #{@nombre}\n" # ¿ En qué momento ha tomado valor @nombre ?
19     tmp += super
20     tmp
21   end
22 end
23
24 puts Profesor.new("Jaime").hablar # Si Profesor no tiene initialize , ¿qué va a ocurrir?
```

Ejemplo en Ruby

Ruby: Ausencia de initialize

```
1 class A
2   def initialize(a)
3     puts "Creando A"
4     @a = a
5   end
6 end
7
8 class B < A
9 end
10
11 A.new(77)
12 B.new
13 B.new(88)
```

★ Una de las 3 últimas líneas es errónea

★ ¿Cuál? ¿por qué?

Ejemplo en Ruby

Ruby: Redefiniendo initialize

```
1 class C
2   def initialize(c)
3     puts "Creando C"
4     @c = c
5   end
6 end
7
8 class D < C
9   def initialize
10    puts "Creando D"
11    @d = 88
12  end
13 end
14
15 C.new(99)
16 d = D.new
17 puts d.inspect
```

★ ¿Qué ocurre en la línea 16?

★ ¿Cuál es el resultado de la línea 17?

Ejemplo en Ruby

Ruby: Redefiniendo initialize, super en initialize

```
1 class E
2   def initialize(e)
3     puts "Creando E"
4     @e = e
5   end
6 end
7
8 class F < E
9   def initialize
10    puts "Creando F"
11    @f = 88
12    super(99) # Se llama al initialize del padre explícitamente
13      # No tiene por qué ser la primera línea
14  end
15 end
16
17 E.new(99)
18 f = F.new
19 puts f.inspect
```

- ★ ¿Qué ocurre en la línea 18?
- ★ ¿Cuál es el resultado de la línea 19?

Ejemplo en Ruby

Ruby: Redefiniendo initialize, sin parámetros

```
1 class G
2   def initialize
3     puts "Creando G"
4     @g = 66
5   end
6 end
7
8 class H < G
9   def initialize
10    puts "Creando H"
11    @h = 88
12  end
13 end
14
15 G.new
16 h = H.new
17 puts h.inspect
```

★ ¿Qué ocurre en la línea 16?

★ ¿Cuál es el resultado de la línea 17?

Herencia

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)

1.3.2. Visibilidad

Especificadores de acceso (Visibilidad)

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

- Entender el propósito de los especificadores de acceso
- Comprender cómo afectan los especificadores de acceso a métodos y atributos
- Saber usarlos en Java y Ruby
 - ▶ (y no confundirse con las diferencias existentes en cada lenguaje)

Contenidos

- 1 Propósito de los especificadores de acceso
- 2 Especificadores de acceso en Java
- 3 Especificadores de acceso en Ruby
- 4 Anexo: Ejemplos
 - Ejemplos en Java
 - Ejemplos en Ruby

Propósito de los especificadores de acceso

- Permiten restringir el acceso a atributos y métodos
- Ocultan detalles de la implementación para que los objetos sean usados a través de una interfaz concreta
- Suele ser aconsejable usar el nivel más restrictivo posible
→ *Diseño* ←
- Dependiendo del lenguaje también pueden ser aplicados a otros elementos como las clases

Especificadores de acceso habituales

- Los especificadores de acceso habituales son:
 - ▶ Privado
 - ★ Puede ser accedido desde la propia clase
 - ▶ Protegido
 - ★ Puede ser accedido desde la propia clase y sus clases derivadas
 - ▶ Público
 - ★ Puede ser accedido desde cualquier sitio
- Según el lenguaje pueden existir otros, por ejemplo:
 - ▶ Java añade un especificador más: Paquete
 - ▶ Smalltalk solo tiene Público y Protegido
- **Atención: Hay diferencias importantes en su significado dependiendo del lenguaje**

Especificadores de acceso en Java

- Permite establecerlos a atributos y métodos
 - ▶ Cada elemento debe incluir el suyo
- Particularidades del especificador **private**
 - ▶ Solo es **accesible desde código de la propia clase** (ya sea desde **ámbito de instancia o de clase**)
 - ★ Desde el **ámbito de instancia** se puede acceder a elementos de clase privados de la misma clase
 - ▶ Se puede acceder a elementos privados de otra **instancia distinta** si es **de la misma clase** (tanto desde **ámbito de instancia** como de **clase**)
 - ★ Esa otra instancia distinta ha podido recibirse como parámetro en un método (de instancia o de clase)

Privado en Java

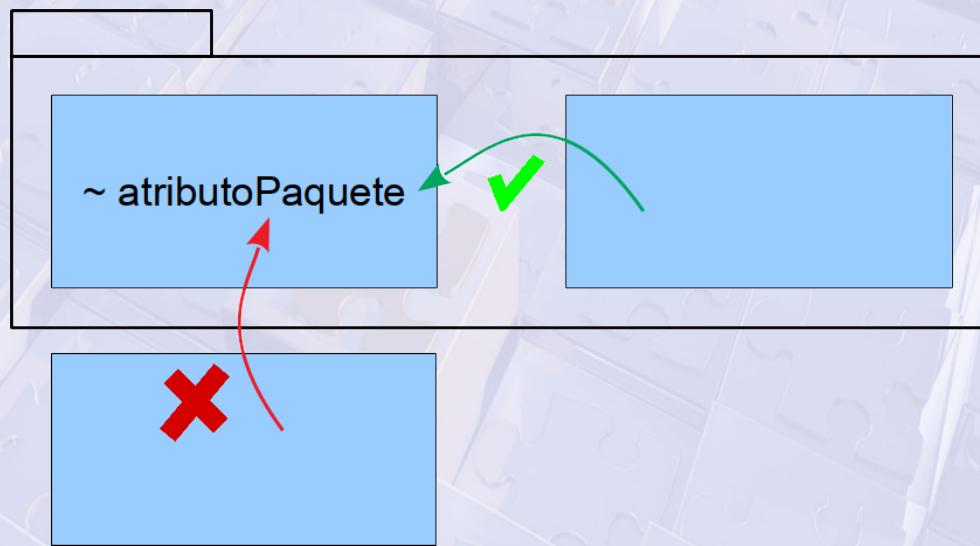
Ejemplo

Java: Especificador de acceso privado

```
1 class Persona {  
2     static private String especie = "humana";  
3  
4     private String nombre;  
5  
6     Persona (String n) {  
7         nombre = n;      // Los atributos privados son accesibles desde la propia clase  
8     }  
9  
10    Persona (Persona p) {  
11        nombre = p.nombre; // En Java se pueden acceder a atributos privados de otros objetos  
12                           // siempre que sean instancias de la MISMA clase  
13    }  
14  
15    String toString() {  
16        return "La persona " + nombre + " pertenece a la especie " + especie;  
17        // También se puede acceder a atributos privados de clase  
18    }  
19 }
```

Especificadores de acceso en Java

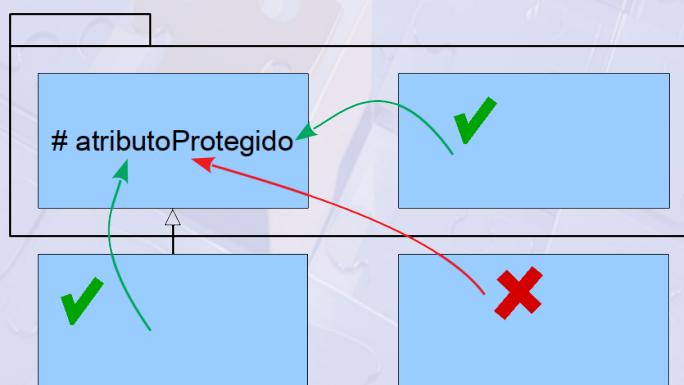
- Particularidades del especificador *de paquete*
 - No poner ningún especificador significa visibilidad de paquete
 - Estos elementos son **públicos** dentro del paquete
 - y **privados** respecto al exterior del paquete



Especificadores de acceso en Java

● Particularidades del especificador **protected**

- ▶ Estos elementos son **públicos dentro del mismo paquete**
 - ★ Son accesibles desde el mismo paquete
(con independencia de la relación de herencia que exista (o no) entre las clases involucradas)
- ▶ También son **accesibles desde subclases de otros paquetes**
 - ★ Dentro de una misma instancia, se podrá acceder a elementos protegidos definidos en cualquiera de sus superclases
(con independencia del paquete en el que estén las clases involucradas)



Especificadores de acceso en Java

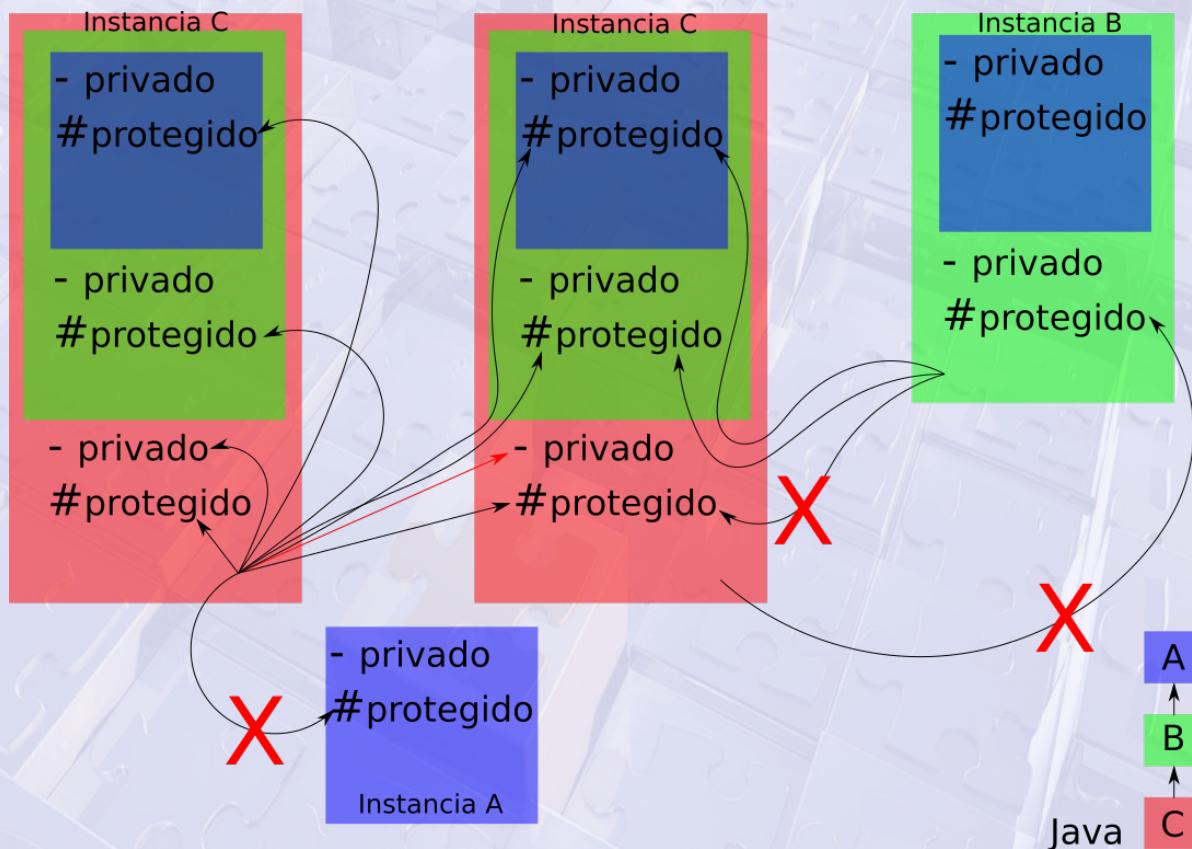
- Particularidades del especificador **protected** (continuación)
- Para poder acceder a
elementos protegidos de una instancia distinta:
 - (tanto desde ámbito de clase como de instancia)
 - ▶ Esa instancia tiene que ser de la misma clase que la propietaria del código desde el que se realiza el acceso o de una subclase de la misma
 - ★ Es decir, **esa instancia debe ser un yo**
 - ▶ El elemento accedido tiene que estar declarado en la clase propietaria del código desde el que se realiza el acceso o en una superclase de la misma
 - ★ Es decir, **el elemento debe ser visible por mí**
 - ▶ **Recordar:** Si las clases involucradas están en el mismo paquete, los elementos protegidos son accesibles siempre

(lo acabamos de ver en la página anterior)

Especificadores de acceso de las clases Java

- Las propias clases Java podrán ser:
 - ▶ Públicas: `public`
Son utilizables desde cualquier sitio
 - ▶ De paquete: *no se indica ningún especificador de acceso*
Son solo utilizables dentro del paquete en las que se definen

Especificadores de acceso en Java: Resumen



Especificadores de acceso en Ruby

- Los atributos son **siempre privados**. No se puede cambiar
- Los métodos son por defecto públicos, aunque esto se puede modificar mediante especificadores de acceso
- El método `initialize` es **siempre privado**. No se puede cambiar
- Cuando se utiliza un especificador de acceso, este afecta a todos los elementos declarados a continuación

Especificadores de acceso en Ruby

- Particularidades del especificador **private**
 - ▶ Un método privado nunca puede ser utilizado mediante un receptor de mensaje explícito
 - ★ **A partir de Ruby 2.7** (diciembre 2019)
sí se permite `self` como receptor de mensaje explícito
 - ▶ Solo se puede utilizar un método privado de la propia instancia
 - ▶ Si B hereda de A
 - ★ Desde un ámbito de instancia de B se puede llamar a métodos de instancia privados de A
 - ★ Desde un ámbito de clase de B se puede llamar a métodos de clase privados de A
 - ▶ No se puede acceder a métodos privados de clase desde el ámbito de instancia
 - ▶ No se puede acceder a métodos privados de instancia desde el ámbito de clase

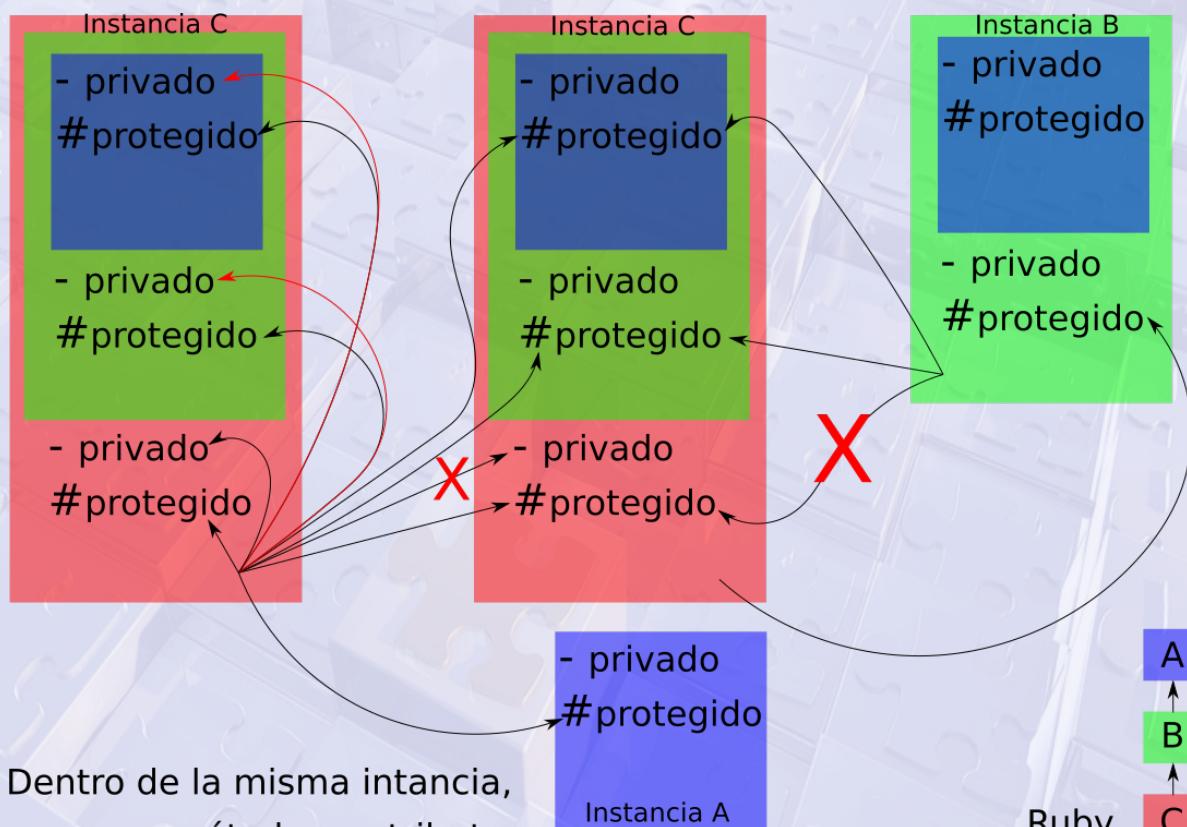
Especificadores de acceso en Ruby

- Particularidades del especificador **protected**
 - ▶ La clase del código que invoca debe ser la misma, o una subclase, de la clase donde se definió dicho método
 - ▶ No existen métodos protegidos de clase

Especificadores de acceso en Ruby

- En general, recordar que en Ruby las clases también son objetos a todos los efectos
 - ▶ Una clase y sus instancias **no son de la misma clase**
 - ▶ Como objetos que son, son instancias de clases distintas
- Los atributos de clase (`@@atributo_de_clase`) sí pueden ser accedidos directamente desde el ámbito de instancia

Especificadores de acceso en Ruby: Resumen



Especificadores de acceso

- Se aconseja usar el nivel más restrictivo posible
- Un atributo privado con un consultor público que devuelve una referencia puede ser modificado “desde fuera”
 - Como ya se comentó en el módulo de consultores y modificadores:
 - ▶ Hay que evaluar cada caso y decidir qué se devuelve (o asigna)
 - ★ Una referencia o una copia
 - ★ Dependiendo de a quién se le dé, o de dónde venga

Anexo: Ejemplos

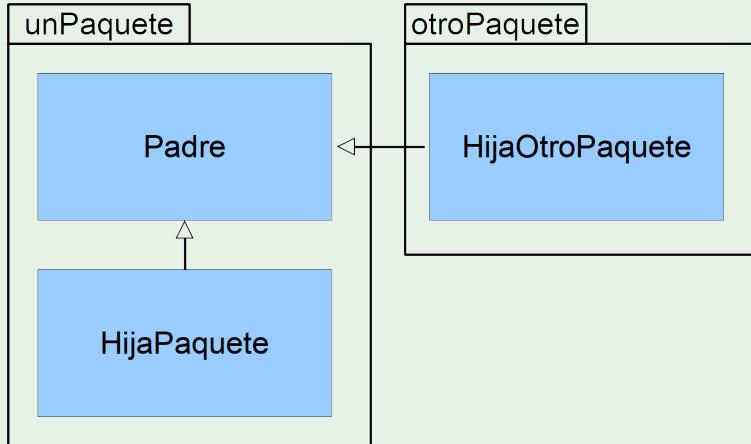
- Diversos ejemplos
- Planteároslos como ejercicios

Especificadores de acceso en Java: Ejemplos

Java: Acceso a elementos de otra instancia de la misma clase

```

1 package unPaquete;
2
3 public class Padre {
4     private int privado;
5     protected int protegido;
6     int paquete;
7     public int publico;
8
9     public void testInstanciaPadre (Padre o) {
10        System.out.println (o.privado);
11        System.out.println (o.protegido);
12        System.out.println (o.paquete);
13        System.out.println (o.publico);
14    }
15
16    public static void testClasePadre (Padre o) {
17        System.out.println (o.privado);
18        System.out.println (o.protegido);
19        System.out.println (o.paquete);
20        System.out.println (o.publico);
21    }
22 }
```



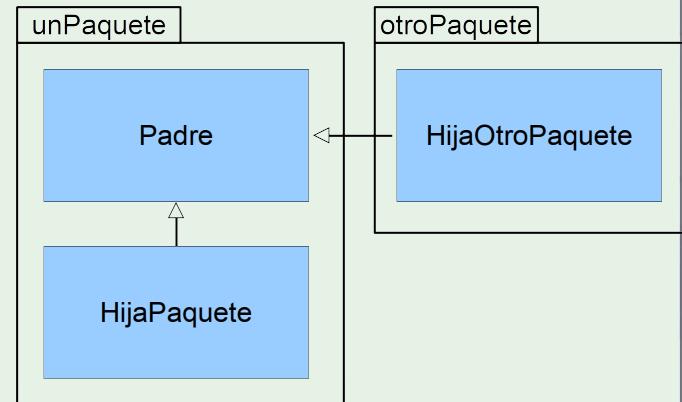
★ ¿Qué ocurre en cada línea? ¿Algún error?

Especificadores de acceso en Java: Ejemplos

Java: Acceso a instancia de la superclase desde el mismo paquete

```

1 package unPaquete;
2
3 public class HijaPaquete extends Padre{
4
5     public void testInstanciaHijaPaquete (Padre o) {
6         System.out.println (privado);
7         System.out.println (o.privado);
8
9         System.out.println (protigido);
10        System.out.println (o.protigido);
11
12        System.out.println (o.paquete);
13        System.out.println (o.publico);
14    }
15
16    public static void testClaseHijaPaquete (Padre o) {
17        System.out.println (o.privado);
18
19        System.out.println (o.protigido);
20
21        System.out.println (o.paquete);
22        System.out.println (o.publico);
23    }
24 }
```



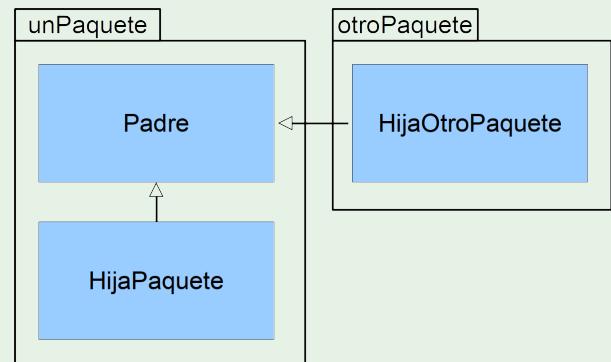
★ ¿Qué ocurre en cada línea? ¿Algún error?

Especificadores de acceso en Java: Ejemplos

Java: Acceso a instancia de la superclase desde otro paquete

```

1 package otroPaquete;
2
3 public class HijaOtroPaquete extends Padre {
4
5     public void testInstanciaHijaOtroPaquete (Padre o){
6         // Acceso a elementos heredados
7         System.out.println (privado);
8         System.out.println (paquete);
9         System.out.println (protegido);
10
11        // Acceso a elementos de otra instancia
12        System.out.println (o.privado);
13        System.out.println (o.protegido);
14        System.out.println (o.paquete);
15        System.out.println (o.publico);
16    }
17
18    public static void testClaseHijaOtroPaquete (Padre o){
19        // Acceso a elementos de otra instancia
20        System.out.println (o.privado);
21        System.out.println (o.protegido);
22        System.out.println (o.paquete);
23        System.out.println (o.publico);
24    }
25 }
```



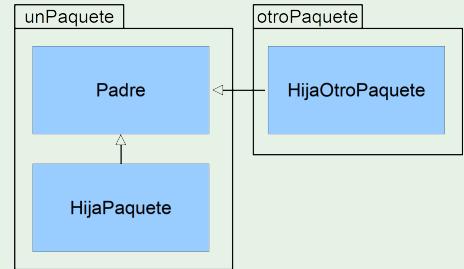
★ ¿Qué ocurre en cada línea? ¿Algún error?

Especificadores de acceso en Java: Ejemplos

Java: Acceso a instancia de la misma clase,
los elementos heredados se declaran en otro paquete

```

1 package otroPaquete;
2
3 public class HijaOtroPaquete extends Padre {
4
5     // El mismo código cambiando solo el tipo del parámetro
6
7     public void testInstanciaHijaOtroPaquete (HijaOtroPaquete o) {
8         System.out.println(o.privado);
9         System.out.println(o.protegido);
10        System.out.println(o.paquete);
11        System.out.println(o.publico);
12    }
13
14    public static void testClaseHijaOtroPaquete (HijaOtroPaquete o) {
15        System.out.println(o.privado);
16        System.out.println(o.protegido);
17        System.out.println(o.paquete);
18        System.out.println(o.publico);
19    }
20 }
```



★ ¿Qué ocurre en cada línea? ¿Algún error?

Especificadores de acceso en Java: Ejemplos

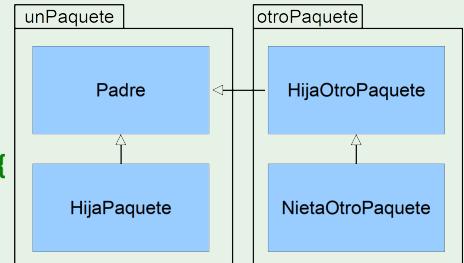
Java: Acceso a instancia de subclase

los elementos heredados se declaran en otro paquete

```

1 package otroPaquete;
2
3 public class HijaOtroPaquete extends Padre {
4
5     // El mismo código. El tipo del parámetro es subclase.
6
7     public void testInstanciaHijaOtroPaquete (NietaOtroPaquete o) {
8         System.out.println(o.privado);
9         System.out.println(o.protegido);
10        System.out.println(o.paquete);
11        System.out.println(o.publico);
12    }
13
14    public static void testClaseHijaOtroPaquete (NietaOtroPaquete o) {
15        System.out.println(o.privado);
16        System.out.println(o.protegido);
17        System.out.println(o.paquete);
18        System.out.println(o.publico);
19    }
20}
21
22 // NietaOtroPaquete deriva de HijaOtroPaquete (ambas están en otroPaquete)

```



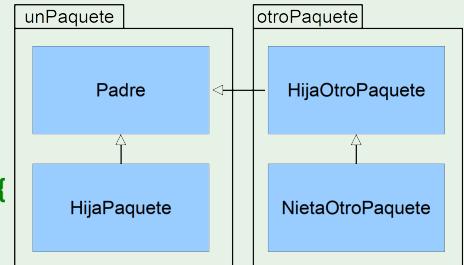
★ ¿Qué ocurre en cada línea? ¿Algún error?

Especificadores de acceso en Java: Ejemplos

Java: Acceso a instancia de la superclase
los elementos heredados se declaran en otro paquete

```

1 package otroPaquete;
2
3 public class NietaOtroPaquete extends HijaOtroPaquete {
4
5     // Ahora probamos con un parámetro de la superclase
6
7     public void testInstanciaNietaOtroPaquete (HijaOtroPaquete o) {
8         System.out.println (o.privado);
9         System.out.println (o.protegido);
10        System.out.println (o.paquete);
11        System.out.println (o.publico);
12    }
13
14    public static void testClaseNietaOtroPaquete (HijaOtroPaquete o) {
15        System.out.println (o.privado);
16        System.out.println (o.protegido);
17        System.out.println (o.paquete);
18        System.out.println (o.publico);
19    }
20 }
```



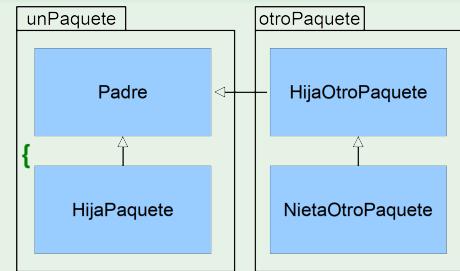
★ ¿Qué ocurre en cada línea? ¿Algún error?

Especificadores de acceso en Java: Ejemplos

Java: Acceso a instancia de la superclase
los elementos heredados se declaran en otro paquete

```

1 package otroPaquete;
2
3 public class NietaOtroPaquete extends HijaOtroPaquete{
4
5     public void testInstanciaNietaOtroPaquete (NietaOtroPaquete o) {
6         System.out.println (o.privado);
7         System.out.println (o.protegido);
8         System.out.println (o.paquete);
9         System.out.println (o.publico);
10    }
11
12    public static void testClaseNietaOtroPaquete (NietaOtroPaquete o) {
13        System.out.println (o.privado);
14        System.out.println (o.protegido);
15        System.out.println (o.paquete);
16        System.out.println (o.publico);
17    }
18 }
```



★ ¿Qué ocurre en cada línea? ¿Algún error?

Especificadores de acceso en Java: Ejemplos

Java: Acceso a protegidos

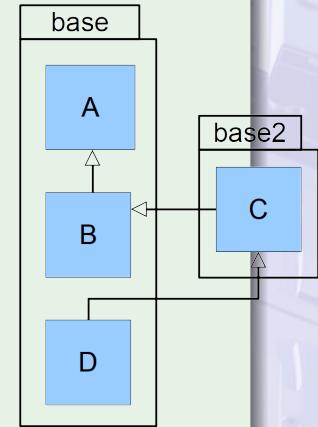
```

1 package base;
2
3 public class A {
4     protected int protegidoA = 0;
5 }
6
7 // ****
8
9 public class B extends A{
10     protected int protegidoB = 1;
11 }
12
13 // ****
14
15 import base2.C;
16
17 public class D extends C {
18     protected int protegidoD = 3;
19 }
20 }
```

Java: Acceso a protegidos

```

1 package base2;
2 import base.*;
3
4 public class C extends B{
5     protected int protegidoC = 2;
6
7     public void test() {
8         A a = new A();
9         a.protegidoA = 666;
10
11         B b = new B();
12         b.protegidoB = 666;
13
14         C c = new C();
15         c.protegidoA = 555;
16
17         D d = new D();
18         d.protegidoA = 555;
19
20         D d2 = new D();
21         d2.protegidoB = 555;
22         d2.protegidoD = 555;
23
24         this.protegidoA = 777;
25     }
26 }
```



★ ¿Qué ocurre en cada línea? ¿Algún error?

Especificadores de acceso en Ruby: Ejemplos

Ruby: Acceso a privados y protegidos

```
1  class Padre
2    private
3    def privado
4    end
5
6    protected
7    def protegido
8    end
9
10   public
11   def publico
12   end
13
14  def test(p)
15    privado
16    self.privado # Correcto solo a partir de Ruby 2.7
17    p.privado
18    protegido
19    self.protegido
20    p.protegido
21  end
22 end
```

★ ¿Qué ocurre en cada línea? ¿Algún error?

Especificadores de acceso en Ruby: Ejemplos

Ruby: Acceso a privados y protegidos

```
1 # Fuera de cualquier clase
2
3 Padre.new.test(Padre.new)
4 p=Padre.new
5
6 # Acceso a métodos fuera de la clase o subclases
7
8 p.privado
9 p.protegido
10 p.publico
```

★ ¿Qué ocurre en cada línea? ¿Algún error?

Especificadores de acceso en Ruby: Ejemplos

Ruby: Acceso a privados y protegidos con relación de herencia

```
1  class Hija < Padre
2    def test(p)
3      privado
4      self.privado # Correcto solo a partir de Ruby 2.7
5      p.privado
6      protegido
7      self.protegido
8      p.protegido
9      publico
10     self.publico
11     p.publico
12   end
13 end
14
15 # Fuera de cualquier clase
16
17 Hija.new.test(Hija.new)
18 Hija.new.test(Padre.new)
19 h=Hija.new
20 h.privado
21 h.protegido
22 h.publico
```

★ ¿Qué ocurre en cada línea? ¿Algún error?

Especificadores de acceso en Ruby: Ejemplos

Ruby: Métodos privados de instancia y de clase

```
1  class Padre
2    private
3    def privado_instancia
4    end
5
6    def self.privado_clase # Por ahora este método es público
7  end
8
9  private_class_method :privado_clase # Atención a la sintaxis
10
11 public
12 def test
13   self.class.privado_clase
14 end
15
16 def self.test(p)
17   p.privado_instancia
18 end
19
20 end
21 # Fuera de cualquier clase
22
23 Padre.new.test
24 Padre.test(Padre.new)
```

★ ¿Qué ocurre en cada línea? ¿Algún error?

Especificadores de acceso en Ruby: Ejemplos

Ruby: Variables de clase y de instancia de la clase

```
1 class Padre
2   @instanciaDeClase = 1
3   @duda = 2
4   @@deClase = 11
5   @@duda = 22
6
7   def initialize
8     @deInstancia = 333
9     @duda = 444
10  end
11
12  def self.salida
13    puts @instanciaDeClase+1
14    puts @duda+1 unless @duda.nil? # desde Hija?
15    puts @@deClase+1
16    puts @@duda+1
17  end
18
19  def salida
20    puts @deInstancia+1
21    puts @duda+1
22    puts @@deClase+1
23    puts @@duda+1
24  end
25 end
```

★ ¿Qué ocurre en cada línea? ¿Algún error?

Especificadores de acceso en Ruby: Ejemplos

Ruby: Variables de clase y de instancia de la clase

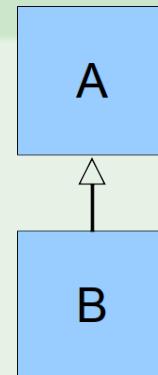
```
1 class Hija < Padre
2   @instanciaDeClase = -1
3
4   # Sobreescribimos el valor fijado anteriormente
5   # Este atributo es compartido
6   @@deClase = -11
7
8   def modifica
9     # Acceso a los atributos definidos en Padre
10    @duda += 111
11  end
12 end
13
14 # Fuera de cualquier clase
15
16 Padre.salida
17 Hija.salida # Atención a lo que ocurre con la segunda línea
18
19 p = Padre.new
20 p.salida
21 h = Hija.new
22 h.salida
23
24 h.modifica
25 h.salida
```

★ ¿Qué ocurre en cada línea? ¿Algún error?

Especificadores de acceso en Ruby: Ejemplos

Ruby: Relaciones de varias clases en cadena

```
1 class A
2   protected
3   def protegidoA
4   end
5 end
6
7 class B < A
8   protected
9   def protegidoB
10  end
11 end
```



★ ¿Qué ocurre en cada línea? ¿Algún error?

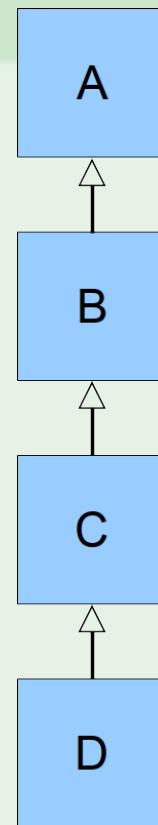
Especificadores de acceso en Ruby: Ejemplos

Ruby: Relaciones de varias clases en cadena

```

1  class C < B
2    protected
3      def protegidoC
4    end
5
6    public
7    def test
8      A.new.protegidoA
9      B.new.protegidoA
10     B.new.protegidoB
11     C.new.protegidoA
12     C.new.protegidoB
13     C.new.protegidoC
14     D.new.protegidoA
15     D.new.protegidoD
16   end
17 end
18
19 class D < C
20   protected
21   def protegidoD
22 end
23
24 C.new.test
25 C.new.protegidoC

```



★ ¿Qué ocurre en cada línea? ¿Algún error?

Especificadores de acceso en Ruby: Ejemplos

Ruby: Falsa seguridad

```
1 class FalsaSeguridad
2   # Un consultor puede ser muy peligroso
3   attr_reader :lista
4
5   def initialize
6     @lista = [1,2,3,4]
7   end
8
9   def info
10    puts @lista.size
11  end
12
13 end
14
15 # Fuera de cualquier clase
16 f = FalsaSeguridad.new
17 f.info #4
18
19 # Modificamos el estado interno
20 # Simplemente usando un consultor
21 # Aunque el atributo sea privado
22 # Cuidado con las referencias
23 f.lista.clear
24
25 f.info #0
```

★ ¿Qué ocurre en cada línea? ¿Algún error?

Especificadores de acceso (Visibilidad)

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)

1.3.3. Polimorfismo

Polimorfismo y Ligadura Dinámica

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

- Entender los conceptos polimorfismo y ligadura dinámica
- Saber usar dichos mecanismos
- Saber detectar situaciones en las que es procedente el uso de dichos mecanismos
- Saber realizar diseños para dar solución a dichas situaciones

Contenidos

1 Introducción

2 Polimorfismo

3 Ligadura dinámica

- Casts
- Comprobaciones explícitas de tipos
- Detalles adicionales

Introducción

Java: Introducción a polimorfismo y ligadura dinámica

```
1 class Empleado { // entre otras cosas ...  
2     float calculaSueldo() { return . . . }  
3 }  
4 class Comercial extends Empleado { // entre otras cosas ...  
5     float calculaSueldo() {  
6         return super.calculaSueldo() + . . .  
7     }  
8 }  
9 class Directivo extends Empleado { // entre otras cosas ...  
10    float calculaSueldo() {  
11        return super.calculaSueldo() + . . .  
12    }  
13 }  
14  
15 // En algún otro sitio ...  
16 ArrayList<Empleado> listaDeEmpleados = new ArrayList<>();  
17 listaDeEmpleados.add (new Empleado (. . .));  
18 listaDeEmpleados.add (new Comercial (. . .));  
19 listaDeEmpleados.add (new Directivo (. . .));  
20  
21 // Se quiere calcular el total de todos los sueldos  
22 float sueldosTotales = 0.0f;  
23 for (Empleado unEmpleado : listaDeEmpleados) {  
24     sueldosTotales += unEmpleado.calculaSueldo();  
25 }
```

Polimorfismo

- Capacidad de un identificador de **referenciar objetos de diferentes tipos** (clases)
 - ▶ En lenguajes sin declaración de variables se da de forma natural y sin limitaciones
 - ▶ Ruby no utiliza el mecanismo de declaración de variables. Cualquier variable puede referenciar cualquier tipo de objeto
 - ▶ En lenguajes con declaración de variables con un tipo específico existen limitaciones al respecto
- **Principio de sustitución de Liskov:**
 - ▶ Si B es un subtipo de A, se pueden utilizar instancias de B donde se esperan instancias de A
 - ▶ Por ejemplo:
 - ★ Si `Director` es subclase de `Persona` se puede usar una instancia de `Director` donde se puedan usar instancias de `Persona`.
 - ★ Recordar la relación **es-un**:
`Director` es-una `Persona` (a todos los efectos)

Tipo estático y dinámico

- **Tipo estático:** tipo (clase) del que se declara la variable
- **Tipo dinámico:** clase al que pertenece, **en un momento determinado**, el objeto referenciado por una variable

Java: Tipo estático y dinámico

```
1 ArrayList<Empleado> listaDeEmpleados = new ArrayList<>();
2 listaDeEmpleados.add (new Empleado (...));
3 listaDeEmpleados.add (new Comercial (...));
4 listaDeEmpleados.add (new Directivo (...));
5
6 // Se quiere calcular el total de todos los sueldos
7 float sueldosTotales = 0.0f;
8 for (Empleado unEmpleado : listaDeEmpleados) {
9     sueldosTotales += unEmpleado.calculaSueldo ();
10 }
```

★ ¿Cuál es el tipo estático de `unEmpleado`?

★ ¿Y su tipo dinámico?

★ ¿Se puede saber con solo mirar el código?

Ligadura dinámica

- **Ligadura estática:**

El enlace del código a ejecutar asociado a una llamada a un método se hace en tiempo de compilación (permitida en C++)

- **Ligadura dinámica:**

El tipo dinámico determina el código que se ejecutará asociado a la llamada de un método

- ▶ Hace que cobre sentido el polimorfismo

Java: Ligadura dinámica

```

1 class Empleado { // entre otras cosas ...
2     float calculaSueldo() { return . . . }
3 }
4 class Comercial extends Empleado { // entre otras cosas ...
5     float calculaSueldo() { return super.calculaSueldo() + . . . }
6 }
7 class Directivo extends Empleado { // entre otras cosas ...
8     float calculaSueldo() { return super.calculaSueldo() + . . . }
9 }
10
11 // En algún otro sitio . . .
12 for (Empleado unEmpleado : listaDeEmpleados) {
13     sueldosTotales += unEmpleado.calculaSueldo();
14 }                                // ¿Qué implementación de calculaSueldo se va a ejecutar?

```

Ejemplo

Java: Ejemplo de polimorfismo y ligadura dinámica

```
1 class Persona {  
2     public String andar() {  
3         return ("Ando como una persona");  
4     }  
5  
6     public String hablar() {  
7         return ("Hablo como una persona");  
8     }  
9 }  
10  
11 class Profesor extends Persona{  
12     @Override  
13     public String hablar() {  
14         return ("Hablo como un profesor");  
15     }  
16 }  
17 // *****  
18  
19 public static void main(String[] args) {  
20     Persona p=new Persona();  
21     Persona p2=new Profesor(); // Puede también referenciar un Profesor  
22  
23     p.hablar(); // "Hablo como una persona"  
24     p2.hablar(); // "Hablo como un profesor"  
25 }
```

Reglas

- El tipo estático limita:
 - ▶ Lo que puede referenciar una variable
 - ★ Instancias de la clase del tipo estático o de sus subclases
 - ▶ Los métodos que pueden ser invocados
 - ★ Los disponibles en las instancias de la clase del tipo estático

Java: Ejemplo

```

1 class Profesor extends Persona{
2
3     public String impartirClase() {
4         // Este método no lo tiene Persona ni ninguna de sus superclases
5         return ("Impartiendo clase");
6     }
7 }
8 // ****
9
10 public static void main(String[] args) {
11     Persona p=new Profesor();
12
13     // Error de compilación. Las personas no tienen ese método
14     p.impartirClase();
15
16     // Error de compilación. Object no es subclase de Persona
17     p=new Object();
18 }
```

Casts

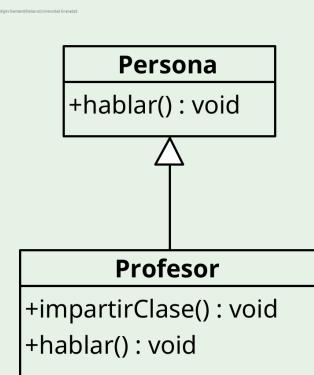
- Se le indica al compilador que considere, **temporalmente**, que el tipo de una variable es otro
 - ▶ Solo para la instrucción en la que aparece y **con limitaciones**
- **Downcasting:**
 - ▶ Se indica al compilador que considere, temporalmente, que el tipo de la variable es una subclase del tipo con que se declaró
 - ▶ Permite invocar métodos que sí existen en el tipo del cast pero que no están en el tipo estático de la variable
- **Upcasting:**
 - ▶ Se indica al compilador que considere, temporalmente, que el tipo de la variable es superclase del tipo con que se declaró
 - ▶ Normalmente es innecesario y redundante
- **Importante:**
 - ▶ Las operaciones de casting no realizan ninguna transformación en el objeto referenciado
 - ▶ Tampoco cambian el comportamiento del objeto referenciado

Ejemplo

Java: Ejemplo de casts

```

1 public static void main(String[] args) {
2     Persona p = new Profesor();      // El objeto es un Profesor
3                                     // y siempre lo será, a pesar de los casts
4
5     // Error de compilación. Las personas no tienen ese método
6     p.impartirClase();
7
8     // Error de compilación. En general una Persona no es un Profesor
9     Profesor prof = p;
10
11    ((Profesor) p).impartirClase();
12    Profesor profe = (Profesor) p;
13
14    profe.hablar(); // "Hablo como un profesor"
15
16    // Upcast innecesario y sin efectos
17    ((Persona) profe).hablar(); // "Hablo como un profesor"
18
19    // Upcast implícito y sin efectos
20    Persona p2 = profe;
21    p2.hablar(); // "Hablo como un profesor"
22 }
```

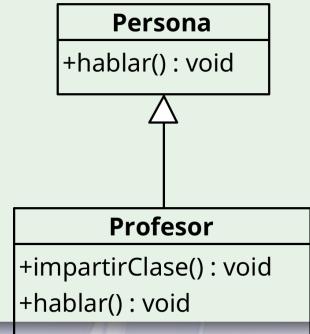


Ejemplo

Java: Ejemplo de casts con errores de ejecución

```

1 public static void main(String[] args) {
2
3     // Errores en tiempo de ejecución
4     // java.lang.ClassCastException: Persona cannot be cast to Profesor
5
6     Persona p = new Persona();
7     Profesor profe = (Profesor) p;      // Error
8
9     profe = ((Profesor) new Persona());    // Error
10
11    ((Profesor) p).impartirClase(); // Error
12
13    ((Profesor) ((Object) new Profesor())).impartirClase(); // OK
14
15 }
```

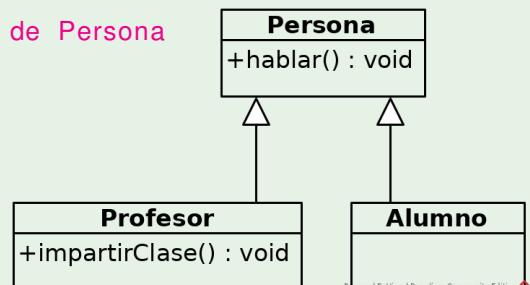


Ejemplo

Java: Ejemplo de casts entre clases “hermanas”

```

1 class Alumno extends Persona {
2     // Clase "hermana" de Profesor
3     // Alumno y Profesor son descendientes directos de Persona
4 }
5
6 public static void main(String[] args) {
7
8     // Error de compilación. Tipos incompatibles
9     Alumno a1 = new Profesor();
10
11    // Error de compilación. Tipos incompatibles
12    Alumno a2 = (Alumno) new Profesor();
13
14    // Error en tiempo de ejecución
15    // java.lang.ClassCastException: Profesor cannot be cast to Alumno
16    Alumno a3 = ((Alumno) ((Persona) new Profesor()));
17
18 }
```



Comprobaciones explícitas de tipos

- Deben evitarse las comprobaciones explícitas de tipos

Java: Mal ejemplo

```
1 public static void main(String[] args) {
2
3     Empleado e;
4     Random r = new Random();
5
6     if (r.nextBoolean()) {
7         e = new Comercial ("Pepe");
8     } else {
9         e = new Directivo ("Pepe");
10    }
11
12    // Nada recomendable
13    // Mal diseño
14    // No lo hagáis
15    // Lo digo en serio , no hagáis este tipo de diseños
16    String s;
17    if (e instanceof Empleado) {
18        s = "Soy " + e.getNombre();
19    } if (e instanceof Directivo) {
20        s = "Soy " + e.getNombre() + ", soy directivo";
21    } else if (e instanceof Comercial) {
22        s = "Soy " + e.getNombre() + ", soy comercial";
23    } else s = "";
24    System.out.println (s);
25 }
```

Comprobaciones explícitas de tipos

- Deben evitarse las comprobaciones explícitas de tipos

Java: Forma correcta de proceder

```
1 class Empleado {  
2     private String nombre;  
3     public Persona(String n) { nombre=n; }  
4     public String getNombre() { return nombre; }  
5     public String presentacion () { return ("Soy " + nombre); }  
6 }  
7  
8 class Comercial extends Empleado {  
9     public Comercial (String n) { super(n); }  
10    @Override  
11    public String presentacion () { return (super.presentacion() + ", soy comercial"); }  
12 }  
13  
14 class Directivo extends Empleado {  
15     public Directivo (String n) { super(n); }  
16     @Override  
17     public String presentacion () { return (super.presentacion() + ", soy directivo"); }  
18 }
```

Comprobaciones explícitas de tipos

- Deben evitarse las comprobaciones explícitas de tipos

Java: Forma correcta de proceder

```
1 public static void main(String[] args) {  
2  
3     Empleado e;  
4     Random r = new Random();  
5  
6     if (r.nextBoolean()) {  
7         e = new Comercial ("Pepe");  
8     } else {  
9         e = new Directivo ("Pepe");  
10    }  
11  
12    // Tenemos el comportamiento correcto de forma automática  
13  
14    System.out.println (e.presentacion());  
15 }
```

Detalles adicionales

- Con ligadura dinámica, **siempre se comienza buscando** el código asociado al método invocado en la clase que coincide **con el tipo dinámico de la referencia**
- Si no se encuentra se busca en la clase padre
- Así sucesivamente hasta encontrarlo o hasta que no existan ascendientes
- Esto sigue siendo cierto para métodos invocados desde otros métodos

Detalles adicionales

Ruby: Búsqueda del método a ejecutar

```
1 class Padre
2
3   def interno
4     puts "Interno padre"
5   end
6
7   def metodo
8     puts "Voy a actuar: "
9     interno
10  end
11
12 end
13
14 class Hija < Padre
15
16   def interno
17     puts "Interno hijo"
18   end
19
20 end
21
22 Padre.new.metodo # Voy a actuar: Interno padre
23 Hija.new.metodo # Voy a actuar: Interno hijo
```

Polimorfismo y ligadura dinámica

→ **Diseño** ←

- Estos mecanismos permiten crear diseños y codificaciones claros y fácilmente mantenibles
 - ▶ Volver a comparar las líneas 16 a 24 de la transparencia 15 con la línea 14 de la transparencia 17
(¡y solo hay involucradas 3 clases!)
- Deben tenerse en cuenta cuando:
 - ▶ Varias clases tienen el mismo método pero con distintas implementaciones
 - ▶ Existe relación de herencia entre dichas clases
 - ▶ No se sabe, a priori, a qué objeto concreto se le va a enviar el mensaje asociado a dicho método

Polimorfismo y Ligadura Dinámica

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)

1.3.4. Clases Abstractas

Clases Abstractas e Interfaces

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

- Tanto para clases abstractas como para interfaces
 - ▶ Conocer los conceptos y su utilidad en el diseño
 - ▶ Saber reconocerlos en un diagrama de clases, así como sus relaciones con otros elementos del diagrama
 - ▶ Saber implementarlos en Java

Contenidos

1 Introducción

2 Clases abstractas

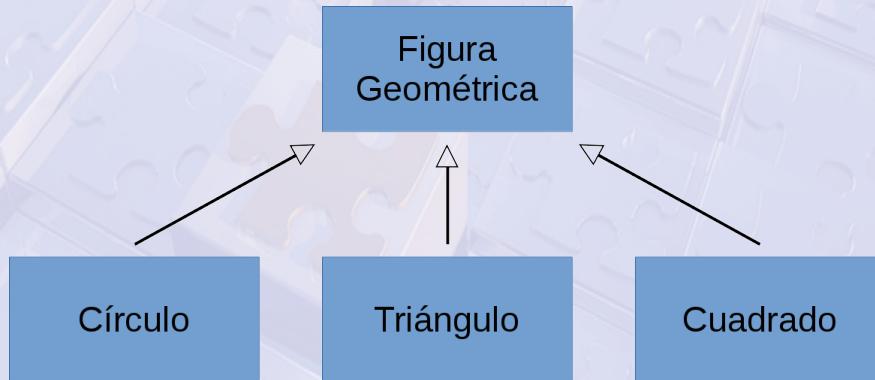
- Clases abstractas en Java
- Clases no instanciables en Ruby
- Clases abstractas en UML
- Ejemplos

3 Interfaces

- Interfaces en Java
- Interfaces en UML
- Ejemplos

Introducción a las clases abstractas

- Puede haber entidades a modelar de las que ...
 - ▶ Se sabe qué información contienen
 - ▶ Se sabe qué funcionalidad tienen
 - ▶ Pero no se sabe cómo realizan alguna de su funcionalidad
- Representan de manera genérica a otras entidades que sí concretan el funcionamiento desconocido



- Estas entidades se modelan mediante **Clases abstractas**

Clases abstractas

- Se declaran como tal y normalmente no proporcionan la implementación para alguno de sus métodos
 - ▶ Esos métodos sin implementación (solo cabecera) se denominan abstractos
- No es posible instanciar una clase abstracta
 - ▶ Pero sí declarar una variable usando una clase abstracta como tipo
- Son una **herramienta de diseño**:
 - ▶ Obligan a sus subclases a implementar una serie de métodos
 - ★ Si no implementan algún método, también serán abstractas
 - ▶ Proporcionan métodos y atributos comunes a esas subclases
 - ★ Sin perjuicio de que las subclases añadan atributos y/o métodos o redefinan métodos heredados
 - ▶ Definen un tipo de dato común a todas sus subclases
 - ★ Facilita usar objetos de dichas subclases sin conocer ni consultar explícitamente a qué clase pertenecen

Ejemplo de uso de clase abstracta

- Se desea tener una colección de figuras geométricas y poder calcular la sumatoria de sus áreas

Java: Un uso práctico de clases abstractas

```

1 abstract class FiguraGeometrica {
2     public abstract float area();
3 }
4 class Triangulo extends FiguraGeometrica { . . . } // Implementa area() adecuadamente
5 class Cuadrado extends FiguraGeometrica { . . . } // Implementa area() adecuadamente
6
7 // En algún otro sitio ...
8 ArrayList<FiguraGeometrica> colecciónDeFiguras = new ArrayList<>();
9
10 // Se rellena la colección con figuras de todo tipo y sin un orden concreto
11 colecciónDeFiguras.add (new Triangulo (lado1, lado2, lado3));
12 colecciónDeFiguras.add (new Cuadrado (lado));
13
14 float suma = 0.0f;
15 for (FiguraGeometrica unaFigura : colecciónDeFiguras) {
16     // No es necesario conocer de qué clase se instanció
17     // el objeto concreto que en cada momento está referenciado por unaFigura
18     suma += unaFigura.area();
19 }
```

Clases Abstractas en Java y Ruby

● Java

- ▶ Se usa la palabra reservada `abstract` para indicar que una clase y/o método son abstractos
- ▶ Permite clases abstractas sin métodos abstractos

● Ruby

- ▶ Ruby no soporta las clases abstractas
 - ★ No incorpora ningún mecanismo de comprobación por adelantado que el uso de una variable se ajusta a lo especificado en una clase
 - ★ ¿Cómo se implementaría el ejemplo de las figuras geométricas?

El ejemplo de las figuras geométricas en Ruby

Ruby: El ejemplo de las figuras geométricas

```
1 class Triangulo
2   ...
3   def area
4     ...
5   end
6 end
7
8 class Cuadrado
9   ...
10  def area
11    ...
12  end
13 end
14
15 colecciónDeFiguras = []
16 colecciónDeFiguras << Triangulo.new(lado1, lado2, lado3)
17 colecciónDeFiguras << Cuadrado.new(lado)
18
19 suma = 0.0
20 for figura in colecciónDeFiguras do
21   suma += figura.area()
22 end
```

- No ha sido necesario disponer de una clase `FiguraGeometrica`

Clases no instanciables en Ruby

- Supuesto práctico:

- ▶ Se necesita una clase, en Ruby, que aglutine atributos y/o métodos comunes de sus clases derivadas
- ▶ Esa clase no podrá instanciarse
- ▶ Sus clases derivadas sí podrán instanciarse

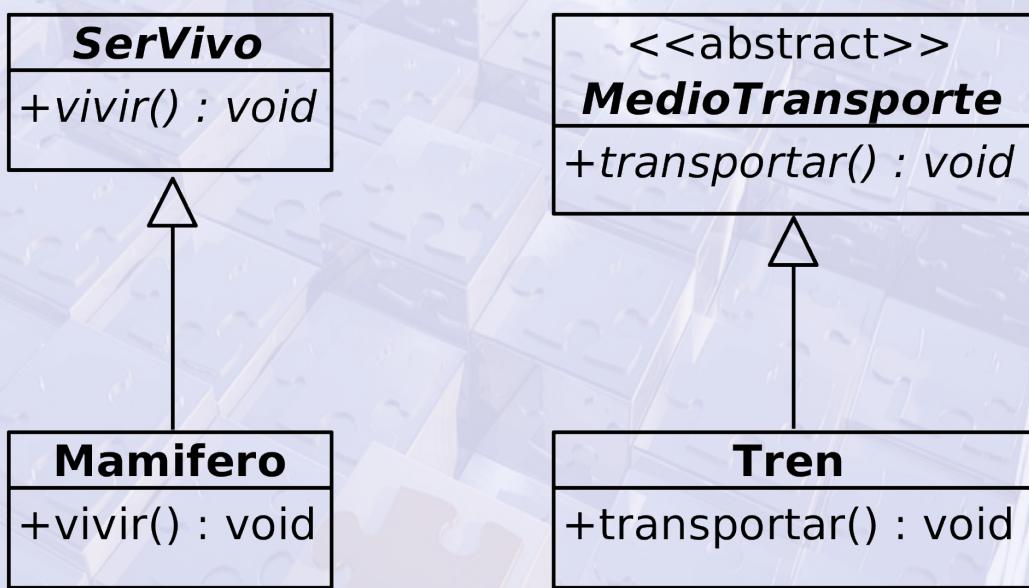
- Solución:

- ▶ Se hace privado el método `new` en la clase padre
- ▶ Se vuelve a hacer público el método `new` en las clases derivadas

Ruby: Clases no instanciables

```
1 class FiguraGeometrica
2     . . .
3     # Atributos y métodos comunes
4     private_class_method :new
5
6 class Cuadrado < FiguraGeometrica
7     public_class_method :new
8     . . .
9     # Atributos y métodos específicos de esta subclase
9 end
```

Representación UML de las clases abstractas



- El nombre de la clase abstracta y de los métodos abstractos se escribe en *cursiva*
 - ▶ ¡Cuidado! A veces se os pasa inadvertido en algún examen 😊

Ejemplo

Java: Ejemplo de clase y método abstracto

```
1 abstract class SerVivo {  
2     String planeta;  
3     SerVivo (String p) {  
4         planeta = p;  
5     }  
6     public String existir() {return "Existiendo";} // No obligatorio  
7     public abstract String vivir();  
8 }  
9  
10 class Humano extends SerVivo {  
11     String nombre;  
12     Humano (String p, String n) {  
13         super (p);  
14         nombre = n;  
15     }  
16     // "Obligatorio" Si no se redefine, esta clase también será abstracta  
17     @Override  
18     public String vivir() {  
19         return "Viviendo como humano";  
20     }  
21  
22     // No obligatorio  
23     @Override  
24     public String existir() {  
25         return super.existir() + " como humano";}  
26     }  
27 }
```

Interfaces

- Una interfaz define un determinado **protocolo de comportamiento** (T. Budd p.88) y permite reutilizar la especificación de dicho comportamiento
- Será una clase la que lo implemente (**realización**)
- Una interfaz define un contrato que cumplen las clases que realizan dicha interfaz
- **Cada interfaz define un tipo**
 - ▶ Se pueden declarar variables de ese tipo
 - ▶ Dichas variables podrán referenciar instancias de clases que realicen dicha interfaz

Interfaces

Interfaces: Ejemplo

```
1 interface ListaEnteros {  
2     // Cabeceras de métodos sin implementación  
3     boolean empty ();  
4     . . .  
5 }  
6  
7 class ListaEnteros_array implements ListaEnteros {  
8     // Estructura de datos que implemente una lista de enteros mediante un array  
9     boolean empty () {  
10         // se implementan los métodos según la estructura de datos concreta usada  
11     }  
12     . . .  
13 }  
14  
15 class ListaEnteros_punteros implements ListaEnteros {  
16     // La estructura de datos que implemente la lista mediante punteros  
17     boolean empty () {  
18         // la implementación es específica para esta estructura de datos  
19     }  
20     . . .  
21 }  
22  
23 void main (String args[]) {  
24     ListaEnteros v1 = new ListaEnteros_array();  
25     ListaEnteros v2 = new ListaEnteros_punteros();  
26     // Ambas variables se usan igual, con los métodos de la interfaz  
27 }
```

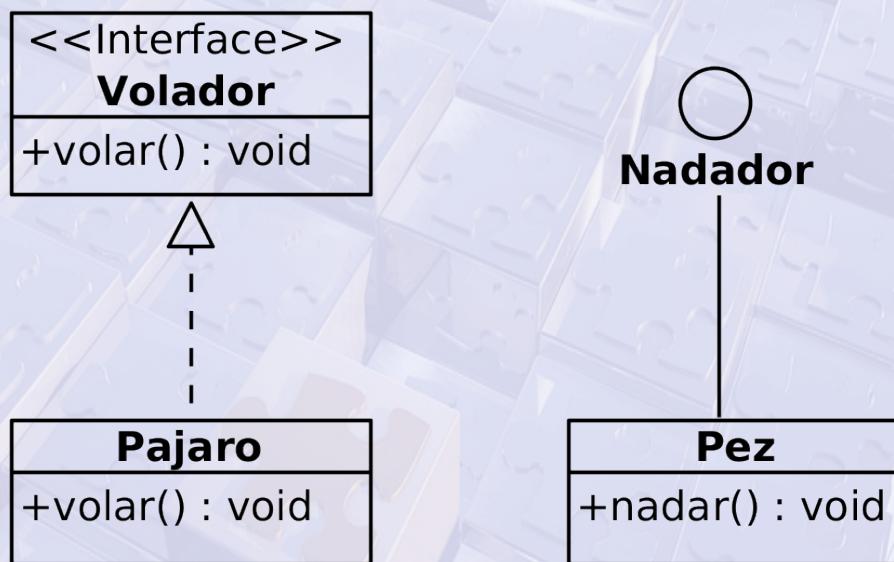
Interfaces en Java

- Una clase puede realizar varias interfaces
- Una interfaz puede heredar de una o más interfaces
- Una interfaz solo puede tener:
 - ▶ Constantes
 - ▶ Signaturas de métodos
 - ▶ Métodos tipo default
 - ▶ El equivalente a los métodos de clase static
- Solo los métodos tipo default y static pueden tener asociada implementación
- Los métodos son public y las constantes public, static, y final
- No pueden ser instanciadas, solo realizadas por clases o extendidas por otras interfaces

Interfaces en Java

- Se pueden redefinir los métodos default en interfaces que heredan y en clases que realizan esa interfaz
- Una clase puede heredar de una clase y además realizar varias interfaces
- Una clase abstracta puede indicar que realiza una interfaz sin implementar alguno de sus métodos.
 - ▶ Fuerza a hacerlo a sus descendientes no abstractos
- Una clase puramente abstracta se parece a una interfaz pero Java no permite herencia múltiple
- **Ruby** no soporta de forma nativa el concepto de interfaz

Representación UML de las interfaces



Ejemplo

Java: Ejemplo de interfaces

```
1 interface Interfaz1 {
2     int CONSTANTE = 33;
3
4     String hazAlgo1 ();
5     default String hazAlgo12 () {return "1";}
6     default String hazAlgo11 () {return "11";}
7 }
8
9 interface Interfaz2 {
10    String hazAlgo2 ();
11    default String hazAlgo12() {return "2";}
12 }
13
14 class Test implements Interfaz1, Interfaz2 {
15     @Override
16     public String hazAlgo1() {return "algo1";}
17
18     @Override
19     public String hazAlgo2() {return "algo2";}
20
21     @Override
22     public String hazAlgo12() { // Por colisión debe redefinir
23         String a=Interfaz1.super.hazAlgo12();
24         String b=Interfaz2.super.hazAlgo12();
25         return (a+" "+b+" "+Integer.toString(Interfaz1.CONSTANTE));
26     }
27 }
```

Clases abstractas

→ **Diseño** ←

- Recurso muy utilizado para *aglutinar clases similares*
 - ▶ Obviamente, debe tener sentido en el contexto del modelo
- Unido al mecanismo de polimorfismo, permite usar esas clases de una manera muy limpia
 - ▶ Recordar el ejemplo de las figuras geométricas
 - ▶ Se puede codificar sin tener que consultar explícitamente a qué clase concreta pertenece cada objeto
 - ▶ Permite que el sistema sea fácilmente extensible con relativamente poco trabajo
 - ★ Por ejemplo, añadir nuevas clases que deriven de la clase abstracta

Interfaces

→ **Diseño** ←

- Muy usados para independizar:
 - ▶ El “qué” (cabeceras de métodos).
En la interfaz
 - ▶ El “cómo” (implementación de los mismos).
En las clases que lo realizan
- Permitiendo tener varias implementaciones
- Ejemplo
 - ▶ Imaginad una interfaz Lista que declara todo lo que se puede hacer con una lista
 - ▶ Se pueden tener varias clases que realizan esa interfaz implementando la lista con arrays, con punteros, doblemente enlazada, etc.
 - ▶ En una aplicación que use listas a través de la interfaz se puede cambiar de una implementación a otra sin apenas modificar nada en la aplicación

Clases Abstractas e Interfaces

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)

1.3.5. Clases Parametrizables

Clases Parametrizables

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

- Conocer las clases parametrizables y su utilidad
- Saber identificarlas en un diagrama de clases
- Saber definirlas y utilizarlas

Contenidos

1 Introducción

2 Clases parametrizables

3 Clases parametrizables en UML

4 Clases parametrizables en Java

- Clases parametrizables e interfaces

Introducción a las clases parametrizables

- Suponed que se necesita

- ▶ Una lista de objetos de la clase Persona
- ▶ Una lista de objetos de la clase Vehículo
- ▶ Una lista de objetos de la clase Mascota
- ▶ Se estima que se pueden necesitar listas de objetos de otras clases
- ▶ Todas las listas se van a gestionar igual: insertar, borrar, etc.

★ ¿De qué modo podría diseñarse/implementarse?

Pseudocódigo: ¿Mejorable?

```
1 class ListaPersona { void insertar (Persona p) { ... } ... }
2 class ListaVehículo { void insertar (Vehículo v) { ... } ... }
3 class ListaMascota { void insertar (Mascota m) { ... } ... }
```



Clases parametrizables

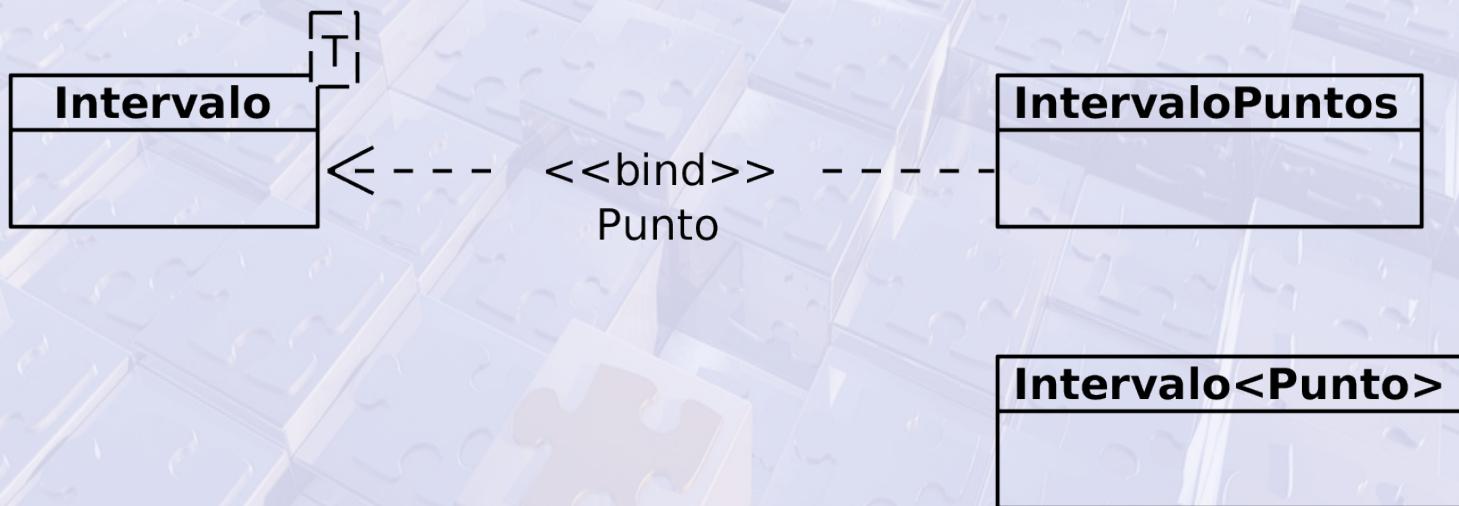
- Clases definidas en función de un tipo de dato (clase)
- Se generalizan un conjunto de operaciones válidas para diferentes tipos de datos
- El ejemplo clásico son los contenedores
 - ▶ Se puede definir una lista independientemente del tipo concreto de elementos que vaya a contener

Pseudocódigo: Clase parametrizable

```
1 // Lista de objetos de la clase cualquiera T
2 class Lista <T> { void insertar (T e) {...} ... }
3
4 // Cuando se necesite una lista de cualquier clase,
5 // solo hay que instanciarla indicando la clase concreta para esa lista
6
7 Lista <Persona> listaPersonas = new ...
8 Lista <Vehiculo> listaVehiculos = new ...
9 Lista <Mascota> listaMascotas = new ...
```



Clases parametrizables en UML



Clases parametrizables en Java

- Este concepto se implementa mediante los tipos genéricos (*generics*)
- Permite pasar tipos como parámetros a clases e interfaces
 - ▶ Esos parámetros (que representan tipos) se pueden usar allí donde habitualmente se usa un tipo, por ejemplo:
 - ★ Al declarar un atributo
 - ★ Al declarar el tipo devuelto por un método
 - ★ Al declarar el tipo de un parámetro de un método
- Se puede forzar que el tipo suministrado a una clase parametrizable:
 - ▶ Tenga que ser subclase de otro, o

```
class Clase <T extends ClaseBase>
```
 - ▶ Tenga que realizar una interfaz

```
class Clase <T extends Interfaz>
```

Ejemplo

Java: Clase parametrizable

```

1 public class Equipo<T extends Jugador> {
2
3     private String nombre;
4     private T capitán;
5     private ArrayList<T> jugadores;
6
7     public Equipo (String nom, T cap) {
8         nombre = nom;
9         capitán = cap;
10        jugadores = new ArrayList<>();
11        jugadores.add (cap);
12    }
13
14    public T getCapitán () {
15        return capitán;
16    }
17
18    public ArrayList<T> getJugadores () {
19        return jugadores;
20    }
21
22    public void addJugador (T jug) {
23        if (!jugadores.contains (jug)) {
24            jugadores.add (jug);
25        }
26    }
27 }
```



: Uso de la clase

```

1 public static void main(String [] args)
2 {
3     Futbolista pele;
4     pele = new Futbolista ("Pelé");
5
6     Equipo<Futbolista> brasil;
7     brasil =new Equipo<>("Brasil",pele);
8
9     Futbolista tostao;
10    tostao = new Futbolista ("Tostao");
11    brasil.addJugador (tostao);
12
13    Baloncestista gasol;
14    gasol = new Baloncestista ("Gasol");
15
16    // Error, gasol no es Futbolista
17    brasil.addJugador(gasol);
18 }
```

Comprobación de tipos en tiempo de compilación

- Suponer el siguiente caso práctico
 - ▶ Un centro de estudios organiza cursos de apoyo para estudiantes de primaria y secundaria
 - ▶ Se necesita una clase `Curso` con (entre otros) un método `matricularEstudiante`
 - ▶ En un curso no puede haber estudiantes de diferentes ciclos

Java: Solución sin clases parametrizables

```

1 abstract class Estudiante { . . . }
2 class EstudiantePrimaria extends Estudiante { . . . }
3 class EstudianteSecundaria extends Estudiante { . . . }
4 class Curso {
5     void matricularEstudiante (Estudiante e) { . . . }
6 } // Es responsabilidad del programador evitar cursos con estudiantes de diferentes ciclos

```

Java: Solución con clases parametrizables

```

1 . .
2 class Curso < T extends Estudiante > {
3     void matricularEstudiante (T e) { . . . }
4 } // La comprobación de tipos evita matricular estudiantes de diferentes ciclos

```

Clases parametrizables e interfaces

- La implementación de un método de una clase parametrizable puede requerir que `T` disponga de un determinado método

Java: Se asume que `T` tiene un determinado método

```
1 class Mazo <T> {  
2     T getCopiaPrimeraCarta () {  
3         T primeraCarta = cartas.remove (0);  
4         cartas.add (primeraCarta);  
5         return primeraCarta.copia ();  
6         // Se requiere que las clases que sustituyan a T tengan un método T copia()  
7     }  
8 }
```

- En ese caso:
 - ▶ El método requerido formará parte de una interfaz
 - ▶ Al declarar la clase parametrizable se indicará que el tipo que sustituya al parámetro debe realizar dicha interfaz

Ejemplo de clases parametrizables e interfaces

Java: Ejemplo de clases parametrizables e interfaces

```
1 // Las interfaces también pueden hacerse paramétricas , como las clases
2 interface Copiable <T> {
3     public T copia();
4 }
5
6 class Sorpresa implements Copiable<Sorpresa> {
7     // Unas cartas Sorpresa para algún juego
8     // Entre otras operaciones, implementa copia
9     public Sorpresa copia () {
10         return Sorpresa(this);    // Hace uso de un constructor de copia
11     }
12 }
13
14 class Mazo < T extends Copiable<T> > {    // Se requiere que T realice Copiable<T>
15     T getCopiaPrimeraCarta () {
16         T primeraCarta = cartas.remove (0);
17         cartas.add (primeraCarta);
18         return primeraCarta.copia ();
19         // primeraCarta , de tipo T, que realiza Copiable , sí dispone del método copia.
20     }
21 }
22
23 // Ya se puede instanciar un mazo de sorpresas
24 Mazo<Sorpresa> mazoSorpresas = new Mazo<>();
```

Clases e interfaces parametrizables → *Diseño* ←

- Tenerlas en cuenta en aquellos casos en los que la responsabilidad de una clase implique trabajar con objetos de clases desconocidas a priori
- Si se requiere que las clases que sustituyan el parámetro implementen unos métodos concretos, recurrir a interfaces para *obligar* a que dichas clases los implementen
- Se tiene el añadido de la comprobación de tipos en tiempo de compilación

Clases Parametrizables

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)

1.3.6. Herencia en Ámbito de Clases

Herencia en el Ámbito de Clase

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

- Entender las diferencias existentes entre Java y Ruby relacionadas con la herencia en el ámbito de clase

Contenidos

1 Java

2 Ruby

Java

- No permite la redefinición de métodos de clase al mismo nivel que de instancia
- Aunque pueden existir métodos de clase con el mismo nombre en una jerarquía de clases, no se obtienen los mismos resultados que a nivel de instancia

Ejemplo

Java: Ejemplo de herencia en el ámbito de clase

```
1 class Padre {  
2     public static final int DECLASE = 1;  
3     public static int getDECLASE() { return DECLASE; }  
4 }  
5  
6 class Hija extends Padre {  
7     public static final int DECLASE = 2; // Variable shadowing  
8 }  
9  
10 class Nieta extends Hija{  
11     public static int getDECLASE() { // No es una redefinición  
12         // super.getDECLASE() No permitido  
13         return DECLASE;  
14     }  
15 }  
16  
17 public static void main(String[] args) {  
18     System.out.println (Padre.DECLASE); // 1  
19     System.out.println (Hija.DECLASE); // 2  
20     System.out.println (Nieta.DECLASE); // 2  
21     System.out.println (Padre.getDECLASE()); // 1  
22     System.out.println (Hija.getDECLASE()); // 1  
23     //porque "redefine" el método de clase  
24     System.out.println (Nieta.getDECLASE()); // 2  
25 }
```

Ejemplo

Java: Ejemplo de herencia en el ámbito de clase

```
1 public static void main(String[] args) {  
2 // El tipo estático de las instancias influye  
3 // Aunque Java lo permite, no se debe invocar a métodos de clase así  
4 // Lo digo en serio  
5  
6 Padre p=new Padre();  
7 System.out.println (p.getDECLASE()); // 1  
8  
9 p = new Nieta();  
10 System.out.println (p.getDECLASE()); // 1  
11  
12 Nieta n = new Nieta();  
13 System.out.println (n.getDECLASE()); // 2  
14  
15 }  
16 }
```

Ruby

- La clases son *first class citizens* y en el ámbito de clase todo funciona como es de esperar

Ejemplo

Ruby: Ejemplo de herencia en el ámbito de clase

```
1 class Padre
2   @atributo_clase1 = 1
3   @atributo_clase2 = 2
4   @@atributo_clase3 = 5
5   def self.salida
6     puts @atributo_clase1+1
7     puts @atributo_clase2+1 unless @atributo_clase2.nil?
8     puts @@atributo_clase3+1
9   end
10  def self.salida2
11    salida
12  end
13 end
14 Padre.salida # 2 3 6
15 class Hija < Padre
16   @atributo_clase1 = 3
17   @@atributo_clase3 = 7
18   def self.salida2
19     super # Las clases son "first class citizens"
20     puts @atributo_clase1+1
21   end
22 end
23 Padre.salida # 2 3 8
24 Hija.salida # 4 8
25 Padre.salida2 # 2 3 8
26 Hija.salida2 # 4 8 4
```

Herencia en el Ámbito de Clase

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)

1.3.7. Herencia Múltiple

Herencia Múltiple

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://medias.maisondumonde.com/image/upload/q_auto,f_auto/w_2000/img/estanteria-de-metal-negro-y-abeto-con-reloj-1000-0-31-188836_1.jpg
- El resto de imágenes e ilustraciones son de creación propia, al igual que los ejemplos de código

Objetivos

- Comprender en qué consiste la herencia múltiple
- Entender los problemas que puede ocasionar
- Conocer alternativas

Contenidos

1 Herencia múltiple

2 Problemas comunes

3 Alternativas

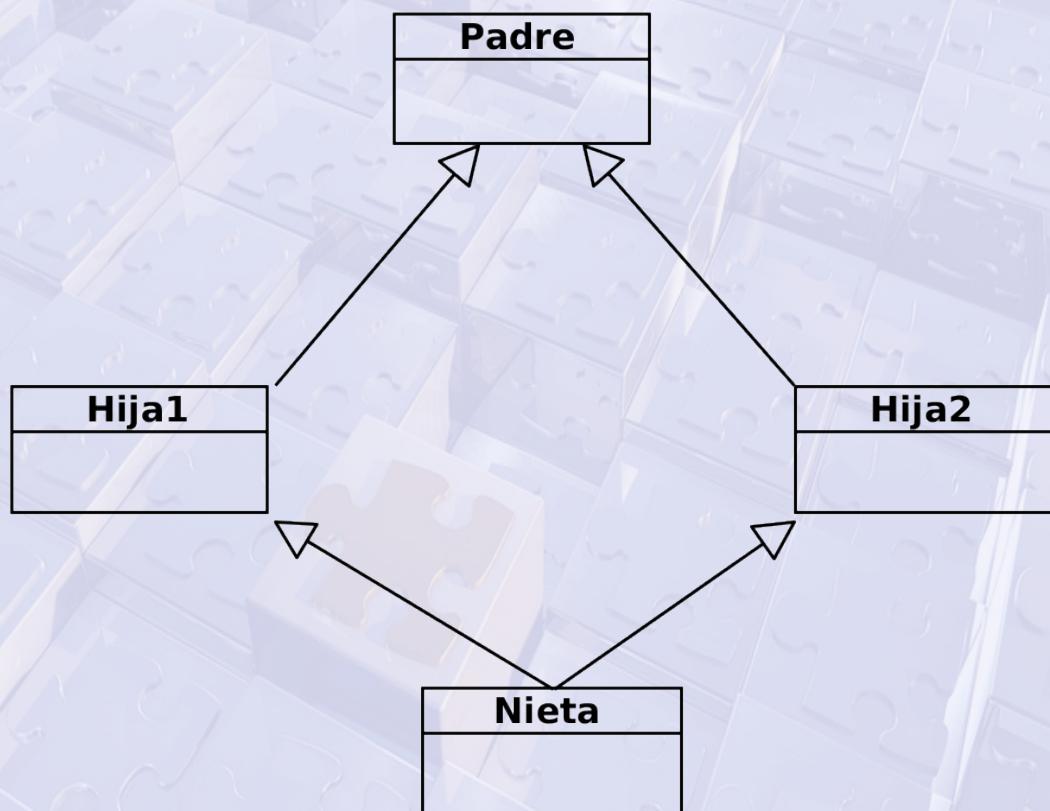
Herencia múltiple

- Se produce cuando una clase es descendiente de más de una superclase
- Permite representar problemas donde un objeto tiene propiedades según criterios distintos (clasificado según criterios distintos)
- Presenta problemas de implementación y pocos lenguajes la soportan (Ej. C++ y Python)
- Java y Ruby no tienen herencia múltiple

Problemas comunes

- Colisión de nombres de métodos y/o atributos
- Problema del diamante:
 - ▶ Provoca duplicidad en los elementos heredados
- No hay que olvidar que para que tenga sentido debe haber una relación es-un de la clase descendiente con todos sus ascendientes
 - ▶ La reutilización de código existente en varias clases no es por si solo un criterio para establecer relaciones de herencia múltiple

Problema del diamante



Ejemplo del problema del diamante

C++: Herencia múltiple. Problema del diamante

```
1 class Persona
2 {
3     private:
4         string nombre;
5     public:
6         Persona() {cout<<"Creada Persona SIN INICIALIZAR" <<endl;}
7         Persona(string n) {cout<<"Creada Persona e inicializada" <<endl; nombre = n;}
8         string getNombre() {return nombre;}
9         void setNombre(string n) {nombre = n;}
10    };
11
12 class Docente: public Persona
13 {
14     public:
15         Docente(string n): Persona(n) {}
16         void presentaDocente() {cout<<"Soy el docente "<<getNombre()<<endl;}
17    };
18
19 class Investigador: public Persona
20 {
21     public:
22         Investigador(string n): Persona(n) {}
23         void presentaInvestigador() {cout<<"Soy el investigador "<<getNombre()<<endl;}
24    };

```

Ejemplo del problema del diamante

C++: Herencia múltiple. Problema del diamante

```
1 class Profesor: public Docente, public Investigador
2 {
3     public:
4         Profesor(string n): Docente(n), Investigador(n){}
5         void presentaProfesor(){presentaDocente(); presentarInvestigador();}
6         void modificaNombre(string n) {cout<<"Me modiflico el nombre" << endl; setNombre(n);}
7     // Error: hay ambigüedad
8     void modificaNombre(string n) {cout<<"Me cambio de nombre" << endl; Docente::setNombre(n);}
9 }
10
11 int main(int argc, char **argv) {
12     Profesor *p = new Profesor("Ana");
13     // Creada Persona e inicializada
14     // Creada Persona e inicializada
15
16     p->presentaProfesor();
17     // Soy el docente Ana
18     // Soy el investigador Ana
19
20     p->modificaNombre("Juan");
21     p->presentaProfesor();
22     // Soy el docente Juan
23     // Soy el investigador Ana
24 }
```

Ejemplo del problema del diamante

C++: Herencia múltiple. Solución C++ a la duplicidad de atributos

```
1 class Persona
2 {
3     private:
4         string nombre;
5     public:
6         Persona() {cout<<"Creado A SIN INICIALIZAR" <<endl;}
7         Persona(string n) {cout<<"Creada Persona e inicializada" <<endl; nombre = n;}
8         string getNombre() {return nombre;}
9         void setNombre(string n) {nombre = n;}
10    };
11
12 class Docente: virtual public Persona
13 {
14     public:
15         Docente(string n): Persona(n) {}
16         void presentaDocente() {cout<<"Soy el docente "<<getNombre()<<endl;}
17    };
18
19 class Investigador: virtual public Persona
20 {
21     public:
22         Investigador(string n): Persona(n) {}
23         void presentaInvestigador() {cout<<"Soy el investigador "<<getNombre()<<endl;}
24    };

```

Ejemplo del problema del diamante

C++: Herencia múltiple. Solución C++ a la duplicidad de atributos

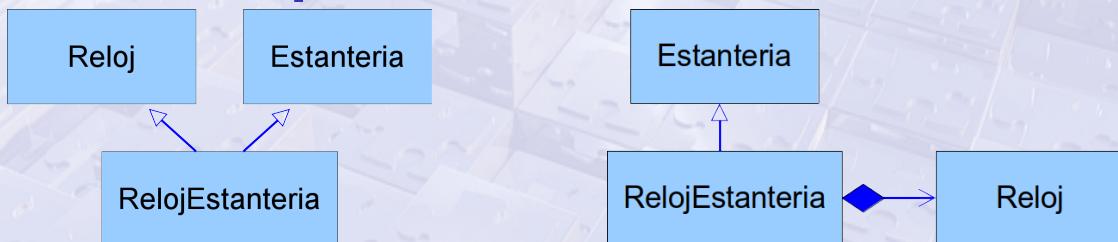
```
1 class Profesor: public Docente, public Investigador
2 {
3     public:
4         Profesor(string n): Persona(n), Docente(n), Investigador(n){}
5         void presentaProfesor(){presentaDocente(); presentalInvestigador();}
6         void modificaNombre(string n) {cout<<"Me modiflico el nombre" << endl; setNombre(n);}
7         // Ya no hay ambigüedad
8     };
9
10 int main(int argc, char **argv) {
11     Profesor *p = new Profesor("Ana");
12     // Creada Persona e inicializada
13     // Creada Persona e inicializada
14
15     p->presentaProfesor();
16     // Soy el docente Ana
17     // Soy el investigador Ana
18
19     p->modificaNombre("Juan");
20     p->presentaProfesor();
21     // Soy el docente Juan
22     // Soy el investigador Juan
23
24     Docente *d = new Docente("Pepe");
25     d->presentaDocente();
26     // Las instancias de Docente tambien tienen el atributo Persona::nombre
27 }
```

Alternativas

- Composición
 - ▶ Sustituir una o varias relaciones de herencia por composición
- Interfaces Java
 - ▶ Se pueden realizar varias interfaces Java y heredar de una superclase
- Mixins de Ruby
 - ▶ Permiten incluir código proveniente de varios módulos como parte de una clase

Alternativas

Ejemplo de composición

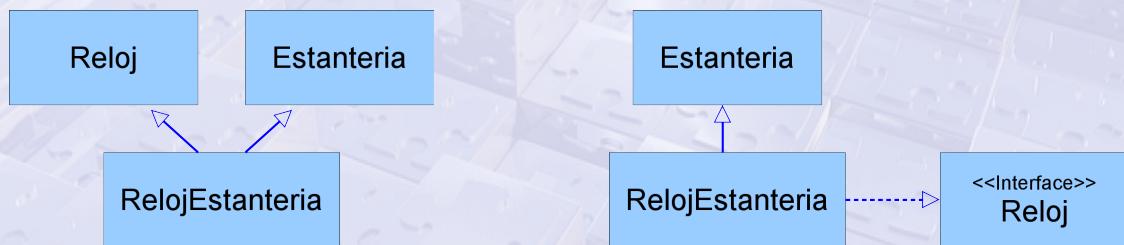


Java: Ejemplo de composicion

```
1 class Estanteria { . . . }
2
3 class Reloj { . . . }
4
5 class RelojEstanteria extends Estanteria {
6     private Reloj reloj;
7
8     RelojEstanteria () {
9         super();
10        reloj = new Reloj();
11    }
12
13 // Los métodos de Estanteria se heredan
14
15 void setHora (Hora h) {      // Los métodos de Reloj se definen
16     reloj.setHora (h);          // se implementan reenviando el mensaje al atributo
17 }
18 }
```



Ejemplo con interfaces Java



Java: Ejemplo con interfaces Java

```

1 class Estanteria { . . . }
2
3 interface Reloj { public void setHora (Hora h); public Hora getHora (); }
4
5 class RelojEstanteria extends Estanteria implements Reloj {
6
7     RelojEstanteria () {
8         super();
9     }
10
11    // Los métodos de Estanteria se heredan
12
13    // Se implementan los métodos de la interfaz
14
15    public void setHora (Hora h) { . . . }
16    public Hora getHora () { . . . }
17 }
  
```

Ejemplo de mixin de Ruby

Ruby: Ejemplo de mixin de Ruby

```
1 module Volador
2   def volar
3     puts "Volando"
4   end
5 end
6
7 module Nadador
8   def nadar
9     puts "Nadando"
10  end
11 end
12
13 class Ejemplo
14   def metodo
15     puts "Método propio"
16   end
17
18 include Volador # Añadimos todo el módulo
19 include Nadador # Añadimos todo el módulo
20 end
21
22 e=Ejemplo.new
23 e.metodo
24 e.volar
25 e.nadar
```

Herencia Múltiple

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)

1.4. Tema 4

1.4.1. Modelo Vista Controlador

Modelo Vista Controlador

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-1299287/>
 - ▶  <https://pixabay.com/images/id-341444/>
 - ▶  <https://pixabay.com/images/id-1020156/>
 - ▶  <https://pixabay.com/images/id-3383459/>
- El resto de imágenes e ilustraciones son de creación propia, al igual que los ejemplos de código

Objetivos

- Entender qué es un patrón de diseño y su utilidad
- Entender el patrón Modelo Vista Controlador y el reparto de responsabilidad que realiza

Contenidos

1 Patrón de diseño

2 Modelo Vista Controlador

Patrón de diseño

- Un patrón de diseño describe un problema que ocurre numerables veces en nuestro entorno
- Describe además el núcleo de una solución a ese problema de forma que sea reutilizable
- Permite aprovechar soluciones previas probadas y validadas a problemas conocidos

Patrón de diseño

Patrón de diseño



Modelo Vista Controlador (MVC)

- Es un patrón de tipo arquitectónico, al igual que la programación por capas, arquitectura orientada a servicios, etc.
- Elementos:

Modelo: Clases que representan la lógica del problema

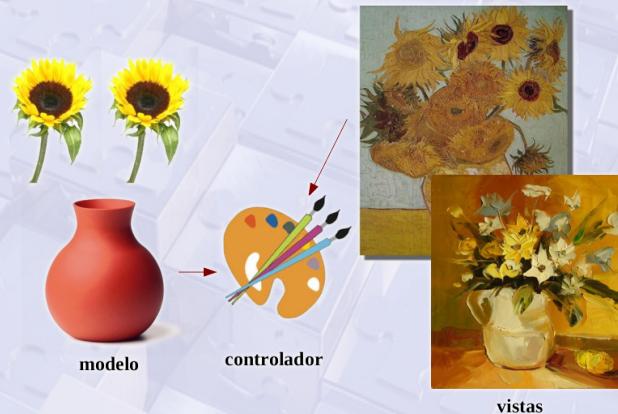
Vista: Una representación visual de los datos del modelo para mostrarlos al usuario

- ▶ La interacción del usuario se produce con elementos de la vista

Controlador: Actúa de intermediario entre la vista y el modelo

Modelo Vista Controlador

- Para un mismo modelo se pueden tener diferentes vistas



- La información fluye en ambas direcciones

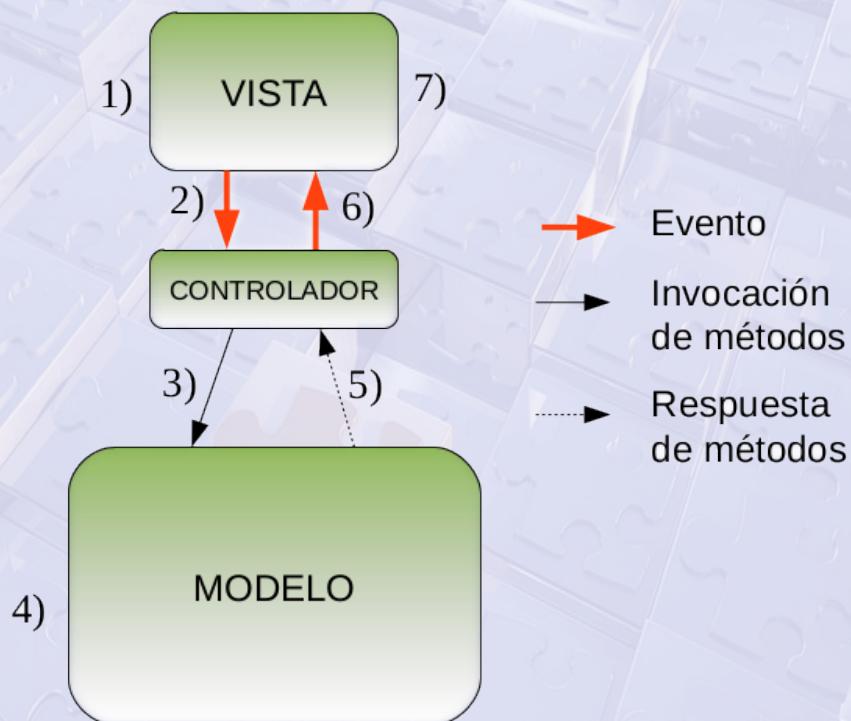


La responsabilidad del Controlador

- Ante una orden del usuario, siempre a través de la vista, que implique cambios en el modelo
 - ▶ Es el controlador quien la ejecuta actuando sobre el modelo
- ← Cuando se producen cambios en el modelo, estos cambios deben verse reflejados en las vistas correspondientes
 - ▶ El controlador puede actuar de intermediario en este proceso
- ↔ La mayoría de las veces, una acción del usuario implica un recorrido de *ida y vuelta*.

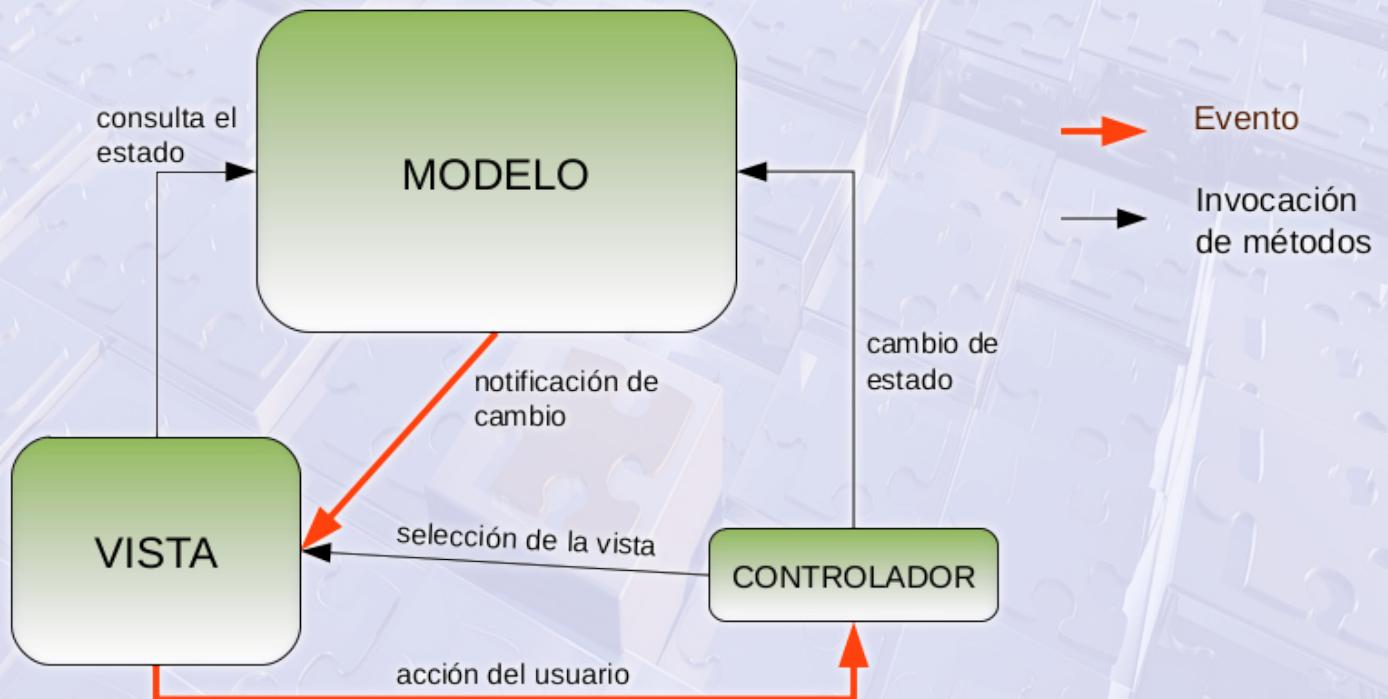
Esquema básico de MVC

- Sin comunicación directa entre modelo y vista



Esquema alternativo de MVC

- El modelo puede informar directamente a la vista para que se actualice



Modelo Vista Controlador

→ *Diseño* ←

- En lecciones anteriores se han comentado los principios de alta cohesión y bajo acoplamiento
- No solo se tienen en cuenta en el diseño a nivel de clases
- También deben contemplarse a otros niveles
- Esta arquitectura Modelo Vista Controlador es un ejemplo de ello

Modelo Vista Controlador

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)

1.4.2. Reflexión

Reflexión

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

- Conocer el qué consiste la reflexión

Contenidos

1 **Reflexión**

2 **Reflexión en Java**

3 **Reflexión en Ruby**

Reflexión

- Capacidad de un programa para manipularse a sí mismo y comprender sus propias estructuras en tiempo de ejecución
- Mecanismos
 - ▶ **Introspección**
Habilidad del programa para observar y razonar sobre su mismo estado (objetos y clases) en tiempo de ejecución
 - ▶ **Modificación**
Habilidad del programa para cambiar su estado (objetos y clases) durante la ejecución
 - ★ Normalmente solo soportado por lenguajes interpretados

Reflexión en Java

- Debido a la estructura de metaclasses desarrollada por Java, el nivel de reflexión que se permite es de introspección
- Toda la funcionalidad para ello está definida en la clase Class de Java

Java: Ejemplos

```
1 // Ejemplos
2 MiClase obj = new MiClase();
3 Class clase = obj.getClass() //método definido en Object
4 Field[] varInstancia = clase.getFields();
5 Constructor[] construct = clase.getConstructors();
6 Method[] metodosInstancia = clase.getMethods();
7 String nombreClase = clase.getSimpleName();
```

Reflexión en Ruby

- Debido a la estructura de metaclasses desarrollada por Ruby, el nivel de reflexión que se permite es de introspección y de modificación.
- En ejecución se puede:
 - ▶ Consultar y modificar una clase
 - ▶ Consultar y modificar la estructura y funcionalidad de un objeto haciéndolo distinto de los demás de la misma clase

Ejemplo en Ruby

Ruby: Modificando la clase. Afecta a todas las instancias

```
1 class Libro
2   def initialize(titulo)
3     @titulo = titulo
4   end
5 end
6
7 libro1 = Libro.new("El señor de los anillos")
8
9 # Se modifica la clase y afecta a todas las instancias
10 Libro.class_eval do
11   def publicacion(añopublicacion)
12     @añoPublicacion = añopublicacion
13   end
14 end
15
16 libro1.publicacion(1997) # Se invoca el nuevo método
17 puts libro1.inspect      # Ahora tiene un atributo adicional
```

Ejemplo en Ruby

Ruby: Modificando una única instancia

```
1 # Se modifica solo una instancia
2 libro1.instance_eval do
3   def autor(autor)
4     @autor = autor
5   end
6 end
7
8 libro1.autor("J. R. R. Tolkien")
9 puts libro1.inspect
10
11 libro2 = Libro.new("Cien años de soledad")
12 libro2.autor("G. García Márquez") # ERROR: undefined method 'autor'
```

Ejemplo en Ruby

Ruby: Ejemplos de introspección

```
1 puts Libro.instance_methods(false) # publicacion
2 # El parámetro indica si queremos solo los métodos de esa clase (false)
3 # o también los heredados (true)
4
5 puts libro1.instance_variables
6      # @autor
7      # @titulo
8      # @añoPublicacion
9
10 puts libro2.instance_variables      # @titulo
11 puts libro1.instance_of?(Libro)      # true
```

Reflexión

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)

1.4.3. Copia de Objetos

Copia 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

- Entender la diferencia entre realizar
 - ▶ Copia de identidad
 - ▶ Copia de estado superficial
 - ▶ Copia de estado profunda
- Entender qué son los objetos inmutables y su relación con la copia de objetos
- Saber qué es la copia defensiva y los problemas que puede haber si no se usa
- Saber realizar copias profundas con clone y con constructores de copia

Contenidos

1 Introducción

2 Copia defensiva

- clone y la interfaz Cloneable
- Constructor copia
- Ruby y clone
- Copia por serialización

Introducción a la copia de objetos

- Una operación muy habitual en programación es esta:

Pseudocódigo: Asignación

```
1 copia = original;  
2 // ¿Qué se realiza con dicha operación?  
3 // Si modiflico original ¿se ve afectada copia? ¿Y si modiflico copia?
```

- No es una operación trivial tratándose de objetos
- De hecho, existen distintos casos:
 - ▶ Copia de identidad
 - ▶ Copia de estado
 - ★ ¿Y si un atributo del objeto a copiar referencia a otro objeto?
 - ★ ¿Cómo lo copiamos? ¿Por identidad o por estado?
 - ★ Si es por estado, ¿cuándo parar?
 - ▶ Todo esto puede quedar “oculto” bajo el operador de asignación

Introducción a la copia de objetos

- Aparecen dos conceptos
 - ▶ Profundidad de la copia
 - ★ Hasta que nivel se van a realizar copias de estado en vez de identidad
 - ▶ Inmutabilidad de los objetos
 - ★ Un objeto es inmutable si no dispone de métodos que modifiquen su estado
 - Hay que tener cuidado con los objetos que referencian a otros si estos no son inmutables
 - ★ ¿Por qué?

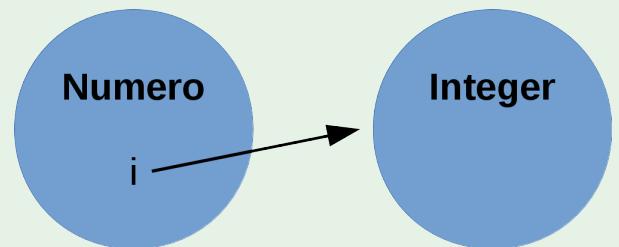
Ejemplo

- ¿Por qué puede ser necesario realizar copias de estado?

Java: Una clase mutable que almacena un entero

```

1 class Numero{
2     private Integer i;
3
4     public Numero(Integer a) {
5         i = a;
6     }
7
8     public void inc() {
9         // Este método modifica el estado del objeto
10        // La clase es mutable
11        i++;
12    }
13
14    @Override
15    public String toString() {
16        return i.toString();
17    }
18 }
19 }
```

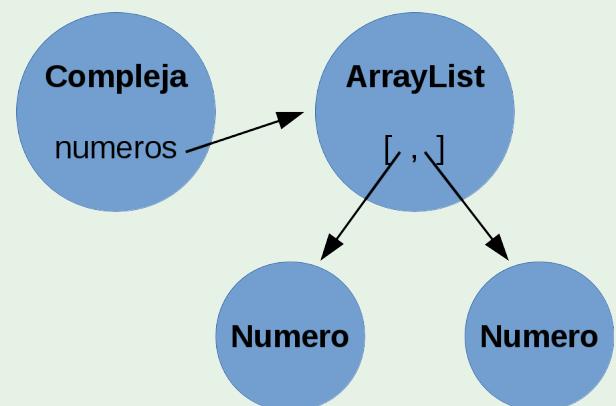


Ejemplo

Java: Una clase que es una colección de los números anteriores

```

1 class Compleja {
2     // Clase mutable que referencia a objetos mutables
3
4     private ArrayList<Numero> numeros;
5
6     public Compleja() {
7         numeros = new ArrayList<>();
8     }
9
10    public void add(Numero i) {
11        numeros.add(i);
12    }
13
14    ArrayList getNumeros() {
15        return numeros;
16    }
17 }
```

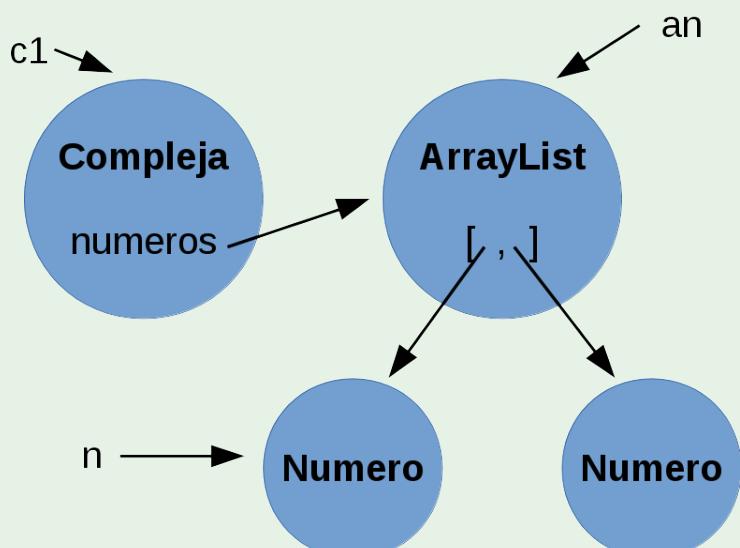


Ejemplo

Java: Probamos las clases anteriores

```

1 Compleja c1 = new Compleja();
2
3 c1.add (new Numero(3));
4 c1.add (new Numero(2));
5
6 // Acceso sin restricción a la lista
7 ArrayList<Numero> an = c1.getNumeros();
8 an.clear();
9 c1.add (new Numero(33));
10
11 for (Numero i : c1.getNumeros()) {
12     System.out.println(i); // R--> 33
13 }
14 //Se devuelven referencias al estado interno
15 Numero n = c1.getNumeros().get(0);
16 n.inc();
17
18 for(Numero i : c1.getNumeros()) {
19     System.out.println(i); // R--> 34
20 }
```



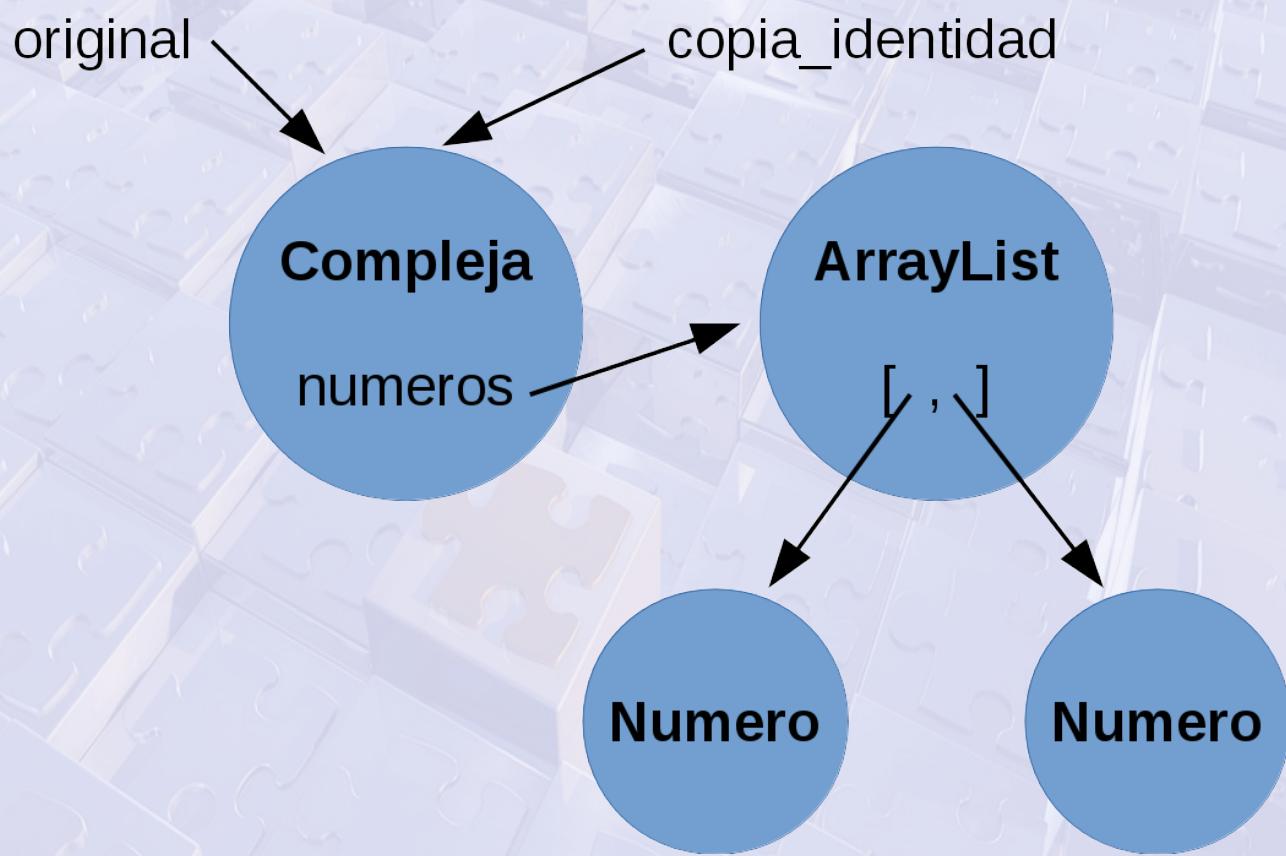
- Se expone:
 - ▶ El estado de los objetos **Compleja**
 - ▶ El de los objetos **Numero** referenciados por los objetos **Compleja**

Copia defensiva

- Alude a devolver una copia de estado en vez de devolver una copia de identidad
 - ▶ Objetivo: Evitar que el estado de un objeto se modifique sin usar los métodos que la clase designa para ello
 - ▶ Requisito: Realizar copias profundas y no solo copias superficiales
 - ▶ Cuándo: Los consultores deben usar este recurso con los objetos mutables que devuelvan
- En el ejemplo anterior:
 - ▶ La lista de números no es inmutable porque existen métodos para poder alterarla
 - ▶ Se debería duplicar la lista y devolver esa copia en el consultor `getNumeros()`

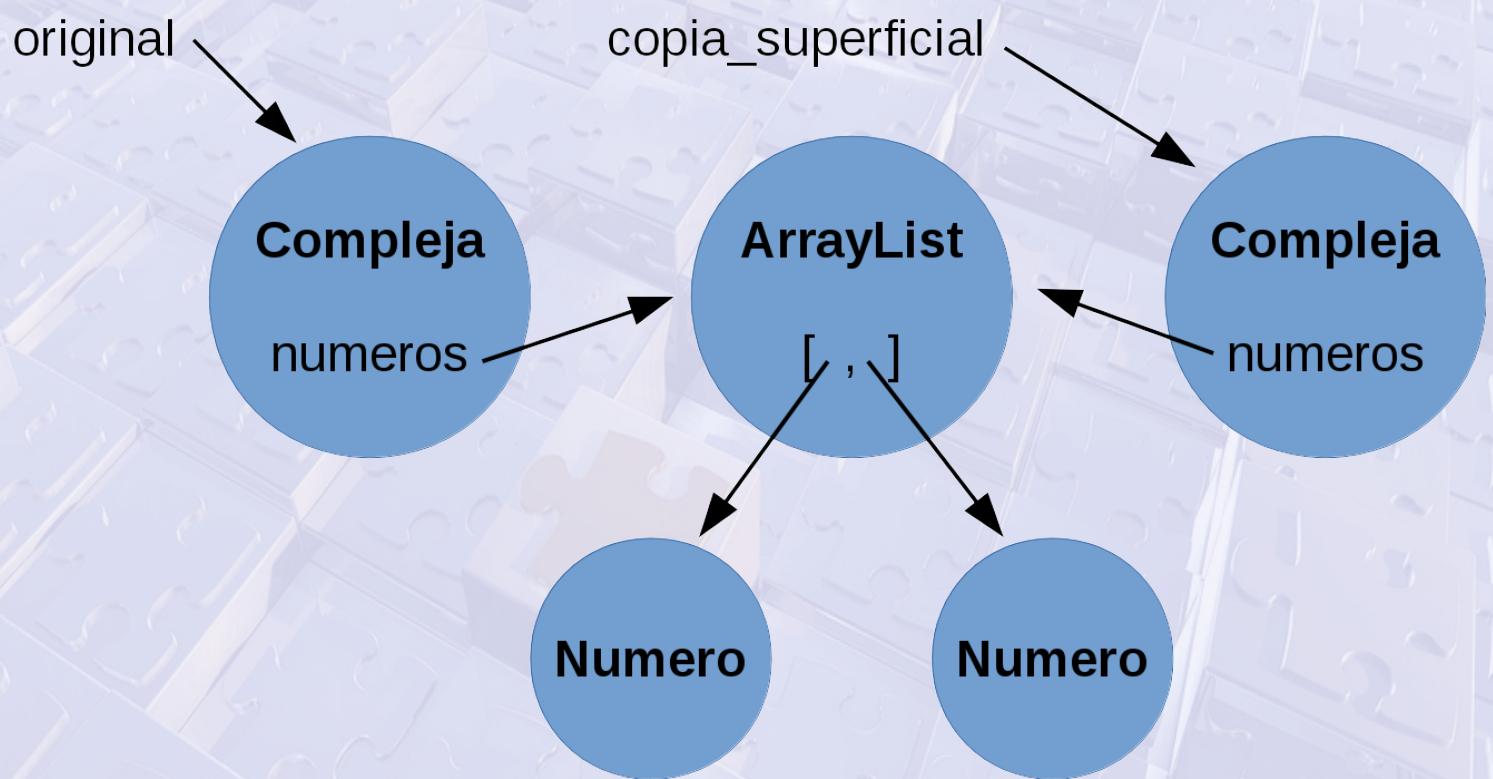
Tipos de copia

Copia de identidad



Tipos de copia

Copia superficial



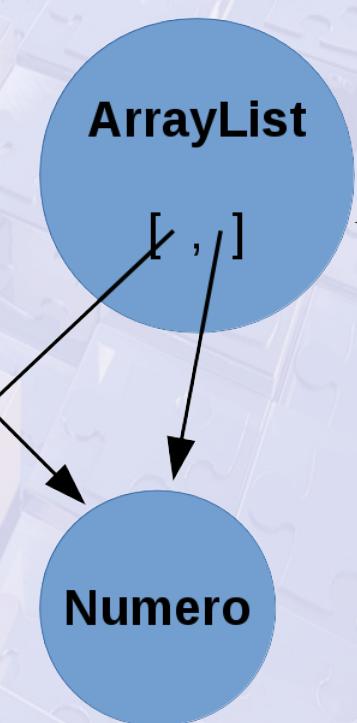
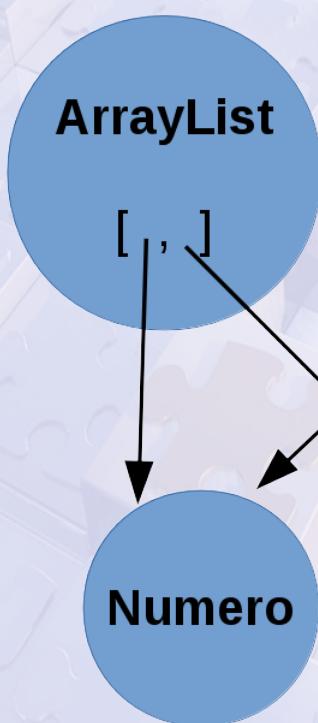
Tipos de copia

Copia superficial

original



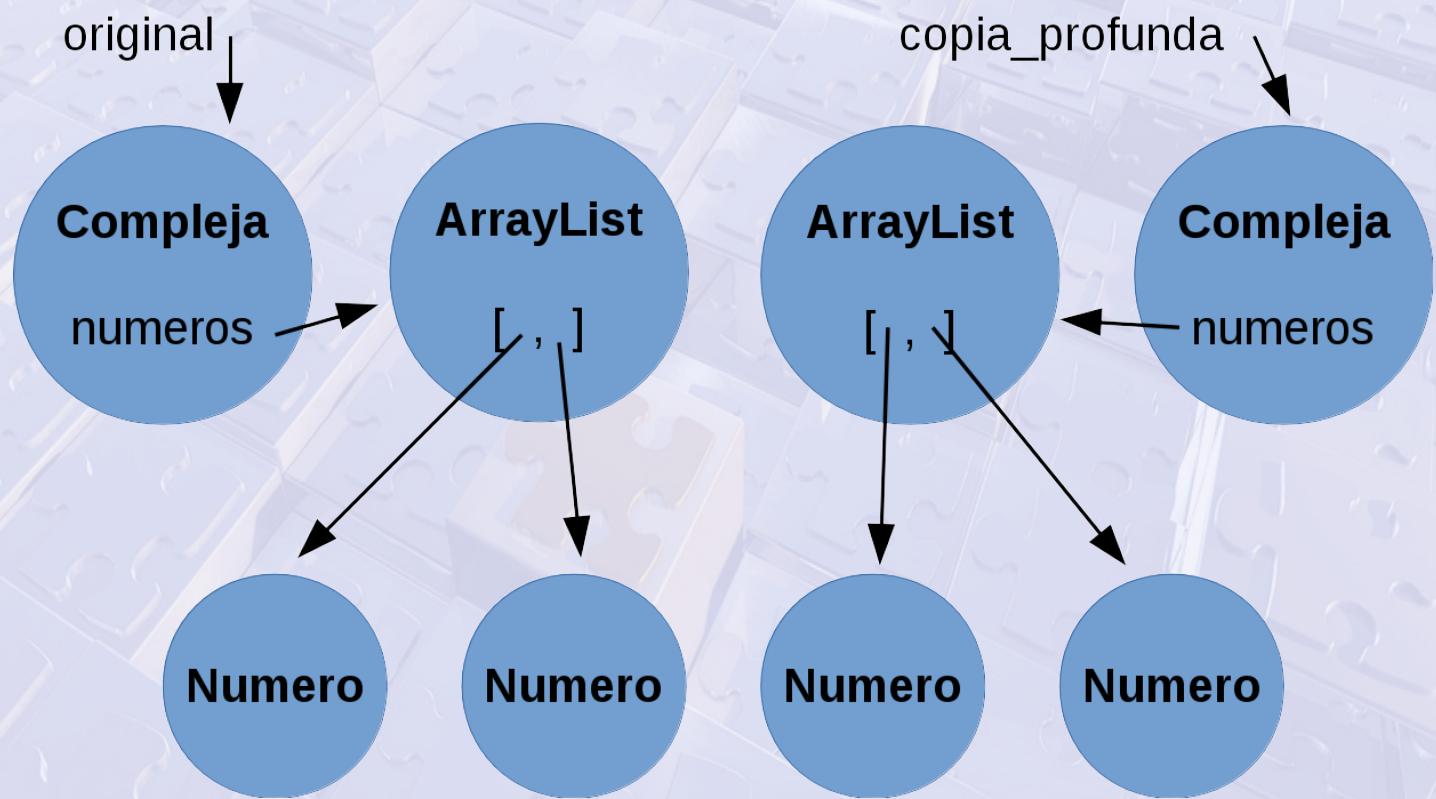
copia_superficial_también



Tipos de copia

Copia profunda

- **Todos** los objetos mutables se han duplicado



Ejemplo

Java: Version un poco más segura de la clase Compleja

```
1 class ComplejaSegura {  
2     private ArrayList numeros;  
3  
4     public ComplejaSegura() {  
5         numeros = new ArrayList();  
6     }  
7  
8     public void add(Numero i) {  
9         numeros.add(i);  
10    }  
11  
12    ArrayList getNumeros() {  
13        return (ArrayList) (numeros.clone());  
14        // Usamos clone para devolver una copia del estado de numeros  
15    }  
16 }
```

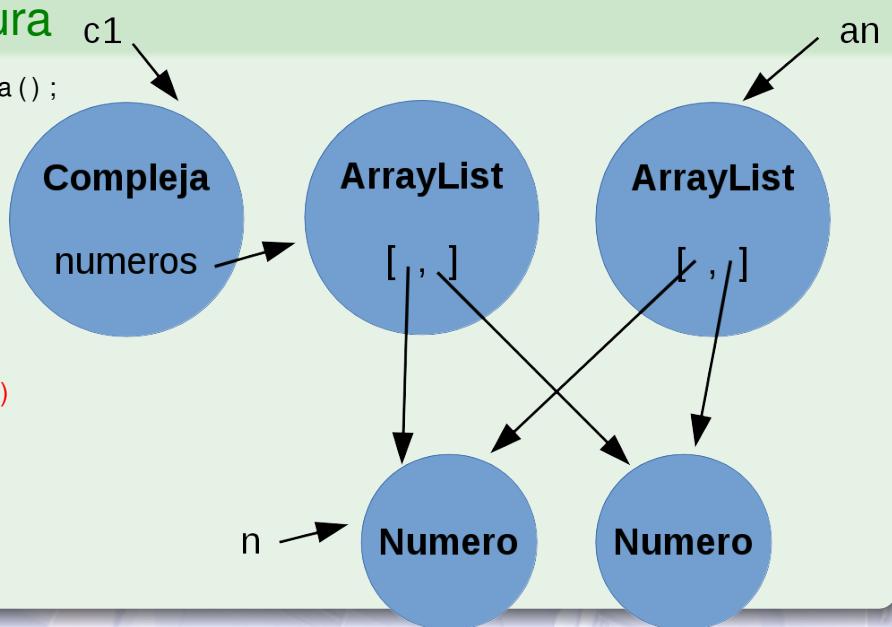
Ejemplo

Java: Usando ComplejaSegura

```

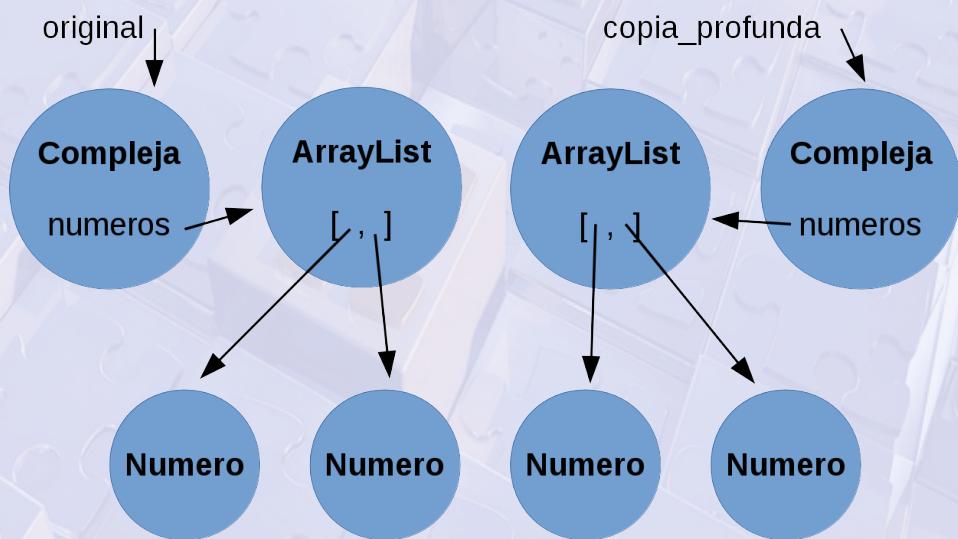
1 ComplejaSegura c1 = new ComplejaSegura();
2
3 c1.add (new Numero(3));
4 c1.add (new Numero(2));
5
6 c1.getNumeros().clear();
7 for (Numero i : c1.getNumeros()) {
8     System.out.println(i);
9 } // R --> 3 2    OK!   :-)
10
11 ArrayList<Numero> an = c1.getNumeros()
12 Numero n = an.get(0)
13 n.inc(); // Sigue siendo un problema
14
15 for (Numero i:c1.getNumeros()) {
16     System.out.println(i);
17 } // R--> 4 2   :-(

```



Copia profunda

- Hay que llegar a nivel requerido en cada caso
- En el ejemplo anterior no solo hay que duplicar la lista sino también los elementos de la misma
 - ▶ En caso contrario ambas listas compartirán las referencias a los mismos objetos
 - ▶ Y eso es un problema al tratarse de objetos mutables



Ejemplo

Java: Mejorando la clase Numero

```
1 class Numero implements Cloneable{
2     private Integer i;
3
4     public Numero(Integer a) {
5         i = a;
6     }
7
8     public void inc() {
9         i++;
10    }
11
12    @Override
13    public Numero clone() throws CloneNotSupportedException {
14
15        return (Numero) super.clone();
16
17        // Los objetos de la clase Integer son inmutables
18        // por ello, la implementación realizada es válida
19        // no siendo necesaria la implementación que está comentada
20
21        /*
22        Numero nuevo = (Numero) super.clone();
23        nuevo.i = i.clone();
24        return nuevo;
25        */
26    }
27 }
```

Ejemplo

Java: Mejorando más la clase Compleja

```
1 class ComplejaMasSegura implements Cloneable {
2     // ...
3     ArrayList getNumeros() {
4         ArrayList nuevo = new ArrayList();
5         Numero n = null;
6         for (Numero i : this.numeros) {
7             try {
8                 n = i.clone();
9             } catch (CloneNotSupportedException e) {
10                 System.err.println ("CloneNotSupportedException" );
11             }
12             nuevo.add (n);
13         }
14         return (nuevo);
15     }
16
17     @Override
18     public ComplejaMasSegura clone() throws CloneNotSupportedException {
19         ComplejaMasSegura nuevo= (ComplejaMasSegura) super.clone();
20         nuevo.numeros = this.getNumeros(); // Ya se hace copia profunda
21         return nuevo;
22     }
23
24     // También habría que modificar el método    public void add (Numero i)
25     // ¿Cómo se implementaría?
26 }
```

clone y la interfaz Cloneable

- Al utilizar este mecanismo se redefine

```
1   protected Object clone() throws CloneNotSupportedException
```

- Se suele redefinir de la siguiente forma

```
1   public MiClase clone() throws CloneNotSupportedException
```

- El método creado debe crear una copia base (`super.clone()`) y crear copias de los atributos no inmutables
 - Esto último puede conseguirse usando también el método `clone` sobre esos atributos

Reflexiones sobre clone y la interfaz Cloneable

- Según la documentación oficial

- ▶ *Creates and returns a copy of this object. The precise meaning of “copy” may depend on the class of the object. The general intent is that, for any object x, the expressions are true:*

```
1  x.clone () != x
2  x.clone () .getClass () == x.getClass ()
3  x.clone () .equals (x)
4  //These are not absolute requirements !!!!
```

- ▶ *By convention, the object returned by this method should be independent of this object (which is being cloned)*

Reflexiones sobre clone y la interfaz Cloneable

- La interfaz Cloneable no define ningún método.
- Esto rompe con el significado habitual de una interfaz en Java

```
1  class Raro implements Cloneable {  
2      //Este código no produce errores  
3  }
```

- En la clase Object se realiza la comprobación de si la clase que originó la llamada a clone implementa la interfaz
- Si no es así se produce una excepción

Reflexiones sobre clone y la interfaz Cloneable

- El hecho de que la interfaz Cloneable no se comporte como el resto de interfaces y que el funcionamiento del sistema de copia de objetos basado en el método clone esté basado en “recomendaciones” ha sido ampliamente criticado por diversos autores
- Otro hecho también muy criticado es el relativo a que al utilizar el método clone no intervienen los constructores en todo el proceso
- Los atributos “final” no son compatibles con el uso de clone

Constructor copia

- También es posible copiar objetos creando un constructor que acepte como parámetro objetos de la misma clase
- Este constructor se encargaría de hacer la copia profunda
- Este esquema presenta algunos problemas cuando se utilizan jerarquías de herencia

Ejemplo

Java: Constructor de copia

```
1 class Padre {  
2     private int x;  
3     private int y;  
4  
5     public Padre (int a,int b) {  
6         x = a;  
7         y = b;  
8     }  
9  
10    public Padre (Padre a) {  
11        x = a.x;  
12        y = a.y;  
13    }  
14  
15    @Override  
16    public String toString() {  
17        return "(" + Integer.toString(x) + "," + Integer.toString(y) + ")";  
18    }  
19 }
```

Ejemplo

Java: Constructor de copia en una clase que hereda de A

```
1 class Hija extends Padre {  
2     private int z;  
3  
4     public Hija (int a, int b, int c) {  
5         super (a, b);  
6         z = c;  
7     }  
8  
9     public Hija (Hija b) {  
10        super (b);  
11        z = b.z;  
12    }  
13  
14    @Override  
15    public String toString () {  
16        return super.toString () + "(" + Integer.toString (z) + ")";  
17    }  
18 }
```

Ejemplo

Java: Usamos las clases anteriores

```
1 Padre a1 = new Padre (11, 22);
2 Padre a2 = new Hija (111, 222, 333);
3 Padre a3;
4
5 // El que se ejecute esta línea o la siguiente determina
6 // el constructor copia al que llamar
7
8 // a3 = a1;
9 a3 = a2;
10
11 Padre a4 = null;
12
13 // ¿ El tipo dinámico de a3 es Padre o es Hija ?
14
15 // a4 = new Padre (a3);
16 a4 = new Hija ((Hija) a3);
17
18 System.out.print(a4);
```

Ejemplo

Java: Usamos reflexión para evitar el problema anterior

```
1 import java.lang.reflect.*;
2 import java.util.logging.*;
3
4 // Obtenemos el constructor copia
5 Class cls = a3.getClass();
6 Constructor constructor = null ;
7 try {
8     constructor = cls.getDeclaredConstructor();
9 } catch (NoSuchMethodException e) {}
10
11 //Usamos el constructor copia
12 try {
13     a4 = (Padre) ((constructor == null) ? null : constructor.newInstance (a3));
14 } catch (InstantiationException ex) {
15     Logger.getLogger(Padre.class.getName()).log(Level.SEVERE, null , ex);
16 } catch (IllegalAccessException ex) {
17     Logger.getLogger(Padre.class.getName()).log(Level.SEVERE, null , ex);
18 } catch (IllegalArgumentException ex) {
19     Logger.getLogger(Padre.class.getName()).log(Level.SEVERE, null , ex);
20 } catch (InvocationTargetException ex) {
21     Logger.getLogger(Padre.class.getName()).log(Level.SEVERE, null , ex);
22 }
```

Ejemplo

Java: También se puede encapsular el constructor de copia

```
1 // Añadimos en la clase Padre el método
2 Padre copia() {
3     return new Padre (this);
4 }
5
6 // Y lo redefinimos en la clase Hija
7 @override
8 Hija copia() {
9     return new Hija (this);
10 }
11
12 // La copia de a3 en a4 quedaría así
13 a4 = a3.copia();
```

- Es decir, dejamos que sea la ligadura dinámica (resolviendo el método copia adecuado) la que elija el constructor de copia correcto

Ruby y clone

- En Ruby el método `clone` de la clase `Object` también realiza la copia superficial
- Si se desea realizar la copia profunda debe realizarla el programador

Copia por serialización

- En Ruby se puede recurrir a la serialización, deserialización para crear una copia profunda

```
1   b = Marshal.load ( Marshal.dump(a) )
```

- En este proceso el objeto se convierte a una secuencia de bits y después se construye otro a partir de esta secuencia
- Esta última técnica también es aplicable a Java y en ambos casos es poco eficiente, no aplicable en todos los escenarios
- La documentación de Ruby advierte que el uso del método load puede llevar a la ejecución de código remoto

Copia de objetos

→ *Diseño* ←

- Entonces, ¿en todas las asignaciones de parámetros mutables y en todas las devoluciones de atributos mutables debemos realizar copias defensivas?
 - ▶ No tiene porqué. Depende:
 - ★ De dónde vengan los parámetros a asignar
 - ★ A quién se le dé el atributo
 - ★ Del software que se esté desarrollando, etc.
 - ▶ Hay que estudiar cada caso y decidir
 - ▶ Por ejemplo:
 - ★ No es igual diseñar un paquete que solo es usado por nuestro equipo de desarrollo (como el caso del paquete Irrgarten)
 - ★ Que diseñar una biblioteca genérica que van a usar terceros (como una biblioteca con clases matemáticas genéricas)
 - ★ En el segundo caso podemos ser “más defensivos” que en el primero

Copia 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)

2 Relaciones de Problemas

2.1. Relación 1 con soluciones

Ejercicios: Objetos

1. Diseñar una clase para representar Personas. Decidir los atributos que consideres necesarios y los métodos que incluirías para que la clase fuese útil en un software.
2. Tomando la clase diseñada en el ejercicio anterior:
 - a) Instanciar un objeto de dicha clase (con datos inventados)
 - b) Indicar cuál es el estado de dicho objeto que has instanciado
 - c) Indicar cuál es su comportamiento
 - d) ¿Cómo harías para tener otro objeto igual en identidad al objeto instanciado en a)? Hazlo.
 - e) ¿Cómo harías para tener otro objeto distinto en identidad pero igual en estado al objeto instanciado en a)? Hazlo.

2.2. Relación 2 con soluciones

Ejercicios: Atributos y Métodos

1. Crear una clase Coche. Para cada coche se almacenará su marca, modelo y matrícula. También se desea una variable que contabilice el número total de coches construidos.
 - a) Hacerlo en Java. ¿Has tenido que usar la pseudovariable this en el constructor? Explicar por qué.
 - b) Hacerlo en Ruby considerando una **variable de clase** (@@@) para el contador de coches
 - c) Hacerlo en Ruby considerando un **atributo de instancia de la clase** (@) para el contador de coches.
2. En la clase del ejercicio anterior añadir un método que devuelva un String con el estado del objeto receptor del mensaje. Decidir si el método debe ser de instancia o de clase.
 - a) Hacerlo en Java
 - b) Hacerlo en Ruby
3. En la clase de los ejercicios anteriores añadir un método que devuelva el número de coches que se han instanciado. Decidir si el método debe ser de instancia o de clase.
 - a) Hacerlo en Java
 - b) Hacerlo en Ruby en las 2 versiones que se han hecho en el ejercicio 1. ¿Has tenido que usar la pseudovariable self en este ejercicio? Explicar por qué.
4. Pensar un ejemplo en el que sea necesario usar una variable de clase para almacenar una constante numérica. Implementar dicha clase y un ejemplo de uso donde se emplee dicha constante en una instrucción.
 - a) Hacerlo en Java
 - b) Hacerlo en Ruby considerando una **variable de clase** para la constante.
 - c) Hacerlo en Ruby considerando un **atributo de instancia de la clase** para la constante.
5. Los siguiente ejemplo en Java es correcto. Fijarse en el uso que se hace de atributos y métodos privados.

```
1: public class CloudDrive {  
2:     private int capacity;  
3:  
4:     public CloudDrive (int capacity) {  
5:         this.capacity = capacity;  
6:     }  
7:  
8:     private int getCapacity() {  
9:         return capacity;  
10:    }  
11:  
12:    public void setFromOther (CloudDrive other) {  
13:        capacity = other.capacity;  
14:    }  
15:  
16:    public void setFromOther_v2 (CloudDrive other) {  
17:        capacity = other.getCapacity();  
18:    }
```

Se ha intentado reproducir dicho código en Ruby con una traducción literal como la mostrada en el fragmento siguiente pero lamentablemente presenta errores. Localizar los errores y proponer soluciones.

```
1: class Cloud_drive
2:   def initialize (capacity)
3:     @capacity = capacity
4:   end
5:
6:   private
7:
8:   def get_capacity
9:     return @capacity
10:  end
11:
12:  public
13:
14:  def set_from_other (other)
15:    @capacity = other.@capacity
16:  end
17:
18:  def set_from_other_v2 (other)
19:    @capacity = other.get_capacity
20:  end
```

6. En Java, ¿se puede acceder desde un método de instancia a una variable de clase de la misma clase? ¿En qué circunstancias?
7. En Ruby, ¿se puede acceder desde un método de instancia a una variable de clase (@@) de la misma clase? ¿En qué circunstancias?
8. En Ruby, ¿se puede acceder desde un método de instancia a un atributo de instancia de la clase (@) de la misma clase? ¿En qué circunstancias? ¿Qué puede hacerse para conseguir el acceso?

1a



```
1 class Coche
2     @num_coches = 0
3
4     attr_accessor :marca, :modelo, :matricula
5
6     def initialize(marca, modelo, matricula)
7         @marca = marca
8         @modelo = modelo
9         @matricula = matricula
10        self.class.incrementar_num_coches
11    end
12
13    def self.incrementar_num_coches
14        @num_coches += 1
15    end
16    def self.num_coches
17        @num_coches
18    end
19
20    def to_s
21        "Marca: #{@marca}, Modelo: #{@modelo}, Matricula: #{@matricula}"
22    end
23 end
24
25 COCHE1 = Coche.new("Ford", "Focus", "1234ABC")
26 puts "Coche 1: \n"
27 puts COCHE1.to_s
28 puts "Coche 1 es una instancia de: #{COCHE1.class}"
29 puts Coche.num_coches
30
31 COCHE2 = Coche.new("Chevrolet", "Corvette", "5678DEF")
32 puts "Coche 2: \n"
33 puts COCHE2.to_s
34 puts "Coche 2 es una instancia de: #{COCHE2.class}"
35 puts Coche.num_coches
```

1b1c



```
1 public class Coche {  
2     private String marca;  
3     private String modelo;  
4     private String matricula;  
5     private static int numCoches = 0;  
6  
7     public Coche(String marca, String modelo, String matricula) {  
8         this.marca = marca;  
9         this.modelo = modelo;  
10        this.matricula = matricula;  
11        numCoches++;  
12    }  
13  
14    public static int getNumCoches() {  
15        return numCoches;  
16    }  
17  
18    public String getMarca() {  
19        return marca;  
20    }  
21  
22    public String getModelo() {  
23        return modelo;  
24    }  
25  
26    public String getMatricula() {  
27        return matricula;  
28    }  
29  
30    public void setMarca(String marca) {  
31        this.marca = marca;  
32    }  
33  
34    public void setModelo(String modelo) {  
35        this.modelo = modelo;  
36    }  
37  
38    public void setMatricula(String matricula) {  
39        this.matricula = matricula;  
40    }  
41 }
```

2a



```
1 #forma 1 usando @@ para el contador de los coches
2 class Coche
3     @@num_coches = 0
4
5     attr_accessor :marca, :modelo, :matricula
6
7     def initialize(marca, modelo, matricula)
8         @marca = marca
9         @modelo = modelo
10        @matricula = matricula
11        @@num_coches += 1
12    end
13 end
14
15 #forma 2 usando @ para el contador de los coches
16 class Coche
17     @num_coches = 0
18
19     attr_accessor :marca, :modelo, :matricula
20
21     def initialize(marca, modelo, matricula)
22         @marca = marca
23         @modelo = modelo
24         @matricula = matricula
25         @num_coches += 1
26     end
27 end
```

2b



```
1 public String toString() {  
2     return "Marca: " + marca + ", Modelo: " + modelo + ", Matricula: " + matricula;  
3 }  
4 }
```

Atributos y Métodos - Relación de Ejercicios

Ismael Sallami Moreno

November 2024

1 Ejercicio 1

1.1 Parte a: Hacerlo en Java

Vamos a crear una clase `Coche` en Java. La clase tendrá atributos para la marca, el modelo y la matrícula del coche, y una variable de clase para contar el número total de coches construidos.

```
public class Coche {  
    private String marca;  
    private String modelo;  
    private String matricula;  
    private static int contadorCoches = 0;  
  
    public Coche(String marca, String modelo, String matricula) {  
        this.marca = marca;  
        this.modelo = modelo;  
        this.matricula = matricula;  
        contadorCoches++;  
    }  
  
    public static int getContadorCoches() {  
        return contadorCoches;  
    }  
  
    public String getEstado() {  
        return "Marca: " + marca + ", Modelo: " + modelo + ", Matrícula: " + matricula;  
    }  
  
    public static void main(String[] args) {  
        Coche coche1 = new Coche("Toyota", "Corolla", "ABC-123");  
        Coche coche2 = new Coche("Honda", "Civic", "XYZ-456");  
        System.out.println("Número de coches construidos: " + Coche.getContadorCoches());  
        System.out.println("Estado del coche 1: " + coche1.getEstado());  
        System.out.println("Estado del coche 2: " + coche2.getEstado());  
    }  
}
```

```
    }
}
```

¿Has tenido que usar la pseudovariable `this` en el constructor? Sí, es necesario usar `this` para diferenciar entre las variables de instancia y los parámetros del constructor que tienen el mismo nombre.

1.2 Parte b: Hacerlo en Ruby (Variable de Clase)

Vamos a crear una clase `Coche` en Ruby utilizando una variable de clase (`@@`) para el contador de coches.

```
class Coche
  @@contador_coches = 0

  def initialize(marca, modelo, matricula)
    @marca = marca
    @modelo = modelo
    @matricula = matricula
    @@contador_coches += 1
  end

  def self.contador_coches
    @@contador_coches
  end

  def estado
    "Marca: #{@marca}, Modelo: #{@modelo}, Matrícula: #{@matricula}"
  end
end

coche1 = Coche.new("Toyota", "Corolla", "ABC-123")
coche2 = Coche.new("Honda", "Civic", "XYZ-456")
puts "Número de coches construidos: #{Coche.contador_coches}"
puts "Estado del coche 1: #{coche1.estado}"
puts "Estado del coche 2: #{coche2.estado}"
```

1.3 Parte c: Hacerlo en Ruby (Atributo de Instancia)

Vamos a crear una clase `Coche` en Ruby utilizando un atributo de instancia (`@`) para el contador de coches.

```
class Coche
  @contador_coches = 0

  class << self
    attr_accessor :contador_coches
```

```

    end

    def initialize(marca, modelo, matricula)
        @marca = marca
        @modelo = modelo
        @matricula = matricula
        self.class.contador_coches += 1
    end

    def self.contador_coches
        @contador_coches
    end

    def estado
        "Marca: #{@marca}, Modelo: #{@modelo}, Matrícula: #{@matricula}"
    end
end

coche1 = Coche.new("Toyota", "Corolla", "ABC-123")
coche2 = Coche.new("Honda", "Civic", "XYZ-456")
puts "Número de coches construidos: #{Coche.contador_coches}"
puts "Estado del coche 1: #{coche1.estado}"
puts "Estado del coche 2: #{coche2.estado}"

```

2 Ejercicio 3

2.1 Pregunta

En la clase de los ejercicios anteriores, añadir un método que devuelva el número de coches que se han instanciado. Decidir si el método debe ser de instancia o de clase.

2.2 Parte a: Hacerlo en Java

Vamos a crear un método de clase que devuelva el número de instancias de la clase `Car`:

```

public class Car {
    private static int instanceCount = 0;
    private String model;

    public Car(String model) {
        this.model = model;
        instanceCount++;
    }
}

```

```

public static int getInstanceCount() {
    return instanceCount;
}

public static void main(String[] args) {
    new Car("Toyota");
    new Car("Honda");
    System.out.println("Número de coches instanciados: " + Car.getInstanceCount());
}
}

```

En este caso, el método `getInstanceCount()` es un método de clase, ya que devuelve el número total de instancias creadas.

2.3 Parte b: Hacerlo en Ruby

2.3.1 Versión 1

Primero, vamos a hacer una versión simple utilizando una variable de clase:

```

class Car
  @@instance_count = 0

  def initialize(model)
    @model = model
    @@instance_count += 1
  end

  def self.instance_count
    @@instance_count
  end
end

car1 = Car.new("Toyota")
car2 = Car.new("Honda")
puts "Número de coches instanciados: #{Car.instance_count}"

```

2.3.2 Versión 2

Ahora, vamos a hacer una versión que no utiliza variables de clase y en su lugar emplea una pseudovariable `self` para contar las instancias:

```

class Car
  @instance_count = 0

  class << self
    attr_accessor :instance_count

```

```

    end

    def initialize(model)
      @model = model
      self.class.instance_count += 1
    end

    def self.instance_count
      @instance_count
    end
  end

car1 = Car.new("Toyota")
car2 = Car.new("Honda")
puts "Número de coches instanciados: #{Car.instance_count}"

```

En ambas versiones, `instance_count` es un método de clase, ya que necesitamos contar todas las instancias de la clase `Car`.

3 Ejercicio 4

3.1 Pregunta

Pensar un ejemplo en el que sea necesario usar una variable de clase para almacenar una constante numérica. Implementar dicha clase y un ejemplo de uso donde se emplee dicha constante en una instrucción.

3.2 Parte a: Hacerlo en Java

Vamos a crear una clase en Java que utilice una variable de clase para almacenar una constante numérica:

```

public class ConstanteEjemplo {
  public static final double PI = 3.14159;

  public double calcularCircunferencia(double radio) {
    return 2 * PI * radio;
  }

  public static void main(String[] args) {
    ConstanteEjemplo ejemplo = new ConstanteEjemplo();
    double circunferencia = ejemplo.calcularCircunferencia(5);
    System.out.println("La circunferencia es: " + circunferencia);
  }
}

```

3.3 Parte b: Hacerlo en Ruby (Variable de Clase)

Vamos a hacerlo en Ruby utilizando una variable de clase:

```
class ConstanteEjemplo
  @@pi = 3.14159

  def calcular_circunferencia(radio)
    2 * @@pi * radio
  end
end

ejemplo = ConstanteEjemplo.new
circunferencia = ejemplo.calcular_circunferencia(5)
puts "La circunferencia es: #{circunferencia}"
```

3.4 Parte c: Hacerlo en Ruby (Atributo de Instancia)

Finalmente, vamos a hacerlo en Ruby considerando un atributo de instancia de la clase para la constante:

```
class ConstanteEjemplo
  def initialize
    @pi = 3.14159
  end

  def calcular_circunferencia(radio)
    2 * @pi * radio
  end
end

ejemplo = ConstanteEjemplo.new
circunferencia = ejemplo.calcular_circunferencia(5)
puts "La circunferencia es: #{circunferencia}"
```

4 Ejercicio 4

4.1 Parte a: Hacerlo en Java

Primero, vamos a crear una clase en Java que utilice una variable de clase para almacenar una constante numérica. Aquí tienes un ejemplo sencillo:

```
public class Ejemplo {
  public static final double CONST_NUMERICA = 3.14;

  public static void mostrarConstante() {
    System.out.println("La constante es: " + CONST_NUMERICA);
```

```

    }

    public static void main(String[] args) {
        Ejemplo.mostrarConstante();
    }
}

```

4.2 Parte b: Hacerlo en Ruby (Variable de Clase)

Ahora, vamos a ver cómo hacerlo en Ruby utilizando una variable de clase:

```

class Ejemplo
    @@const_numerica = 3.14

    def self.mostrar_constante
        puts "La constante es: #{@const_numerica}"
    end
end

Ejemplo.mostrar_constante

```

4.3 Parte c: Hacerlo en Ruby (Atributo de Instancia)

Finalmente, vamos a hacerlo en Ruby considerando un atributo de instancia de la clase:

```

class Ejemplo
    def initialize
        @const_numerica = 3.14
    end

    def mostrar_constante
        puts "La constante es: #{@const_numerica}"
    end
end

ejemplo = Ejemplo.new
ejemplo.mostrar_constante

```

5 Ejercicio 5

Para lograr la solución sin usar `instance_variable_get`, hemos hecho un pequeño ajuste a la clase para permitir el acceso al atributo privado de una manera controlada. Vamos a agregar un método `public` para obtener la capacidad de otro objeto.

5.1 Solución en Ruby

```
class CloudDrive
    def initialize(capacity)
        @capacity = capacity
    end

    private

    def get_capacity
        @capacity
    end

    public

    def set_from_other(other)
        @capacity = other.capacity
    end

    def set_from_other_v2(other)
        @capacity = other.get_capacity
    end

    # Método público para acceder a la capacidad de otro objeto
    def capacity
        @capacity
    end
end

# Crear instancias y probar los métodos
cd1 = CloudDrive.new(100)
cd2 = CloudDrive.new(200)
cd1.set_from_other(cd2)
puts cd1.capacity # Salida esperada: 200
cd1.set_from_other_v2(cd2)
puts cd1.capacity # Salida esperada: 200
```

5.2 Explicación

En este código:

- He añadido un método `public` llamado `capacity` que devuelve el valor de `@capacity`.
- Esto permite que el método `set_from_other` acceda a la capacidad de otro objeto sin necesidad de usar `instance_variable_get`.

De esta manera, el código cumple con el encapsulamiento y sigue siendo claro y mantenable.

6 Ejercicio 6

6.1 Pregunta

En Java, ¿se puede acceder desde un método de instancia a una variable de clase de la misma clase? ¿En qué circunstancias?

6.2 Respuesta

Sí, en Java se puede acceder desde un método de instancia a una variable de clase de la misma clase. Esto es posible porque las variables de clase (también conocidas como variables estáticas) son compartidas por todas las instancias de la clase. Aquí hay un ejemplo ilustrativo:

```
public class Ejemplo {  
    private static int variableDeClase = 10;  
  
    public int obtenerVariableDeClase() {  
        return variableDeClase;  
    }  
  
    public static void main(String[] args) {  
        Ejemplo instancia = new Ejemplo();  
        System.out.println("Variable de clase: " + instancia.obtenerVariableDeClase());  
    }  
}
```

En este ejemplo, el método de instancia `obtenerVariableDeClase()` puede acceder a la variable de clase `variableDeClase`.

7 Ejercicio 7

7.1 Pregunta

En Ruby, ¿se puede acceder desde un método de instancia a una variable de clase (@@) de la misma clase? ¿En qué circunstancias?

7.2 Respuesta

Sí, en Ruby se puede acceder desde un método de instancia a una variable de clase (@@) de la misma clase. Las variables de clase en Ruby son compartidas por todas las instancias de la clase. Aquí hay un ejemplo:

```

class Ejemplo
  @@variable_de_clase = 10

  def obtener_variable_de_clase
    @@variable_de_clase
  end
end

instancia = Ejemplo.new
puts "Variable de clase: #{instancia.obtener_variable_de_clase}"

```

En este ejemplo, el método de instancia `obtener_variable_de_clase` puede acceder a la variable de clase `@@variable_de_clase`.

8 Ejercicio 8

8.1 Pregunta

En Ruby, ¿se puede acceder desde un método de instancia a un atributo de instancia de la clase (@) de la misma clase? ¿En qué circunstancias? ¿Qué puede hacerse para conseguir el acceso?

8.2 Respuesta

Sí, en Ruby se puede acceder desde un método de instancia a un atributo de instancia de la misma clase. Los métodos de instancia tienen acceso a los atributos de instancia (variables prefijadas con @) de la misma instancia. Aquí hay un ejemplo:

```

class Ejemplo
  def initialize(valor)
    @atributo_de_instancia = valor
  end

  def obtener_atributo_de_instancia
    @atributo_de_instancia
  end
end

instancia = Ejemplo.new(10)
puts "Atributo de instancia: #{instancia.obtener_atributo_de_instancia}"

```

En este ejemplo, el método de instancia `obtener_atributo_de_instancia` puede acceder al atributo de instancia `@atributo_de_instancia`.

8.3 Explicación

Para acceder a un atributo de instancia desde otro método de instancia en la misma clase, simplemente se puede hacer referencia al atributo de instancia utilizando el prefijo `@`. No se necesitan métodos adicionales para este acceso básico. Sin embargo, si se requiere acceder a atributos de instancia desde fuera de la clase, se pueden definir métodos getter y setter públicos.

Ejercicios: Constructores, consultores y modificadores

1. Crear una clase Coche. Para cada coche se almacenará su marca, modelo, año de construcción y matrícula. Se desea poder instanciar coches de 2 maneras distintas:
 - a) Pasando como parámetros una marca, un modelo, un año de construcción y una matrícula.
 - b) Pasando como parámetros una marca, un modelo y un año de construcción (para coches que aún no están matriculados).
 - c) Hacerlo en Java
 - d) Si no lo has hecho así, hacerlo en Java de manera que desde uno de los constructores se reutilice el otro
 - e) Hacerlo en Ruby definiendo 2 métodos de clase para ello
 - f) Hacerlo en Ruby definiendo parámetros con valores por defecto
 - g) Hacerlo en Ruby de una manera distinta a las anteriores
 - h) ¿Qué valor has elegido para indicar que un coche no tiene matrícula aún?
2. Añade a la clase anterior un constructor de copia superficial
 - a) En Java
 - b) En Ruby
3. Crea una clase denominada Flota que tenga un único atributo que sea un vector de Coches.
Solo en Java:
 - a) Añadirle un constructor sin parámetros
 - b) Añadirle un método que permita añadir un Coche a la flota
 - c) Añadirle un constructor de copia profunda. ¿Cómo lo harías? ¿Coméntalo y consensúalo con los compañeros?
4. En la clase Flota, añadir un método que reciba un entero (i) y devuelva el Coche i-ésimo de la flota (considerar que el primero es el 0). ¡Ojo!, puedes recibir un entero (i) que esté fuera del rango del vector de Coches, realizar una implementación que no dé ningún error de ejecución si el entero (i) está fuera de rango. ¿Qué vas a hacer si eso ocurre?
 - a) Hacer una versión del consultor que devuelva una referencia al Coche
 - b) Hacer otra versión del consultor que devuelva una copia del Coche
5. Con respecto a la clase Coche del ejercicio 1:
 - a) En JAVA
 - Añadir un consultor que devuelva un String con el estado del objeto consultado
 - Añadir un modificador para actualizar el año de construcción del coche, solo debe permitir poner años mayores a 2000 e inferiores a 2025. Decidir qué hacer si se intenta poner un año fuera de ese rango.
 - Modificar los constructores del ejercicio 1 para no permitir años de construcción fuera del rango indicado en este ejercicio.
 - b) En RUBY
 - Repetir en este lenguaje los 3 puntos que has implementado para Java en este ejercicio.
 - Añadir un consultor implícito para la marca
 - Añadir un modificador implícito para el modelo
 - Añadir un consultor/modificador implícito para la matrícula

Ejercicio 1

Crear una clase **Coche**. Para cada coche se almacenará su marca, modelo, año de construcción y matrícula. Se desea poder instanciar coches de 2 maneras distintas:

- Pasando como parámetros una marca, un modelo, un año de construcción y una matrícula.
- Pasando como parámetros una marca, un modelo y un año de construcción (para coches que aún no están matriculados).
- Hacerlo en Java.

```
1 public class Coche {  
2     private String marca;  
3     private String modelo;  
4     private int anio;  
5     private String matricula;  
6  
7     // Constructor con matr cula  
8     public Coche(String marca, String modelo, int anio,  
9     String matricula) {  
10        this.marca = marca;  
11        this.modelo = modelo;  
12        this.anio = anio;  
13        this.matricula = matricula;  
14    }  
15  
16     // Constructor sin matr cula  
17     public Coche(String marca, String modelo, int anio) {  
18        this(marca, modelo, anio, "SIN_MATRICULA");  
19    }  
20}
```

- Si no lo has hecho así, hacerlo en Java de manera que desde uno de los constructores se reutilice el otro.

```
1 public Coche(String marca, String modelo, int anio) {  
2     this(marca, modelo, anio, "SIN_MATRICULA");  
3 }
```

- Hacerlo en Ruby definiendo 2 métodos de clase para ello.

```
1 class Coche  
2     attr_accessor :marca, :modelo, :anio, :matricula  
3  
4     def initialize(marca, modelo, anio, matricula = "SIN_MATRICULA")  
5         @marca = marca  
6         @modelo = modelo
```

```

7     @anio = anio
8     @matricula = matricula
9   end
10
11  def self.con_matricula(marca, modelo, anio, matricula)
12    new(marca, modelo, anio, matricula)
13  end
14
15  def self.sin_matricula(marca, modelo, anio)
16    new(marca, modelo, anio)
17  end
18 end

```

f) Hacerlo en Ruby definiendo parámetros con valores por defecto.

```

1 def initialize(marca="marca", modelo="modelo", anio="anio",
2   matricula = "SIN_MATRICULA")
3   @marca = marca
4   @modelo = modelo
5   @anio = anio
6   @matricula = matricula
7 end

```

g) Hacerlo en Ruby de una manera distinta a las anteriores.

```

1 class Coche
2   attr_accessor :marca, :modelo, :anio, :matricula
3
4   def initialize(marca, modelo, anio, matricula = nil)
5     @marca = marca
6     @modelo = modelo
7     @anio = anio
8     @matricula = matricula || "SIN_MATRICULA"
9   end
10 end

```

```

1 class Coche
2   def initialize(*args)
3     @marca = args[0]
4     @modelo = args[1]
5     @anio = args[2]
6     @matricula = args[3]
7   end
8 end

```

h) ¿Qué valor has elegido para indicar que un coche no tiene matrícula aún?

Para indicar que un coche no tiene matrícula aún, he elegido el valor "SIN_MATRICULA".

Ejercicio 2

Añade a la clase anterior un constructor de copia superficial.

- a) En Java.

```
1 public Coche(Coche otro) {  
2     this(otro.marca, otro.modelo, otro.anio, otro.matricula);  
3 }
```

- b) En Ruby.

```
1 class Coche  
2     def initialize(otro)  
3         @marca = otro.marca  
4         @modelo = otro.modelo  
5         @anio = otro.anio  
6         @matricula = otro.matricula  
7     end  
8 end
```

Ejercicio 3

Crea una clase denominada **Flota** que tenga un único atributo que sea un vector de **Coches**. Solo en Java:

- a) Añadirle un constructor sin parámetros.
- b) Añadirle un método que permita añadir un **Coche** a la flota.
- c) Añadirle un constructor de copia profunda. ¿Cómo lo harías? Coméntalo y consénsualo con los compañeros.

Solución

Listing 1: Clase Flota con clonación manual

```
1 import java.util.Vector;  
2  
3 class Coche {  
4     private String modelo;  
5     private int anio;  
6  
7     // Constructor parametrizado  
8     public Coche(String modelo, int anio) {  
9         this.modelo = modelo;  
10        this.anio = anio;  
11    }
```

```

12 // M todo de clonaci n manual
13 public Coche clonar() {
14     return new Coche(this.modelo, this.anio);
15 }
16
17 // M todos getters y setters si son necesarios
18 }
19
20 public class Flota {
21     private Vector<Coche> coches;
22
23     // Constructor sin par metros
24     public Flota() {
25         this.coches = new Vector<Coche>();
26     }
27
28     // M todo paraadir un Coche a la flota
29     public void addCoche(Coche coche) {
30         this.coches.add(coche);
31     }
32
33     // Constructor de copia profunda
34     public Flota(Flota otraFlota) {
35         this.coches = new Vector<Coche>();
36         for (Coche coche : otraFlota.coches) {
37             this.coches.add(coche.clonar());
38         }
39     }
40 }
41

```

Ejercicio 4

En la clase `Flota`, añadir un método que reciba un entero i y devuelva el Coche i -ésimo de la flota. Implementar esto en dos versiones: una que devuelva una referencia al Coche y otra que devuelva una copia del Coche.

Solución en Java

```

1 public class Flota {
2     private List<Coche> coches;
3
4     public Flota() {
5         coches = new ArrayList<>();
6     }
7
8     // Aadir coche a la flota
9     public void agregarCoche(Coche coche) {
10        coches.add(coche);
11    }
12
13     // Devuelve una referencia al Coche
14     public Coche obtenerCoche(int i) {

```

```

15     if (i >= 0 && i < coches.size()) {
16         return coches.get(i);
17     } else {
18         return null; // o lanzar una excepción
19     }
20 }
21
22 // Devuelve una copia del Coche
23 public Coche obtenerCocheCopia(int i) {
24     if (i >= 0 && i < coches.size()) {
25         return new Coche(coches.get(i));
26     } else {
27         return null; // o lanzar una excepción
28     }
29 }
30 }
```

Solución en Ruby

```

1 class Flota
2     def initialize
3         @coches = []
4     end
5
6     # Aadir coche a la flota
7     def agregar_coche(coche)
8         @coches << coche
9     end
10
11    # Devuelve una referencia al Coche
12    def obtener_coche(i)
13        if i >= 0 && i < @coches.size
14            @coches[i]
15        else
16            nil # o lanzar una excepción
17        end
18    end
19
20    # Devuelve una copia del Coche
21    def obtener_coche_copia(i)
22        if i >= 0 && i < @coches.size
23            Coche.new(@coches[i])
24        else
25            nil # o lanzar una excepción
26        end
27    end
28 end
```

Ejercicio 5

Con respecto a la clase **Coche** del ejercicio 1:

- Añadir un consultor que devuelva un String con el estado del objeto.

- Añadir un modificador para actualizar el año de construcción del coche, solo permitiendo años mayores a 2000 e inferiores a 2025.
- Modificar los constructores del ejercicio 1 para no permitir años de construcción fuera del rango indicado.

Solución en Java

```

1  public class Coche {
2      private String marca;
3      private String modelo;
4      private int anio;
5      private String matricula;
6
7      // Constructor con matr cula
8      public Coche(String marca, String modelo, int anio,
9          String matricula) {
10         if (anio < 2000 || anio > 2025) {
11             throw new IllegalArgumentException("El a o debe estar
12                 entre 2000 y 2025");
13         }
14         this.marca = marca;
15         this.modelo = modelo;
16         this.anio = anio;
17         this.matricula = matricula;
18     }
19
20     // Constructor sin matr cula
21     public Coche(String marca, String modelo, int anio) {
22         this(marca, modelo, anio, "SIN_MATRICULA");
23     }
24
25     // Consultor que devuelve el estado del objeto
26     @Override
27     public String toString() {
28         return "Coche[marca=" + marca + ", modelo=" +
29             modelo + ", anio=" + anio + ", matricula=" +
30             matricula + "]";
31     }
32
33     // Modificador para actualizar el a o de construcci n
34     public void setAnio(int anio) {
35         if (anio < 2000 || anio > 2025) {
36             throw new IllegalArgumentException("El a o debe
37                 estar entre 2000 y 2025");
38         }
39         this.anio = anio;
40     }
41 }
```

Solución en Ruby

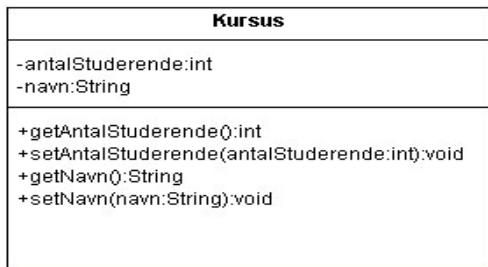
```

1  class Coche
```

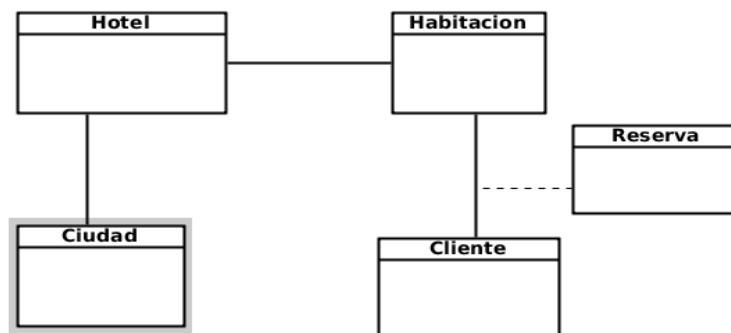
```
2     attr_accessor :marca
3     attr_reader :modelo, :matricula, :anio
4
5     def initialize(marca, modelo, anio, matricula = "SIN_MATRICULA")
6         if anio < 2000 || anio > 2025
7             raise "El año debe estar entre 2000 y 2025"
8         end
9         @marca = marca
10        @modelo = modelo
11        @anio = anio
12        @matricula = matricula
13    end
14
15    # Consultor que devuelve el estado del objeto
16    def to_s
17        "Coche[marca=#{@marca}, modelo=#{@modelo},
18        anio=#{@anio}, matricula=#{@matricula}]"
19    end
20
21    # Modificador para actualizar el año de construcción
22    def anio=(anio)
23        if anio < 2000 || anio > 2025
24            raise "El año debe estar entre 2000 y 2025"
25        end
26        @anio = anio
27    end
28 end
```

Ejercicios: Elementos de agrupación y diagramas estructurales

1. UML es un lenguaje “universal”. Escribe el código Java y Ruby correspondiente a la declaración de la siguiente clase en danés (incluyendo la declaración de atributos y la cabecera de los métodos).

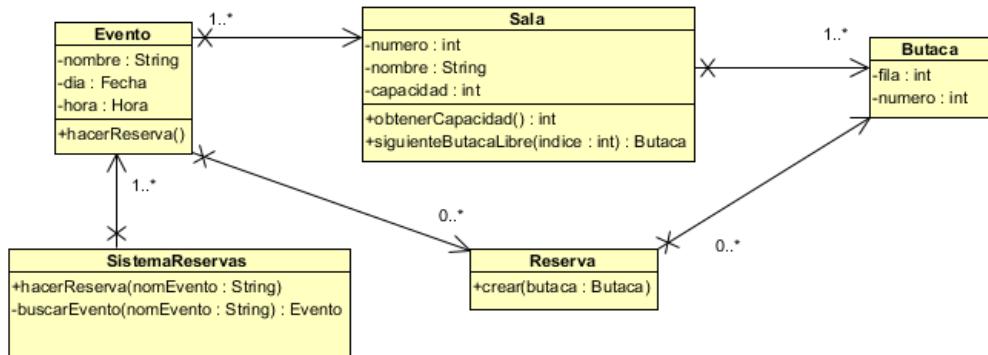


2. El siguiente diagrama de clases representa hoteles con sus habitaciones y las reservas hechas por sus clientes:



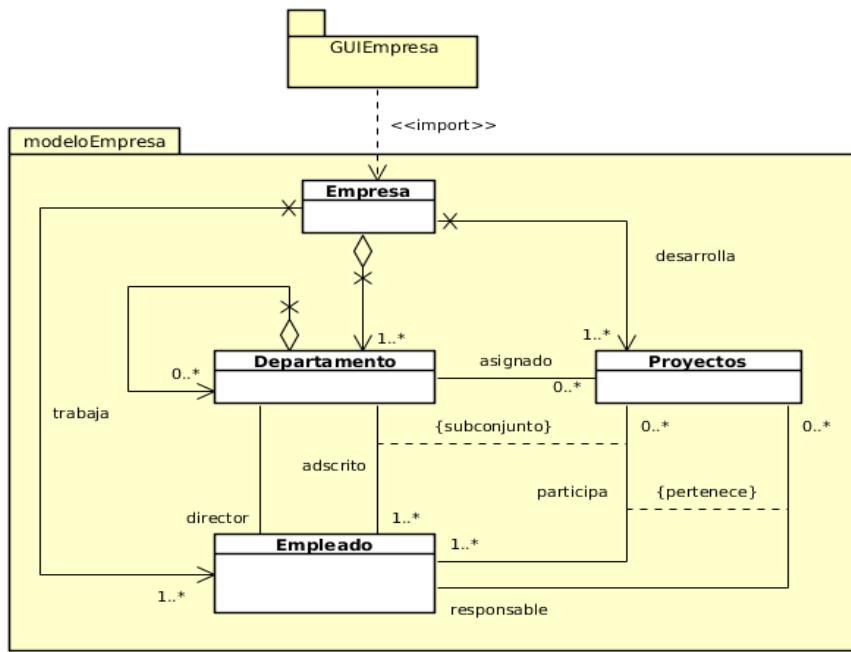
- a) Complétalo con los atributos de las clases y el nombre, roles, multiplicidad y navegabilidad de las asociaciones, haciendo las suposiciones que consideres oportunas.
b) Implementa en Java el resultado obtenido.
c) Implementa en Ruby el resultado obtenido.

3. A partir del siguiente diagrama de clases.



- a) ¿La asociación que existe entre Sala y Butaca podría ser una agregación? Y una composición?
b) Partiendo de una sala, ¿podría saberse para qué eventos está siendo usada?
c) Escribe el código Java correspondiente.
d) Escribe el código Ruby correspondiente.

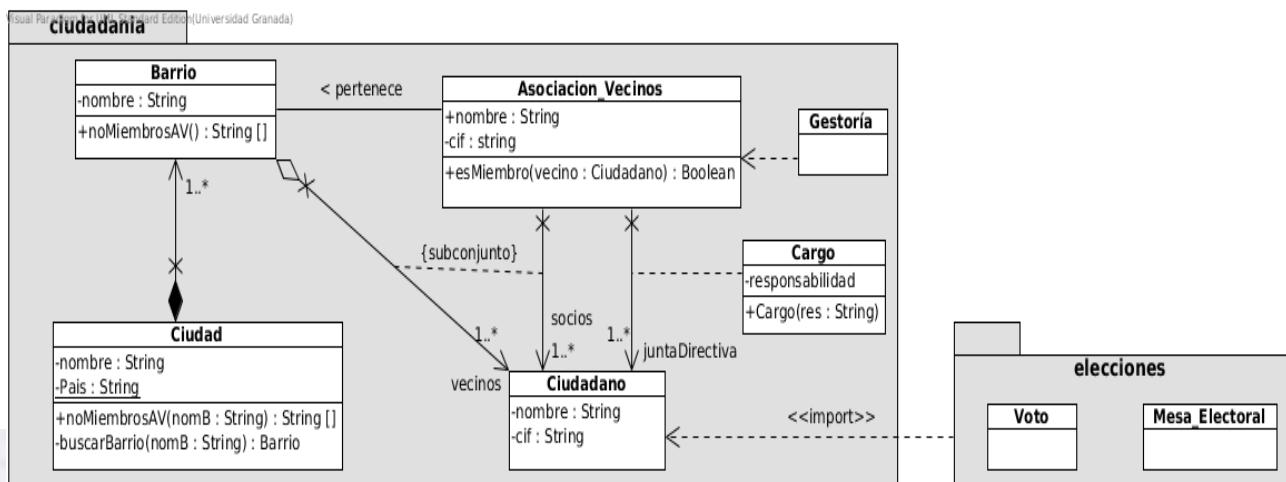
4. A partir del siguiente diagrama de clases, responde a las cuestiones plateadas:



- a) ¿Qué son GUIEmpresa y modeloEmpresa?
 - b) La relación que hay entre GUIEmpresa y Empresa ¿de qué tipo es?, es decir ¿qué significa <<import>> sobre ella?
 - c) Desde las clases de GUIEmpresa, en Java, se puede instanciar cualquier clase de modeloEmpresa? ¿Cómo?
 - d) ¿Qué representa la línea discontinua entre las asociaciones adscrito y participa y cuál es su significado en este diagrama?
 - e) Hay dos asociaciones entre Proyectos y Empleado, ¿qué representan?
 - f) ¿Por qué hay 0..* y no 1..* en la multiplicidad de la asociación “participa” entre Empleado y Proyectos? ¿Cómo expresarías en lenguaje natural lo que representa ese 0?

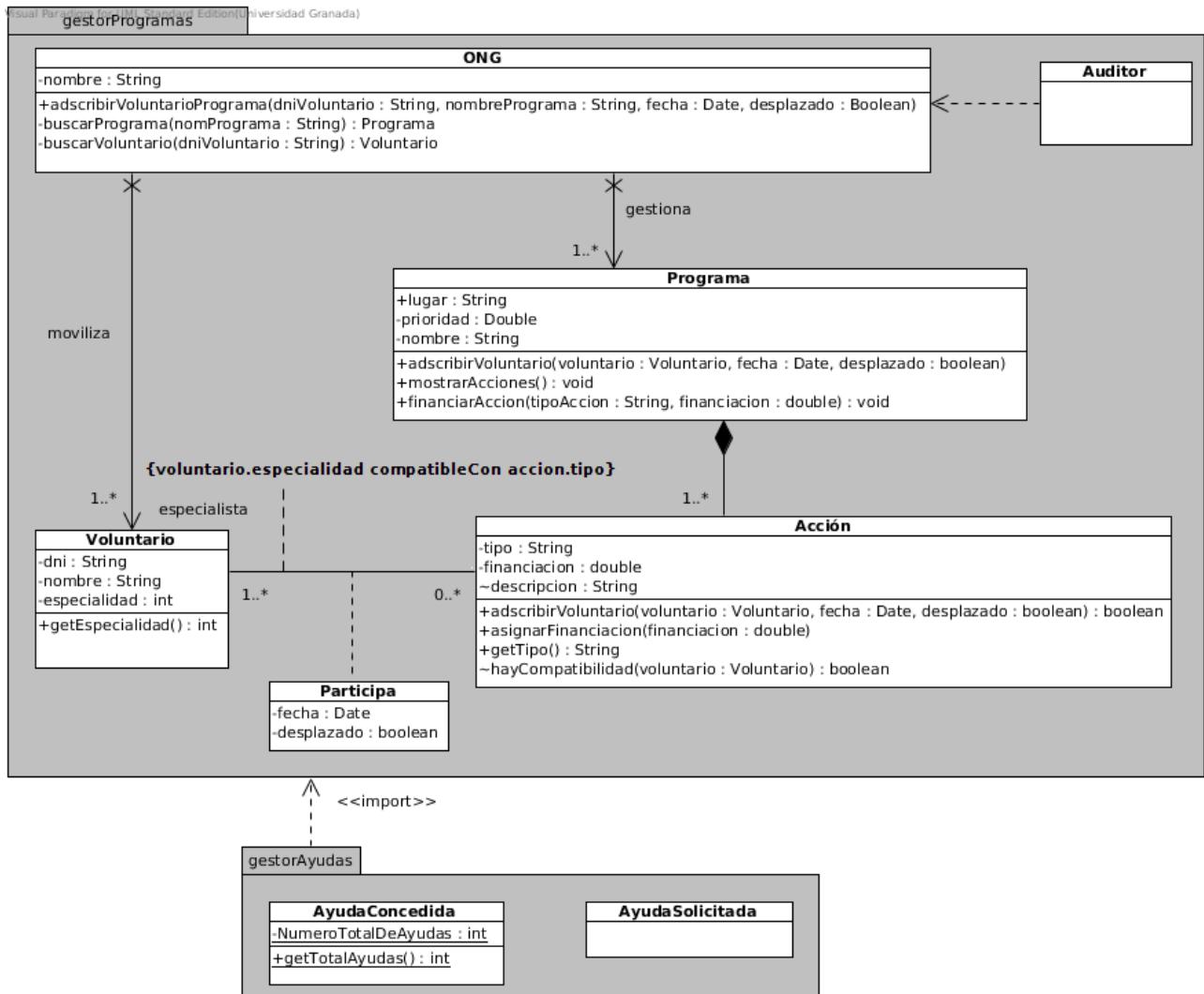
5. Partiendo del siguiente diagrama de clases de UML, resolver las cuestiones planteadas a continuación.

5. Partiendo del siguiente diagrama de clases de UML, resolver las cuestiones planteadas a continuación.



- a) ¿Desde un objeto de la clase Ciudadano puede saberse si es presidente de la asociación de vecinos de su barrio? ¿Por qué?
 - b) ¿De qué tipo es la relación entre Ciudad y Barrio? ¿qué significa?
 - c) ¿De qué tipo es la relación entre Barrio y Ciudadano? ¿qué significa? ¿podría ser de otro tipo?

- d) Si la junta directiva de una asociación de vecinos está formada por exactamente 6 ciudadanos, ¿cómo lo indicarías en el diagrama de clases?
- e) ¿Qué significa que la clase Cargo esté ligada a la asociación entre Asociacion_Vecinos y Ciudadano? ¿De qué otra forma podrías representar en un diagrama de clases las clases Asociacion_Vecinos, Cargo y Ciudadano?
- f) Define en Java los atributos de referencia de la clase Barrio.
6. Partiendo del siguiente diagrama de clases de UML, responde verdadero (V) o falso (F) a las cuestiones:



Desde la clase AyudaSolicitada se puede acceder a todos los elementos públicos del paquete GestorProgramas

Un voluntario puede participar en cualquier acción de un programa sin ningún tipo de restricción

El estado de un objeto de la clase Auditor viene determinado por el estado de un objeto de la clase ONG

Un voluntario podría participar en acciones de distintos programas

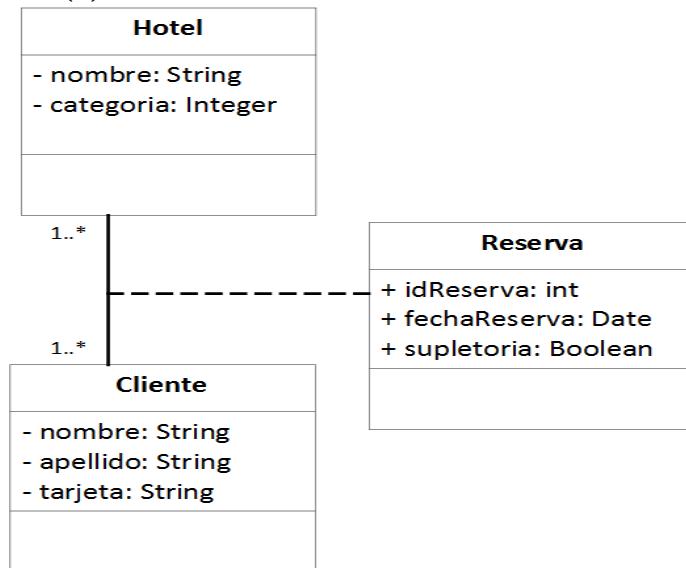
Un voluntario puede pertenecer a varias ONG

Cuando se define un objeto de la clase Acción, éste tiene que asociarse a un determinado objeto de la clase Programa

En una acción puede participar más de un voluntario como especialista

Desde un objeto de la clase ONG se puede llegar a conocer a todos los especialistas de una determinada acción en un programa	
El estado de un objeto Voluntario está exclusivamente determinado por su dni, nombre y especialidad	
Todos los métodos de la clase Acción pueden ser accedidos desde la clases AyudaConcedida	

7. Teniendo en cuenta el siguiente diagrama de clases, responde si las afirmaciones son verdaderas (V) o falsas (F)



Hotel tendrá dos atributos de referencia, uno relativo a las reservas y otro relativo a los clientes	
Reserva tendrá los atributos de referencia: <pre>private Hotel hotel; private Cliente cliente;</pre>	

Ejercicio 1

Enunciado:

UML es un lenguaje “universal”. Escribe el código Java y Ruby correspondiente a la declaración de la siguiente clase en danés (incluyendo la declaración de atributos y la cabecera de los métodos):

```
1 Kursus
2 - antalStuderende: int
3 - navn: String
4
5 + getAntalStuderende(): int
6 + setAntalStuderende(antalStuderende: int): void
7 + getNavn(): String
8 + setNavn(navn: String): void
```

Listing 1: Clase Kursus en Java (la implementación de los métodos no es necesaria)

Solución en Java:

```
1 public class Kursus {
2     private int antalStuderende;
3     private String navn;
4
5     public int getAntalStuderende() {
6         return antalStuderende;
7     }
8
9     public void setAntalStuderende(int antalStuderende) {
10         this.antalStuderende = antalStuderende;
11     }
12
13     public String getNavn() {
14         return navn;
15     }
16
17     public void setNavn(String navn) {
18         this.navn = navn;
19     }
20 }
```

Listing 2: Clase Kursus en Java

Solución en Ruby:

```
1 class Kursus
2   attr_accessor :antal_studerende, :navn
3
4   def initialize(antal_studerende, navn)
5     @antal_studerende = antal_studerende
6     @navn = navn
7   end
8 end
```

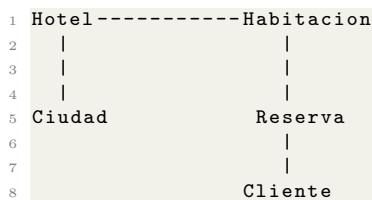
Listing 3: Clase Kursus en Ruby

Como podemos ver en el diagrama de clase, las funciones que aparecen son getters y setters, por lo que la función del constructor es para evitar dejar de lado la estructura adecuada y/o coherente de los ficheros de lenguaje acorde a la teoría.

Ejercicio 2

Enunciado:

El siguiente diagrama de clases representa hoteles con sus habitaciones y las reservas hechas por sus clientes:

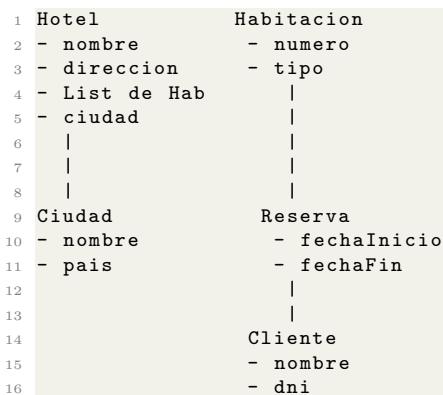


Listing 4: Diagrama de Clases

- Complétalo con los atributos de las clases y el nombre, roles, multiplicidad y navegabilidad de las asociaciones, haciendo las suposiciones que consideres oportunas.
- Implementa en Java el resultado obtenido.
- Implementa en Ruby el resultado obtenido.

Solución:

- Diagrama de Clases Completado:



Listing 5: Diagrama de Clases Completado

- Implementación en Java:

```

1 import java.util.ArrayList;
2 import java.util.List;
3
4 class Hotel {
5     private String nombre;
6     private String direccion;
7     private List<Habitacion> habitaciones;
8
9     public Hotel(String nombre, String direccion) {
10         this.nombre = nombre;
11         this.direccion = direccion;
12         this.habitaciones = new ArrayList<>();
13     }
14
15     // M todos getters y setters
16     //suponemos que la direccion contiene a la ciudad, en caso
17     //contrario un implementacion acorde seria:
18
19     private String nombre;
20     private String direccion;
21     private List<Habitacion> habitaciones;
22     private Ciudad ciudad;
23
24     public Hotel(String nombre, String direccion, Ciudad ciudad) {
25         this.nombre = nombre;
26         this.direccion = direccion;
27         this.habitaciones = new ArrayList<>();
28         this.ciudad=ciudad;
29         //deber amos de aadir lo nuevo al diagrama del apartado
30         a
31     }
32
33 class Habitacion {
34     private int numero;
35     private String tipo;
36
37     public Habitacion(int numero, String tipo) {
38         this.numero = numero;
39         this.tipo = tipo;
40     }
41
42     // M todos getters y setters
43 }
44
45 class Ciudad {
46     private String nombre;
47     private String pais;
48
49     public Ciudad(String nombre, String pais) {
50         this.nombre = nombre;
51         this.pais = pais;
52     }
53
54     // M todos getters y setters
55 }
```

```

56
57 class Reserva {
58     private String fechaInicio;
59     private String fechaFin;
60     private Cliente cliente; //no se incluye en el diagrama del
61     //apartado a porque podemos considerar que viene inclusive o se
62     //deduce de la relacion (LO MISMO PASAR A CON HOTEL)
63     //Consideraremos la opcion de asociar a una habitacion un hotel
64
65     public Reserva(String fechaInicio, String fechaFin, Cliente
66         cliente) {
67         this.fechaInicio = fechaInicio;
68         this.fechaFin = fechaFin;
69         this.cliente = cliente;
70     }
71     //tambien podemos considerar la opcion de asociar a una reserva
72     //un cliente
73
74     // M todos getters y setters
75 }
76
77 class Cliente {
78     private String nombre;
79     private String dni;
80
81     public Cliente(String nombre, String dni) {
82         this.nombre = nombre;
83         this.dni = dni;
84     }
85     // M todos getters y setters
86 }
```

Listing 6: Implementación en Java

c) Implementación en Ruby:

```

1 class Hotel
2   attr_accessor :nombre, :direccion, :habitaciones
3
4   def initialize(nombre, direccion)
5     @nombre = nombre
6     @direccion = direccion
7     @habitaciones = []
8   end
9 end
10
11 class Habitacion
12   attr_accessor :numero, :tipo
13
14   def initialize(numero, tipo)
15     @numero = numero
16     @tipo = tipo
17   end
18 end
19
20 class Ciudad
```

```

21 attr_accessor :nombre, :pais
22
23 def initialize(nombre, pais)
24   @nombre = nombre
25   @pais = pais
26 end
27 end
28
29 class Reserva
30   attr_accessor :fecha_inicio, :fecha_fin, :cliente
31
32   def initialize(fecha_inicio, fecha_fin, cliente)
33     @fecha_inicio = fecha_inicio
34     @fecha_fin = fecha_fin
35     @cliente = cliente
36   end
37 end
38
39 class Cliente
40   attr_accessor :nombre, :dni
41
42   def initialize(nombre, dni)
43     @nombre = nombre
44     @dni = dni
45   end
46 end

```

Listing 7: Implementación en Ruby

Uno de los porques de estas implementaciones puede ser, debido a la teoría **hotel** y **habitacion** no son padre e hijo, son hermanos, por lo que no debemos de incluirlos, mientras que **reserva** esta incluida con líneas discontinuas que sabemos que eso quiere decir que puede o no tenerla. Por último, podemos tener en cuenta que se incluye en **dirección** la **ciudad**.

Ejercicio 3

- a) ¿La asociación que existe entre Sala y Butaca podría ser una agregación?
¿Y una composición?
- b) Partiendo de una sala, ¿podría saberse para qué eventos está siendo usada?
- c) Escribe el código Java correspondiente.
- d) Escribe el código Ruby correspondiente.

Solución

a) Agregación vs Composición:

La relación entre Sala y Butaca podría ser una agregación o una composición dependiendo del contexto:

- **Agregación:** Si una butaca puede existir sin estar necesariamente asociada a una sala (por ejemplo, una butaca se puede mover de una sala a otra). Yo diría que si puede existir ya que si puede existir una butaca que no necesariamente este asociada a una sala. - **Composición:** Si una butaca no puede existir sin una sala (por ejemplo, las butacas se crean y se destruyen junto con la sala). Es que depende del contexto en este caso no, debido a lo explicado en el anterior punto, pero si se antepone esa condición en el contexto, entonces sí.

b) **Relación entre Sala y Eventos:**

Si una sala está reservada para varios eventos, se podría rastrear esta información a través de las reservas hechas en el sistema. Cada reserva tendría información sobre la sala y el evento asociado.

c) **Código en Java:**

```
1 import java.util.ArrayList;
2 import java.util.List;
3
4 // Clase Evento
5 class Evento {
6     private String nombre;
7     private Fecha dia;
8     private Hora hora;
9     private List<Sala> salas;// se puede usar listas o arrays
10    private List<Reserva> reservas;
11
12    public Evento(String nombre, Fecha dia, Hora hora) {
13        this.nombre = nombre;
14        this.dia = dia;
15        this.hora = hora;
16        this.salas = new ArrayList<>();
17        this.reservas = new ArrayList<>();
18    }
19
20    //constructor para la clase SistemaReservas
21    public Evento(string nombre){
22        Fecha dia = null;
23        Hora hora = null;
24        Evento(nombre,dia,hora);
25    }
26
27    //podemos aadir otros constructores con solo unos parametros,
28    //como puede ser pas ndole solo el tama o de alguna de las
29    //listas
30
31    public void hacerReserva() {
32        Butaca butacaLibre=null;
33        // Buscar la primera sala disponible con butacas libres
34        for (Sala sala : salas) {
35            butacaLibre = sala.siguienteButacaLibre();
36            if (butacaLibre != null) {
37                // Ocupar la butaca
38                butacaLibre.ocupar();
39                Reserva r = new Reserva(sala,this); //this hace
referencia a este evento
                    reservas.add(r);
                    break;

```

```

40         }
41     }
42
43     // Si no hay butacas libres, no se puede hacer la reserva
44     if (butacaLibre == null) System.out.println("No hay salas
45     con butacas disponibles.");
46   }
47
48   // M todos getters y setters
49 }
50
51 // Clase Sala
52 class Sala {
53   private String nombre;
54   private int numero;
55   private int capacidad;
56   private List<Butaca> butacas;
57
58   public Sala(String nombre, int numero, int capacidad) {
59     this.nombre = nombre;
60     this.numero = numero;
61     this.capacidad = capacidad;
62     this.butacas = new ArrayList<>();
63   }
64
65   public void addButaca(Butaca butaca) {
66     butacas.add(butaca);
67   }
68
69   public int obtenerCapacidad() {
70     return capacidad;
71   }
72
73   public Butaca siguienteButacaLibre() {
74     for (Butaca butaca : butacas) {
75       if (butaca.isEmpty()) {
76         return butaca;
77       }
78     }
79     return null; // No hay butacas libres
80   }
81
82   // M todos getters y setters
83 }
84
85 // Clase Butaca
86 class Butaca {
87   private int numero;
88   private int fila;
89   private boolean empty;
90
91   public Butaca(int numero, int fila) {
92     this.numero = numero;
93     this.fila = fila;
94     this.empty = true; // Por defecto est libre

```

```

96     }
97
98     public boolean isEmpty() {
99         return empty;
100    }
101
102    public void ocupar() {
103        this.empty = false;
104    }
105}
106
107 // Clase SistemaReservas
108 class SistemaReservas {
109     private List<Evento> eventos;
110
111     public SistemaReservas() {
112         this.eventos = new ArrayList<>();
113     }
114
115     public void hacerReserva(String nombreEvento) {
116         Evento nuevo = new Evento(nombreEvento);
117         eventos.add(nuevo);
118     }
119
120     public Evento buscarEvento(String nomEvento) {
121         for (Evento evento : eventos) {
122             if (evento.getNombre().equals(nomEvento)) {
123                 return evento;
124             }
125         }
126         return null; // No se encontró el evento
127     }
128 }
129
130 // Clase Reserva
131 class Reserva {
132     private Sala sala;
133     private Evento evento;
134
135     public Reserva(Sala sala, Evento evento) {
136         this.sala = sala;
137         this.evento = evento;
138     } // podemos pensar en la opción de incluir una clase cliente,
139     // que sea la persona a la que se le otorga la butaca
140
141     // M todos getters y setters
142 }
143
144 // Clases auxiliares
145 class Fecha {
146     // suponemos que ya está implementada
147 }
148 class Hora {
149     // suponemos que ya está implementada

```

```
150 }
```

Listing 8: Implementación en Java

Esto estaría implementado de manera que cada clase tendría su propio fichero.

d) Código en Ruby:

```
1 # Clase Evento
2 class Evento
3   attr_accessor :nombre, :dia, :hora, :salas, :reservas
4
5   def initialize(nombre, dia = nil, hora = nil)
6     @nombre = nombre
7     @dia = dia
8     @hora = hora
9     @salas = new Array
10    @reservas = new Array
11  end
12
13  def hacer_reserva
14    nueva_reserva = nil # Variable local para rastrear la reserva
15
16    @salas.each do |sala|
17      butaca_libre = sala.siguiente_butaca_libre
18      if butaca_libre
19        # Ocupar la butaca
20        butaca_libre.ocupar
21
22        # Crear y registrar la reserva
23        nueva_reserva = Reserva.new(sala, self)
24        @reservas.push(nueva_reserva)
25
26        # Detener la búsqueda una vez encontrada una sala
27        disponible
28        break
29      end
30
31      # Informar si no se encontró butaca libre
32      if nueva_reserva.nil?
33        puts "No hay salas con butacas disponibles."
34      else
35        puts "Reserva realizada con éxito."
36      end
37
38      # No retornar nada (implícito) para que sea void
39      nil
40    end
41  end
42
43 # Clase Sala
44 class Sala
45   attr_accessor :nombre, :numero, :capacidad, :butacas
46
47   def initialize(nombre, numero, capacidad)
48     @nombre = nombre
49     @numero = numero
```

```

50     @capacidad = capacidad
51     @butacas = Array.new
52   end
53
54   def add_butaca(butaca)
55     butacas.push(butaca)
56   end
57
58   def obtener_capacidad
59     @capacidad
60   end
61
62   def siguiente_butaca_libre
63     @butacas.find { |butaca| butaca.empty? } //creo que es as
64   end
65 end
66
67 # Clase Butaca
68 class Butaca
69   attr_accessor :numero, :fila, :empty
70
71   def initialize(numero, fila)
72     @numero = numero
73     @fila = fila
74     @empty = true # Por defecto est libre
75   end
76
77   def ocupar
78     @empty = false
79   end
80
81   def empty?
82     @empty
83   end
84 end
85
86 # Clase SistemaReservas
87 class SistemaReservas
88   attr_accessor :eventos
89
90   def initialize
91     @eventos = Array.new
92   end
93
94   def hacer_reserva(nombre_evento)
95     nuevo = Evento.new(nombre_evento)
96     @eventos.push(nuevo)
97   end
98
99   def buscar_evento(nombre_evento)
100    @eventos.find { |evento| evento.nombre == nombre_evento }
101  end
102 end
103
104 # Clase Reserva
105 class Reserva
106   attr_accessor :sala, :evento

```

```

107 #podemos incluir tambien la clase cliente para mas coherencia
108
109 def initialize(sala, evento)
110   @sala = sala
111   @evento = evento
112 end
113 end
114
115 # Clases auxiliares
116 class Fecha
117   # suponer que esta implementada
118 end
119
120 class Hora
121   # suponer que esta implementada
122 end

```

Listing 9: Implementación en Ruby

4. A partir del siguiente diagrama de clases, responde a las cuestiones planteadas:

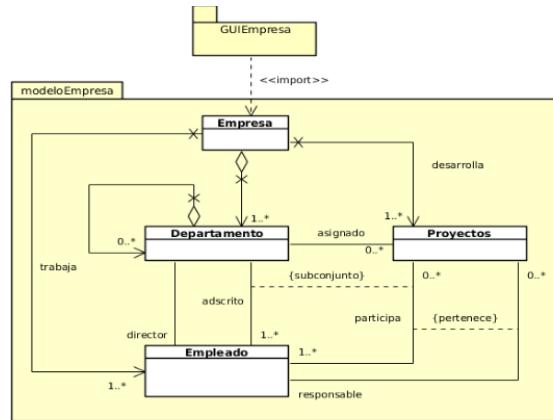


Figure 1: Diagrama de clases.

- ¿Qué son GUI Empresa y modelo Empresa?**
GUI Empresa: Se trata de otro paquete. modelo Empresa: del nombre de clase.
- La relación que hay entre GUI Empresa y Empresario ¿de qué tipo es?, es decir ¿qué significa <<import>> sobre ella?**
Quiere decir que se va a usar el paquete GUI Empresa en la clase modelo Empresa y para ello debe de importarlo.
- Desde las clases de GUI Empresa, en Java, se puede instanciar cualquier clase de modelo Empresa? ¿Cómo?**

- Sí, desde las clases de **GUIEmpresa** se puede instanciar cualquier clase de **modeloEmpresa**. Esto se hace importando las clases de **modeloEmpresa** en **GUIEmpresa** y creando instancias según sea necesario. Por ejemplo:

```
1 import modeloEmpresa.Empresa;
2 public class GUIEmpresa {
3     public static void main(String[] args) {
4         Empresa empresa = new Empresa(); // Operaciones con la
5             instancia de Empresa
6     }
6 }
```

Listing 10: Instanciación de clase de **modeloEmpresa** en **GUIEmpresa**

- d) **¿Qué representa la línea discontinua entre las asociaciones adscrito y participa y cuál es su significado en este diagrama?**

La línea discontinua representa una **dependencia** entre las asociaciones **adscrito** y **participa**. Significa que la asociación **participa** depende de la asociación **adscrito** en cierto sentido, probablemente indicando que un empleado debe estar adscrito a un departamento antes de poder participar en un proyecto.

- e) **Hay dos asociaciones entre Proyectos y Empleado, ¿qué representan?** - Las dos asociaciones entre **Proyectos** y **Empleado** representan distintas formas de interacción:

- **participa:** uno o varios empleados pueden participar en múltiples proyectos o en ninguno.
- Un empleado es responsable de múltiples proyectos o de ninguno.

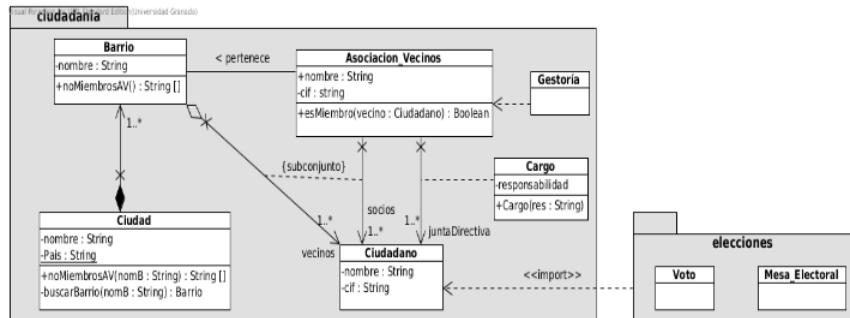
- f) **¿Por qué hay 0..* y no 1..* en la multiplicidad de la asociación “participa” entre Empleado y Proyectos? ¿Cómo expresarías en lenguaje natural lo que representa ese 0?**

El **0..*** en la multiplicidad de la asociación **participa** indica que un empleado puede no participar en ningún proyecto (0 proyectos) o puede participar en varios proyectos (* proyectos). Esto expresa la flexibilidad de que un empleado no está obligado a participar en proyectos. En lenguaje natural: "Un empleado puede no estar involucrado en ningún proyecto o puede estar involucrado en varios proyectos."

Ejercicio 5

Enunciado:

Partiendo del siguiente diagrama de clases de UML, resolver las cuestiones planteadas a continuación.



a) ¿Desde un objeto de la clase Ciudadano puede saberse si es presidente de la asociación de vecinos de su barrio? ¿Por qué?

Respuesta: No, desde un objeto de la clase Ciudadano no se puede saber directamente si es presidente de la asociación de vecinos de su barrio porque la clase Ciudadano no tiene un atributo o método que almacene o verifique esta información.

b) ¿De qué tipo es la relación entre Ciudad y Barrio? ¿qué significa?

Respuesta: La relación entre Ciudad y Barrio es una relación de composición, lo que significa que una ciudad está compuesta de varios barrios y que los barrios no pueden existir sin la ciudad.

c) ¿De qué tipo es la relación entre Barrio y Ciudadano? ¿qué significa? ¿podría ser de otro tipo?

Respuesta: La relación entre Barrio y Ciudadano es una relación de agrupación, lo que significa que un barrio puede tener varios ciudadanos y los ciudadanos pueden existir independientemente del barrio. Podría ser de otro tipo si se quisiera indicar, por ejemplo, que un ciudadano solo puede pertenecer a un barrio a la vez o en el caso de que se quiera decir que es de composición, pero en este caso si lo trasladamos a la vida real, no tiene sentido que un ciudadano no exista si no pertenece a un barrio.

d) Si la junta directiva de una asociación de vecinos está formada por exactamente 6 ciudadanos, ¿cómo lo indicarías en el diagrama de clases?

Respuesta: Se indicaría con una multiplicidad de 6 en la relación entre Ciudadano y Asociacion_Vecinos:

```

1     Asociacion_Vecinos
2     +nombre: String

```

```
3     +junta: Ciudadano[6]
```

- e) ¿Qué significa que la clase Cargo esté ligada a la asociación entre Asociacion_Vecinos y Ciudadano? ¿De qué otra forma podrías representar en un diagrama de clases las clases Asociacion_Vecinos, Cargo y Ciudadano?

Respuesta: Que la clase Cargo esté ligada a la asociación entre Asociacion_Vecinos y Ciudadano significa que los cargos están asignados a los ciudadanos en el contexto de su asociación vecinal, es decir, que un ciudadano puede tener una responsabilidad en la asociación de vecinos. Otra forma de representarlo sería mediante una relación de composición entre Asociacion_Vecinos y Cargo, y una relación de asociación entre Cargo y Ciudadano:

- Incluir las siguientes modificaciones:

```
Asociacion_Vecinos
+nombre: String
+cargos: Cargo[*]

Cargo
+nombre: String
+ocupante: Ciudadano
```

- O poner en el diagrama de clases la clase cargo en medio de las dos clases con unión de flechas continuas.

f) Define en Java los atributos de referencia de la clase Barrio.

Solución en Java:

```
1 class Barrio {
2     private String nombre;
3     private Ciudad ciudad;
4     private List<Ciudadano> ciudadanos;
5     private AsociacionVecinos asociacionVecinos;
6
7     // Constructor, getters y setters
8 }
```

Listing 11: Definición de la clase Barrio en Java

Ejercicio 6

Enunciado:

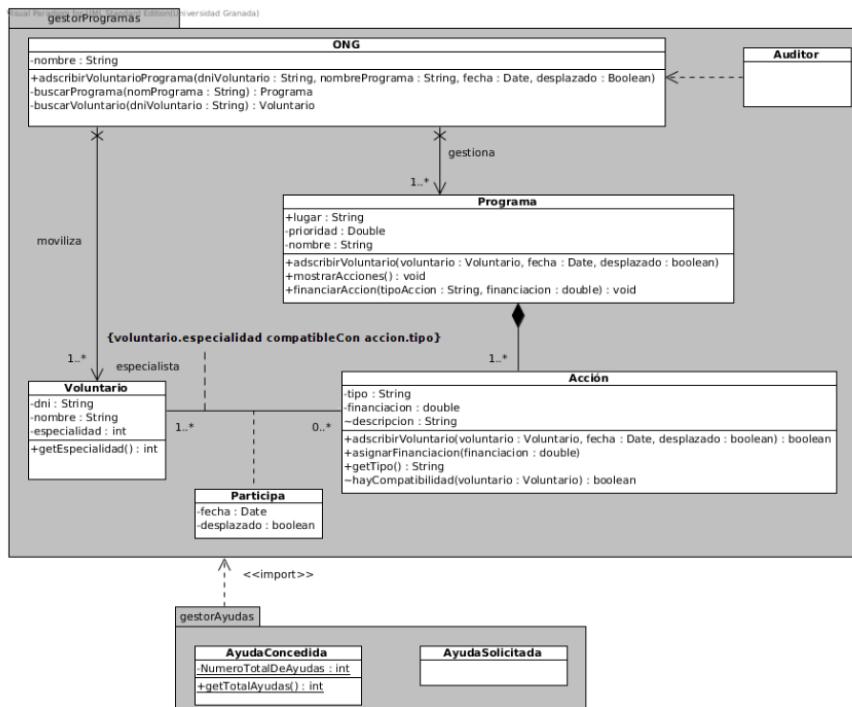


Figure 2: Caption

Partiendo del siguiente diagrama de clases de UML, responde verdadero (V) o falso (F) a las cuestiones:

Verdader o Falso

1. Desde la clase **AyudaSolicitada** se puede acceder a todos los elementos públicos del paquete **GestorProgramas**.
2. Un voluntario puede participar en cualquier acción de un programa sin ningún tipo de restricción.
3. El estado de un objeto de la clase **Auditor** viene determinado por el estado de un objeto de la clase **ONG**.
4. Un voluntario podría participar en acciones de distintos programas.
5. Un voluntario puede pertenecer a varias **ONG**.
6. Cuando se define un objeto de la clase **Acción**, éste tiene que asociarse a un determinado objeto de la clase **Programa**.
7. En una acción puede participar más de un voluntario como especialista.
8. Desde un objeto de la clase **ONG** se puede llegar a conocer a todos los especialistas de una determinada acción en un programa.
9. El estado de un objeto **Voluntario** está exclusivamente determinado por su **dni**, **nombre** y **especialidad**.
10. Todos los métodos de la clase **Acción** pueden ser accedidos desde la clase **AyudaConcedida**.

Respuestas y Justificaciones

- 1) Desde la clase **AyudaSolicitada** se puede acceder a todos los elementos públicos del paquete **GestorProgramas**: **Falso (F)**

Justificación: El acceso a elementos públicos de un paquete depende de la visibilidad y alcance definido en el mismo. No necesariamente todas las clases tienen acceso a todos los elementos públicos de otro paquete sin importar su relación.

- 2) Un voluntario puede participar en cualquier acción de un programa sin ningún tipo de restricción: **Falso (F)**

Justificación: Los voluntarios pueden tener restricciones basadas en su rol, habilidades o en las necesidades específicas del programa, por lo que no pueden participar en cualquier acción sin restricciones.

- 3) El estado de un objeto de la clase **Auditor** viene determinado por el estado de un objeto de la clase **ONG**: **Falso (F)**

Justificación: Aunque el Auditor puede estar asociado a la ONG, su estado puede ser independiente y determinado por sus propios atributos y métodos.

- 4) Un voluntario podría participar en acciones de distintos programas: **Verdadero (V)**

Justificación: Es posible que un voluntario tenga la capacidad de participar en múltiples programas, especialmente si tiene habilidades o experiencias relevantes para diferentes acciones.

- 5) Un voluntario puede pertenecer a varias ONG: **Verdadero (V)**

Justificación: No hay una restricción inherente que impida que un voluntario sea miembro de múltiples organizaciones no gubernamentales (ONG).

- 6) Cuando se define un objeto de la clase **Acción**, éste tiene que asociarse a un determinado objeto de la clase **Programa**: **Verdadero (V)**

Justificación: Según el diagrama de clases, una acción forma parte de un programa, por lo que debe estar asociada a un programa específico al ser definida.

- 7) En una acción puede participar más de un voluntario como especialista: **Verdadero (V)**

Justificación: Una acción puede requerir múltiples especialistas en diferentes áreas, permitiendo así la participación de varios voluntarios en la misma acción.

- 8) Desde un objeto de la clase **ONG** se puede llegar a conocer a todos los especialistas de una determinada acción en un programa: **Verdadero (V)**

Justificación: La clase ONG puede gestionar y conocer todos los especialistas involucrados en sus acciones y programas, teniendo acceso a esta información.

- 9) El estado de un objeto **Voluntario** está exclusivamente determinado por su dni, nombre y especialidad: **Falso (F)**

Justificación: El estado de un voluntario puede depender de más factores además de su dni, nombre y especialidad, como su disponibilidad, ubicación, entre otros atributos.

- 10) Todos los métodos de la clase **Acción** pueden ser accedidos desde la clase **AyudaConcedida**: **Falso (F)**

Justificación: El acceso a métodos depende de su visibilidad (público, protegido, privado). No todos los métodos de la clase Acción serán necesariamente accesibles desde AyudaConcedida.

Ejercicio 7

Enunciado:

Teniendo en cuenta el siguiente diagrama de clases, responde si las afirmaciones son verdaderas o falsas.

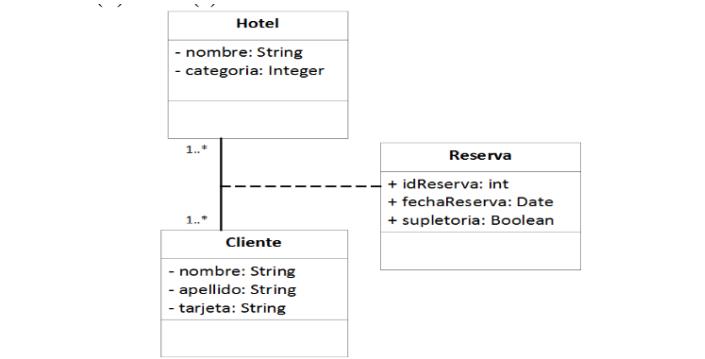


Figure 3: Diagrama de Clases UML para el sistema de reservas de hotel

Clases y Atributos

- Hotel
 - nombre: String
 - categoria: Integer
- Cliente
 - nombre: String
 - apellido: String
 - tarjeta: String
- Reserva
 - idReserva: int
 - fechaReserva: Date
 - supletoria: Boolean

Respuesta

Enunciado del Ejercicio

Resuelve el siguiente problema:

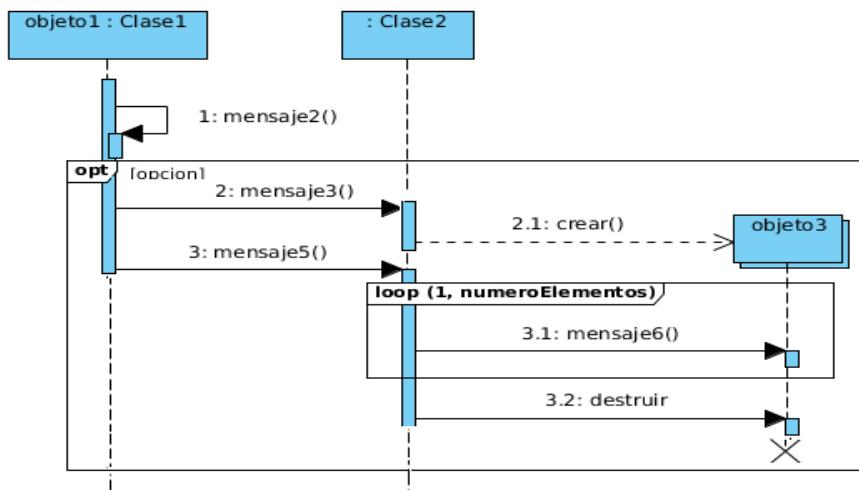
1. Hotel tendrá dos atributos de referencia, uno relativo a las reservas y otro relativo a los clientes.
2. Reserva tendrá los atributos de referencia:
 - private Hotel hotel;
 - private Cliente cliente;

Respuestas:

1. Falso.
2. Verdadero.

Ejercicios: Diagramas de interacción

1. Dado el siguiente Diagrama de secuencia, responde verdadero (V) o falso (F) a las siguientes cuestiones:



El envío del mensaje n.^o 1 (**mensaje2()**) es un envío de mensaje a **self** y además **recursivo**

El objeto **objeto3** es un objeto que vive sólo en esta operación

En la clase **Clase2** tienen que estar definidos los siguientes métodos: **mensaje3()**, **mensaje5()** y **mensaje6()**

El **fragmento combinado** tipo **loop** solo se puede usar para el envío de mensajes a multiobjetos, tal y como está representado en el ejemplo

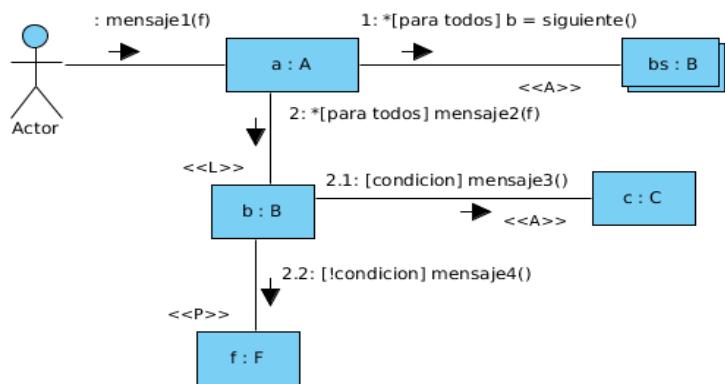
La representación del multiobjeto está mal, falta especificar la clase a la que pertenecen los objetos que forman ese multiobjeto

La numeración está mal, los envíos de mensaje 3.1 y 3.2 deberían ser 2.2 y 2.3

El siguiente código Ruby es correcto:

```
class Clase2
    def mensaje5
        objeto3.each [ |obj| obj.mensaje6() ]
        objeto3=nil
    end
end
```

2. Dado el siguiente Diagrama de comunicación, responde verdadero (V) o falso (F) a las siguientes cuestiones



El enlace o canal de comunicación estereotipado como <<P>> no puede ser de ese tipo ya que el objeto **f:F** no ha entrado como parámetro a la operación

La estructura de control usada en los envíos de mensajes números **2.1** y **2.2** es la estructura **if (condicion) {...} else {...}**

En la clase B debe estar implementado el método **siguiente()** para poder responder al envío de mensaje número 1

A los envíos de mensaje 2.1 y 2.2 les falta ***[para todos]**

El siguiente código Java es correcto:

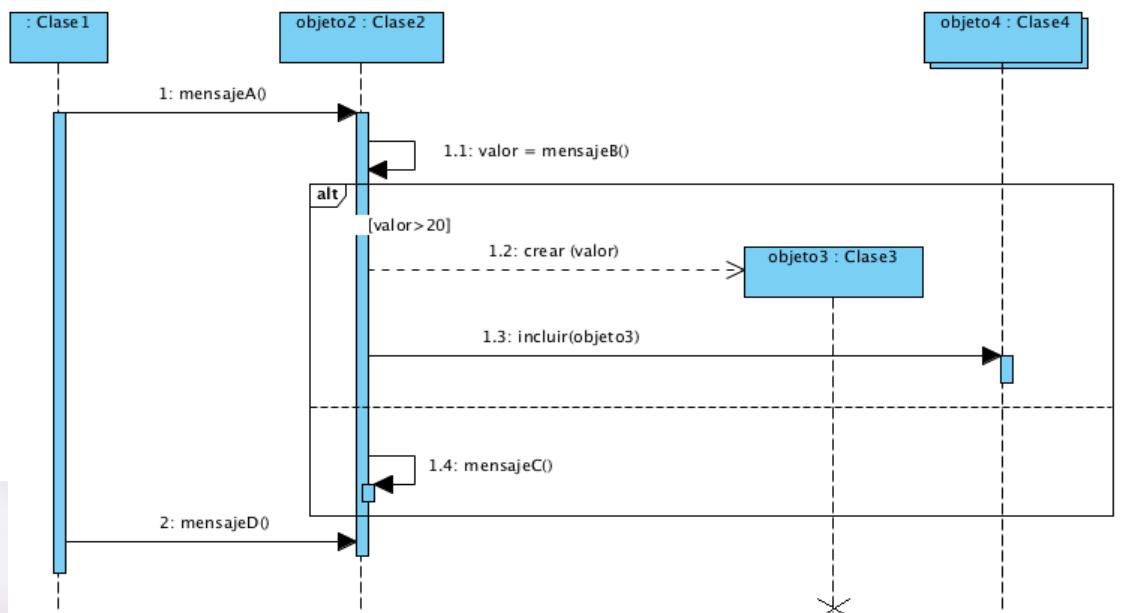
```

public class A {
    public void mensaje1(F f) {
        for(B b : bs) {
            b.mensaje2(f); } }
}

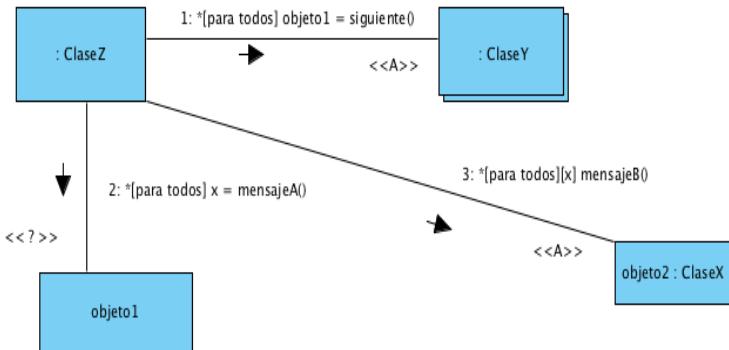
```

3. Dados los siguientes diagramas de secuencia y comunicación, implementarlos tanto en Java como en Ruby

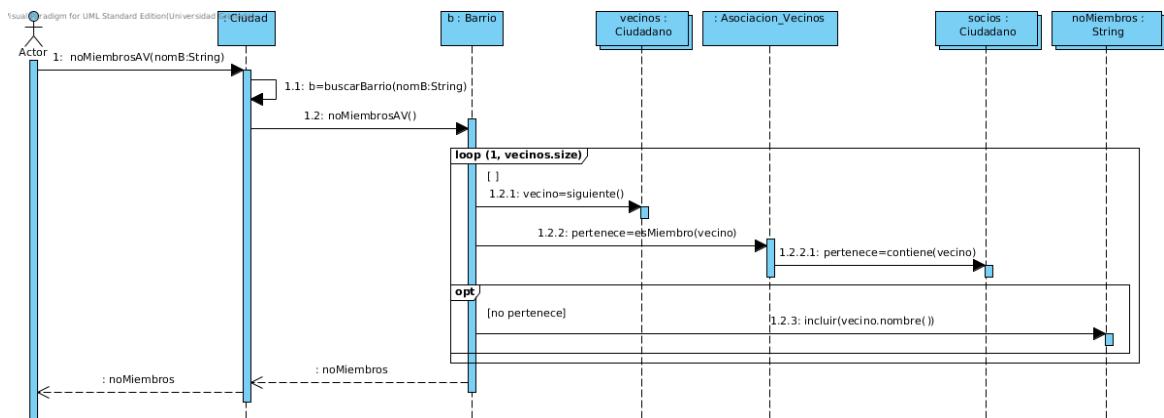
a)



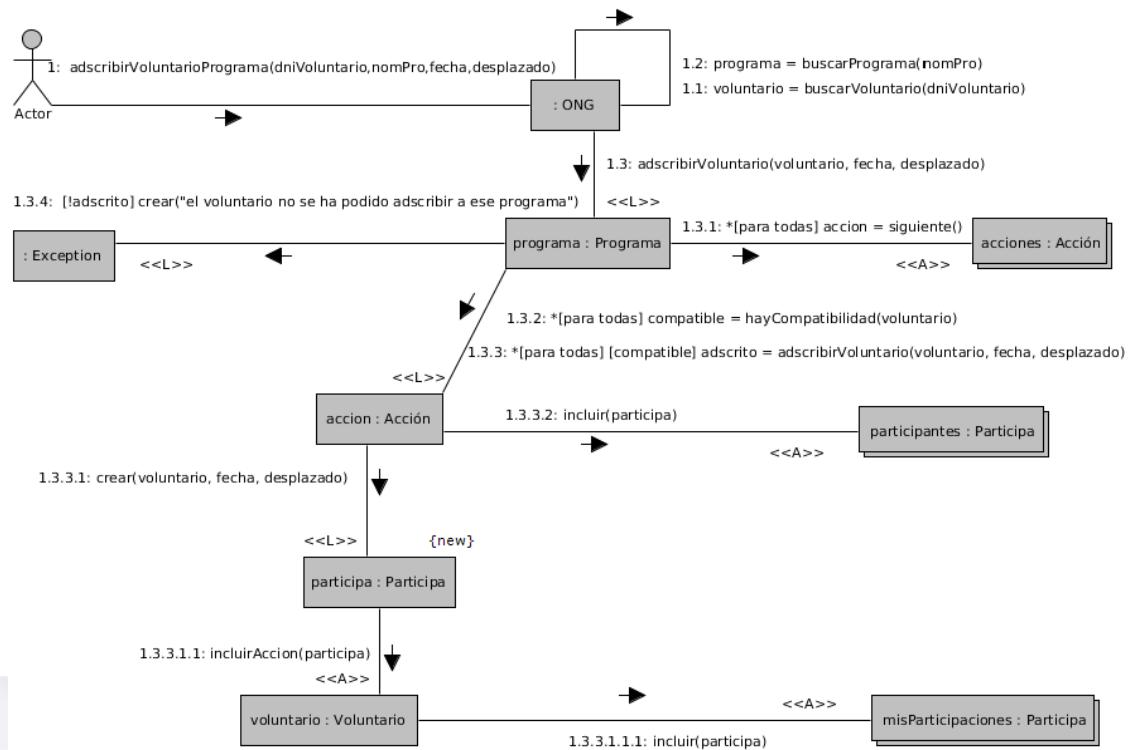
b)

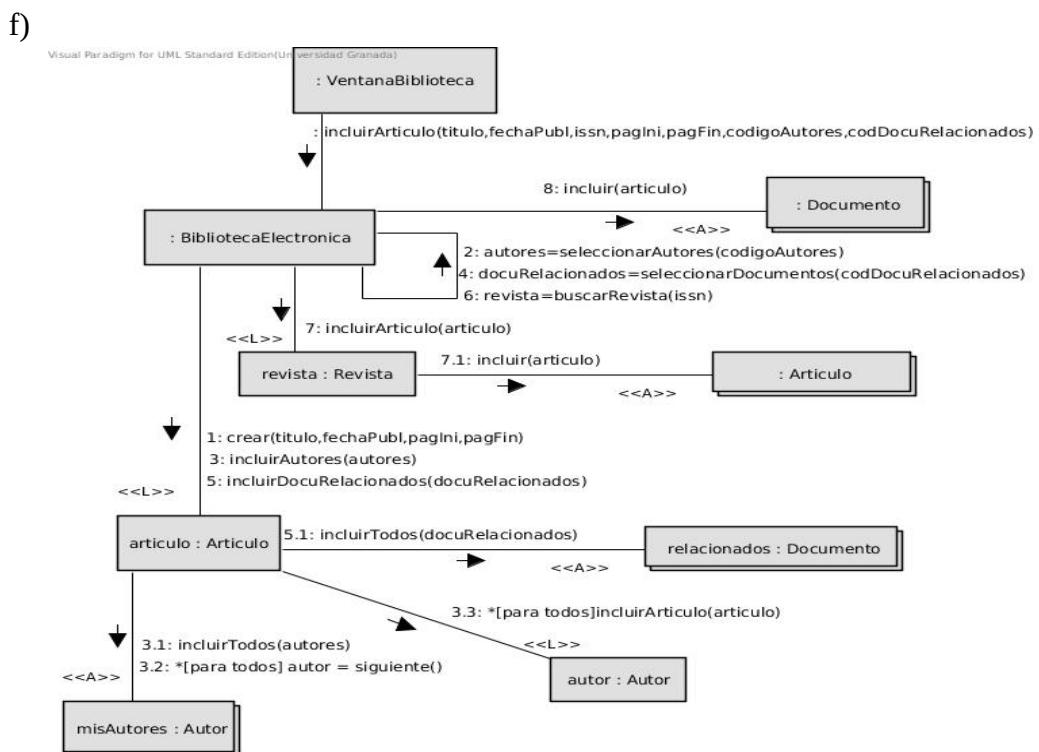
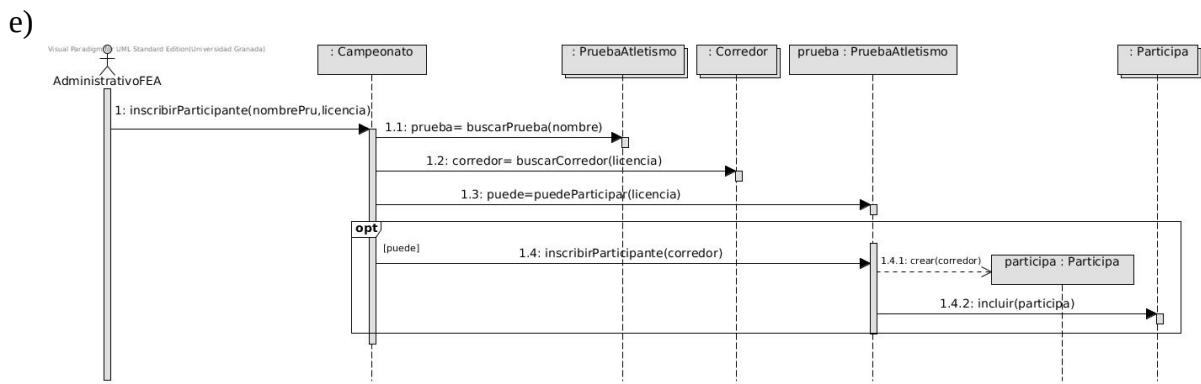


c)



d)





Ejercicio 1

Dado el siguiente Diagrama de secuencia, responde verdadero (V) o falso (F) a las siguientes cuestiones:

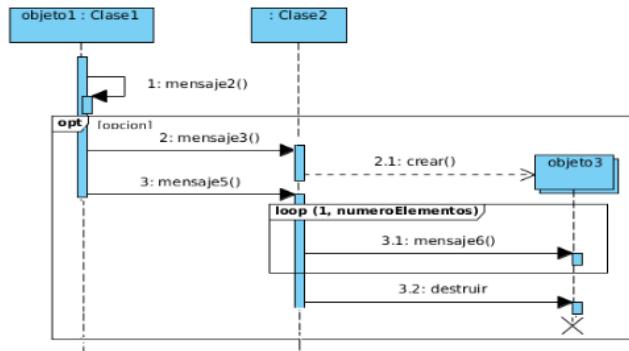


Figure 1: Diagrama de secuencia

- El envío del mensaje n.^o 1 (`mensaje2()`) es un envío de mensaje a `self` y además recursivo.
- El objeto `objeto3` es un objeto que vive sólo en esta operación.
- En la clase `Clase2` tienen que estar definidos los siguientes métodos: `mensaje3()`, `mensaje5()` y `mensaje6()`.
- El fragmento combinado tipo `loop` solo se puede usar para el envío de mensajes a multiobjetos, tal y como está representado en el ejemplo.
- La representación del multiobjeto está mal, falta especificar la clase a la que pertenecen los objetos que forman ese multiobjeto.
- La numeración está mal, los envíos de mensaje 3.1 y 3.2 deberían ser 2.2 y 2.3.
- El siguiente código Ruby es correcto:

```
1 class Clase2
2   def mensaje5
3     objeto3.each {|obj| obj.mensaje6}
4     objeto3=nil
5   end
6 end
```

Respuestas

- **Verdadero (V):** El envío del mensaje n.^o 1 (`mensaje2()`) es un envío de mensaje a `self` y además recursivo.
- **Falso (F):** El objeto `objeto3` no vive sólo en esta operación, ya que aparece en otros puntos del diagrama.
- **Verdadero (V):** En la clase `Clase2` deben estar definidos los métodos `mensaje3()`, `mensaje5()` y `mensaje6()`.
- **Falso (F):** El fragmento combinado tipo `loop` puede usarse para el envío de mensajes a cualquier tipo de objeto, no solo a multiobjetos.
- **Verdadero (V):** La representación del multiobjeto está mal, falta especificar la clase a la que pertenecen los objetos que forman ese multiobjeto.
- **Verdadero (V):** La numeración está mal, los envíos de mensaje 3.1 y 3.2 deberían ser 2.2 y 2.3.
- **Verdadero (V):** El código Ruby proporcionado es correcto.

Ejercicio 2

Dado el siguiente Diagrama de comunicación, responde verdadero (V) o falso (F) a las siguientes cuestiones:

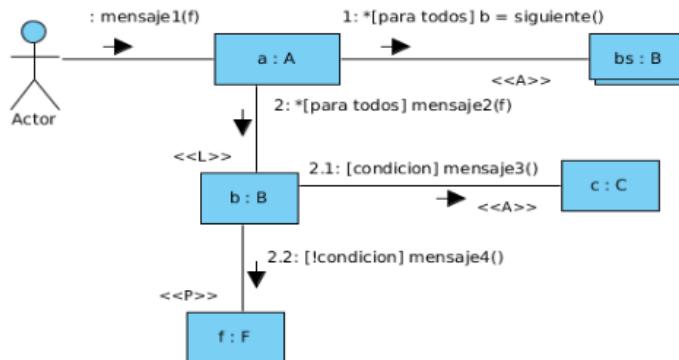


Figure 2: Diagrama de comunicación

- El enlace o canal de comunicación estereotipado como `<<P>>` no puede ser de ese tipo ya que el objeto `f:F` no ha entrado como parámetro a la operación.

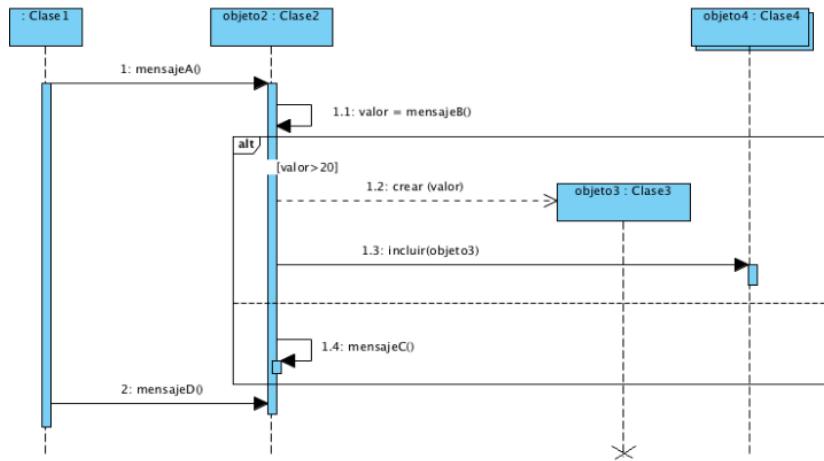
- La estructura de control usada en los envíos de mensajes números 2.1 y 2.2 es la estructura `if (condicion) ... else`
- En la clase B debe estar implementado el método `siguiente()` para poder responder al envío de mensaje número 1.
- A los envíos de mensaje 2.1 y 2.2 les falta `*[para todos]`.
- El siguiente código Java es correcto:

```
1 public class A {  
2     public void mensaje1(F f){  
3         for(B b : bs){  
4             b.mensaje2(f);  
5         }  
6     }  
7 }
```

Respuestas

- **Falso (F):** El enlace o canal de comunicación estereotipado como `<>P>>` no puede ser de ese tipo ya que el objeto `f:F` no ha entrado como parámetro a la operación.
- **Verdadero (V):** La estructura de control usada en los envíos de mensajes números 2.1 y 2.2 es la estructura `if (condicion) ... else`
- **Verdadero (V):** En la clase B debe estar implementado el método `siguiente()` para poder responder al envío de mensaje número 1.
- **Falso (F):** A los envíos de mensaje 2.1 y 2.2 les falta `*[para todos]`.
- **Verdadero (V):** El siguiente código Java es correcto:

Ejercicio 3: Implementar códigos en Java y en Ruby



Java

```

1 public class Clase1{
2     public Tipo mensajeA(){
3         int valor = objeto2.mensajeB();
4         if(valor>20){
5             Clase3 objeto3 = new Clase3(valor);
6             objeto4.incluir(objeto3);
7         }
8         else{
9             objeto2.mensajeC();
10        }
11    }
12    public Tipo mensajeD();
13}
14
//Es lo que podemos deducir con la informaci n que se nos proporciona.
  
```

Ruby

```

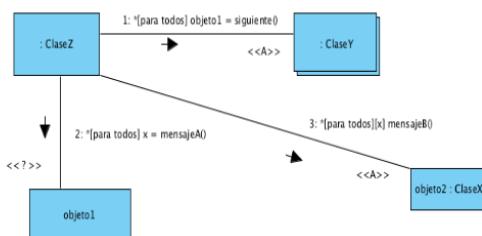
1 class Clase1
2   def mensajeA
3     int valor = objeto.mensajeB();
4     if valor>20
5       objeto3 = Clase3.new(valor);
6       objeto4.incluir(objeto3);
  
```

```

7         else
8             objeto2.mensajeC();
9         end
10    end
11  def mensajeD
12 end

```

b)



Java

```

1 public class ClaseZ{
2     //no recibe entrada por lo que denotamos al metodo como
3     //nombreMetodo
4     public Tipo nombreMetodo{
5         for(ClaseY objeto : ElementosDeLaClaseY){
6             tipo x = objeto.mensajeA();
7             if(x){
8                 objeto2.mensajeB();
9             }
10        }
11    }
12 }

```

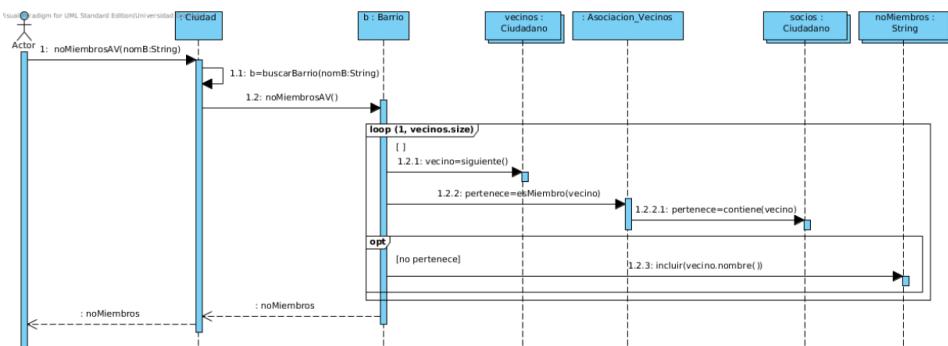
Ruby

```

# C digo Ruby para el diagrama de secuencia
2 class ClaseZ
3   def nombreMetodo
4     objetoY.each do |objeto1|
5       x = objeto1.mensajeA
6       if x
7         objeto2.mensajeB
8       end
9     end
10   end
11 end

```

c)



Java

```

1 public class Ciudad{
2     public String noMiembrosAV(String nomB){
3         Barrio b = buscarBarrio(nomB);
4         String noMiembros = b.noMiembrosAV();
5         return noMiembros;
6     }
7 }
8
9 public class Barrio{
10    public String noMiembrosAV(){
11        for(int i=1;i<vecinos.size();i++){
12            Ciudadano vecino = vecinos[i];
13            boolean pertenece = Asociacion_Vecinos.esMiembro(vecino);
14            if(!pertenece){
15                noMiembros.incluir(vecino.getNombre());
16            }
17        }
18        return noMiembros;
19    }
20 }
21
22 public class Asociacion_Vecinos{
23     public boolean esMiembro(Vecino vecino){
24         boolean pertenece;
25         pertenece = socios.contiene(vecino);
26         return pertenece;
27     }
28 }

```

Ruby

```

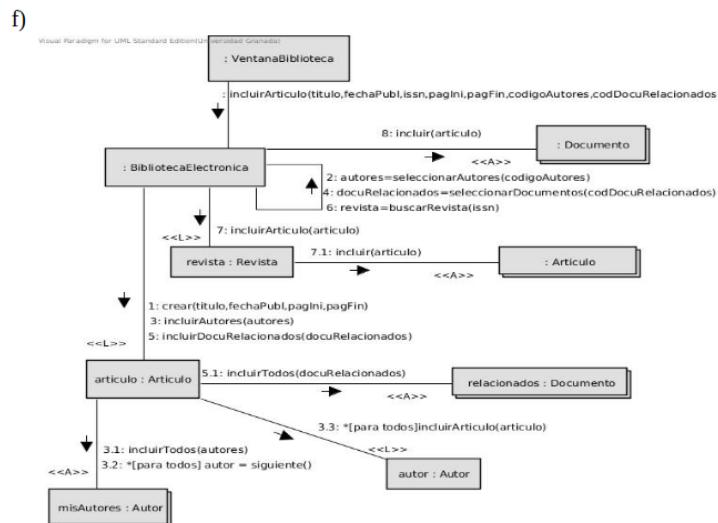
1 class Ciudad
2     def noMiembrosAV(nomB)
3         b = buscarBarrio(nomB)
4         noMiembros = b.noMiembrosAV

```

```

5         noMiembros
6     end
7 end
8
9 class Barrio
10    #suponemos que noMiembros ya esta declarado
11    def noMiembrosAV
12        for i in 1...vecinos.size
13            vecino = vecinos[i]
14            pertenece = Asociacion_Vecinos.esMiembro(vecino)
15            unless pertenece
16                @noMiembros.incluir(vecino.nombre)
17            end
18        end
19        @noMiembros
20    end
21 end
22
23 class Asociacion_Vecinos
24    def esMiembro(vecino)
25        pertenece = socios.contiene(vecino)
26        pertenece
27    end
28 end

```



Java

```

1 //en la cabecera falta el tipo de cada atributo
2 //suponemos que los objetos no est n creados debido a que no
   tenemos el diagrama de clases, es mas logico que ya esten

```

```

    declarados
3  public class BibliotecaElectronica{
4
5      public void incluirArticulo(titulo,fechaPubl,issn,pagIni,pagFin
6          ,codigoAutores,codDocuRelacionados){
7          Articulo articulo = new Articulo(titulo, fechaPubl,pagIni,
8          pagFin);
9          Autor autores = seleccionarAutores(codigoAutores);
10         articulo.incluirAutores(autores);
11         Documento [] docuRelacionados = seleccionarDocumentos(
12         codDocRelacionados);
13         articulo.incluirDocuRelacionados(docuRelacionados);
14         Revista revista = buscarRevista(issn);
15         revista.incluirArticulo(articulo);
16         Documento.incluir(articulo);
17     }
18 }
19 public class Articulo{
20     public void incluirAutores(Autor autores){
21         misAutores.incluirTodos(autores);
22         for(Autor autor : autores){
23             autor.incluirArticulos(articulo);
24             //suponemos que articulo esta en la parte privada de la
25             clase.
26         }
27     }
28     public void IncluirDocuRelacionados(Documento []
29         docuRelacionados){
30         relacionados.incluirTodos(docuRelacionados);
31     }
32 }
33 public class Revista{
34     public void incluirArticulo(Articulo articulo){
35         articulo.incluir(articulo);
36     }

```

Ruby

```

1 class BibliotecaElectronica
2     def incluir_articulo(titulo, fecha_publ, issn, pag_ini, pag_fin,
3         codigo_autores, cod_docu_relacionados)
4         articulo = Articulo.new(titulo, fecha_publ, pag_ini, pag_fin)
5         autores = seleccionar_autores(codigo_autores)
6         articulo.incluir_autores(autores)
7         docu_relacionados = seleccionar_documentos(
8             cod_docu_relacionados)
9         articulo.incluir_docu_relacionados(docu_relacionados)
10        revista = buscar_revista(issn)
11        revista.incluir_articulo(articulo)
12        Documento.incluir(articulo)
13    end

```

```
12 end
13
14 class Articulo
15   def incluir_autores(autores)
16     mis_autores.incluir.todos(autores)
17     autores.each do |autor|
18       autor.incluir_articulos(@articulo)
19       # suponemos que @articulo est en la parte privada de la
20       # clase.
21     end
22   end
23   def incluir_docu_relacionados(docu_relacionados)
24     relacionados.incluir.todos(docu_relacionados)
25   end
26 end
27
28 class Revista
29   def incluir_articulo(articulo)
30     Articulo.incluir(articulo)
31   end
32 end
```

2.3. Relación 3

Ejercicios: Herencia

1. Dada la siguiente clase en Java

```
public class Vertebrado
{
    ...
    public ArrayList<String> partesDelAbdomen();
    public void desplazarse();
    public String comunicarse (Vertebrado vertebrado);
    protected Vertebrado obtenerCopia();
}
```

Indica con una cruz en la casilla correspondiente según si la declaración de los siguientes métodos de la clase Mamifero, subclase de Vertebrado, sobrecargan (overloading) o redefinen (overriding) a los de la clase Vertebrado:

	Sobrec.	Redef.
public ArrayList<String> partesDelAbdomen();		
public void desplazarse(Modo m);		
public String comunicarse(Vertebrado vertebrado);		
public String comunicarse(Mamifero mamifero);		
public Vertebrado obtenerCopia();		
protected Mamifero obtenerCopia();		

2. A partir de las siguientes clases, implementa la clase EstudianteInformatica que hereda de Estudiante. Debe tener una nueva variable de instancia que es una colección de Strings con los dispositivos que utiliza: como por ejemplo PC, tablet, smartphone. Esa variable debe poder consultarse. También debe heredar el método estudiar(), pero modificando su implementación para que además de estudiar como los otros alumnos, diga que estudia con el último dispositivo de su colección. No olvides implementar su constructor. Implementa este mismo problema en Ruby.

```
public class Persona {
    protected String nombre ;
    public Persona(String nom) { this.setNombre(nom); }
    protected String getNombre() { return this.nombre; }
    public String hablar(){ return 'bla bla bla';}
}
```

```

3. public class Estudiante extends Persona {
    public String carrera;
    public int curso;

    public Estudiante (String nom, String carr, int cur) {
        super(nom);
        carrera = carr;
        curso = cur;
    }
    public void estudiar() { System.out.println ('estudiando'); }
}

```

4. Dada la siguiente clase abstracta en Java,

```

abstract class Transporte {
    protected String marca;
    protected String getMarca () { return this.marca; }
    protected void setMarca (String marc) { this.marca=marc; }
    public abstract String hacerRuta (String origen, String destino);
}

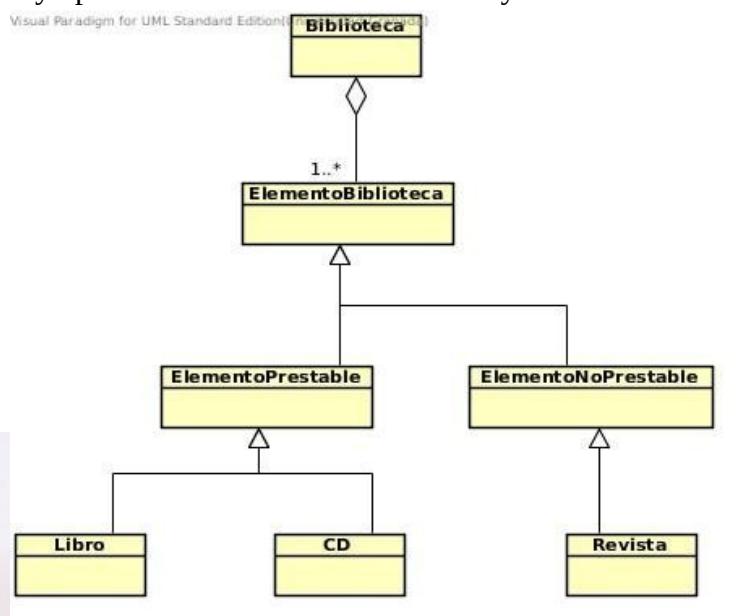
```

Crea dos nuevas clases, por ejemplo Bicicleta y NaveEspacial, que hereden de ella y que implementen el método hacerRuta de diferente forma cada una. Dibuja el diagrama de clases UML correspondiente a este ejercicio.

5. Implementar el ejercicio anterior en Ruby

6. Implementa en Java una clase paramétrica a partir de la cual se puedan definir clases de grupos de personas con un líder, como por ejemplo grupos de música con un cantante, o empresas con un jefe. Implementa también alguna de esas clases a partir de la clase paramétrica definida.

7. Partiendo del siguiente diagrama de clases, modifícalo añadiendo una interfaz Prestable implementada por la clase ElementoPrestable y sus subclases. Añade a este diagrama de clases los atributos y operaciones de las distintas clases y de la interfaz Prestable.



8. Teniendo en cuenta el siguiente diagrama de clases, señala en la segunda columna de la tabla aquellas líneas de código que contienen un error de compilación por incompatibilidad de tipos (ECIT) y escribe en la tercera columna la corrección necesaria para evitarlos.



Código	ECIT	Corrección
Girable arti=new Artista();		
Cantante cant1=new Cantante();		
Cantante sol=new Solista();		
Solistा cant2=new Cantante();		
Bailarin bail= new Artista();		

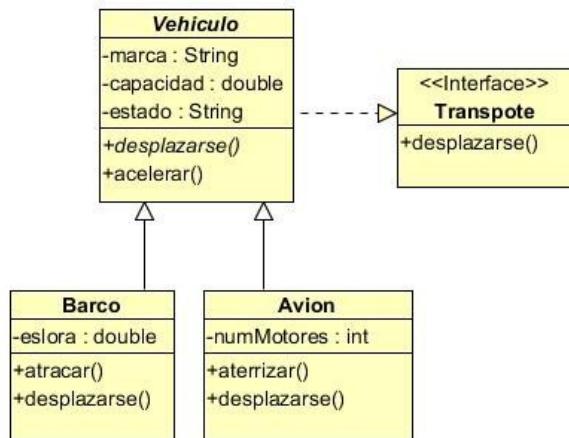
9. Teniendo en cuenta:

- El diagrama de clases del ejercicio anterior
- Que se han resuelto todos los errores del ejercicio
- Que en Java la primera posición de los contenedores es la 0
- Que todos los artistas actúan, pero sólo los cantantes cantan y sólo los solistas cantan solos

Marca las líneas donde se produce un error, indicando en la segunda columna si es de compilación (C) o de ejecución (E) y en la tercera el código correcto.

Código	Error(C/E)	Corrección
List<Artista> lista=new ArrayList();		
lista.add(arti);		
lista.add(cant1);		
lista.add(sol);		
lista.add(cant2);		
lista.add(bail);		
lista.get(1).canta();		
lista.get(0).actua();		
(Solista) lista.get(3).cantaSolo();		

10. Dado el siguiente diagrama de UML, impleméntalo en Java. Presta atención a que el nombre de la clase Vehículo y su método desplazarse() aparecen en cursiva.



11. Teniendo presente el diagrama anterior, indica cuáles de los siguientes trozos de código son correctos y cuáles darían error de compilación o de ejecución. En caso de que den error, indica por qué y cómo lo solucionarías.

Código	Error y si procede corrección
Transporte x = new Barco(); x.atrancar();	
Avion av = new Avion(); av.acelerar();	
Vehiculo v = new Vehiculo(); v.desplazarse();	
Vehiculo v2 = new Vehiculo(); v2.acelerar();	
Transporte t = new Avion(); Barco b= new Barco(); t = b;	
Avion a = new Avion(); String est = a.estado;	
Vehiculo v3 = new Barco(); ((Barco) v3).atrancar();	
List<Transporte> listaTransportes = new ArrayList(); listaTransportes.add(new Barco()); listaTransportes.add(new Avion()); listaTransportes.add(new Barco()); for(Transporte tr: listaTransportes){ tr.acelerar(); tr.atrancar();}	
List<Transporte> otraLista = new ArrayList(); otraLista.add(new Barco()); otraLista.add(new Avion()); otraLista.add(new Barco()); for(Transporte tr: otraLista){ ((Barco) tr).acelerar(); ((Barco) tr).atrancar();}	

12. A partir del siguiente diagrama de clases y del código proporcionado, detecta errores de compilación y de ejecución. Corrígelos para que funcione correctamente.

```

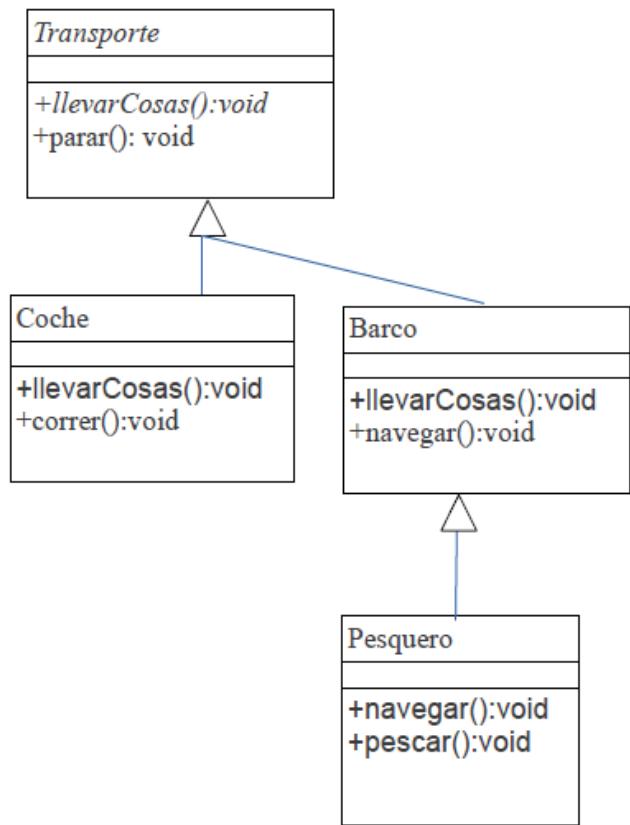
class PruebaLigadura {
    public static void main ( String args[]){
        Coche c1 = new Coche();
        c1.llevarCosas();
        c1.correr();

        Barco b = new Barco();
        b.llevarCosas();
        b.navegar();
        b.correr();
        b=c1;

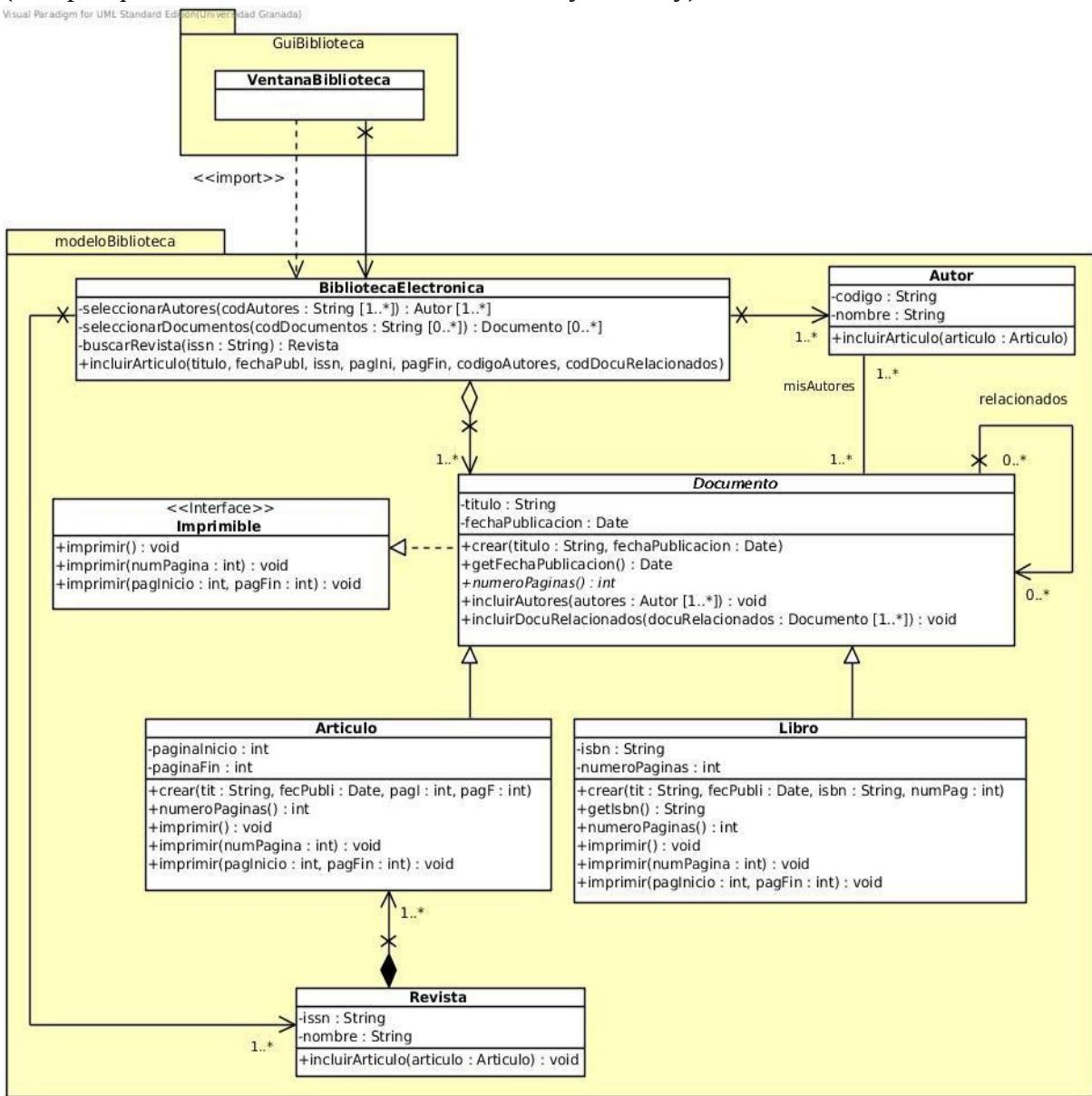
        Pesquero p = new Pesquero();
        p.navegar();
        p.pescar();
        p.llevarCosas();
        p=b;
        b=p;
        b.navegar();
        b.pescar();

        ArrayList v= new ArrayList();
        v.add(c1);
        v.add(p);
        (v.get(1)).navegar();
        (v.get(1)).llevarCosas();
    }
}

```



13. A partir de los siguientes diagramas de clases, resuelve las cuestiones que se plantea (siempre que sean de codificación hazlo en Java y en Ruby):



- Define la clase *Documento* (cabecera, atributos y cabeceras de los métodos).
- Define la clase *Articulo* (cabecera, atributos y cabeceras de los métodos).
- Implementa el constructor que se indica en la clase *Articulo*.
- Define la interfaz *Imprimible*.
- Indica los atributos que definen el estado de la clase *BibliotecaElectronica*.
- La clase *Documento* figura en cursiva, lo que indica que es abstracta. Indica los dos motivos por los que lo es.
- En este modelo, ¿puede haber artículos que no estén ligados a una revista?
- Indica sobre las relaciones del diagrama de clases con cuál de los siguientes tipos se corresponden: asociación (AS), agregación (AG), composición (C), realización (R), herencia (H), dependencia (D).
- Escribe el contenido del fichero *VentanaBiblioteca* completo (pero sin añadir nada que no aparezca en el diagrama).

- En la clase *Articulo*, indica cuáles de sus métodos están sobrecargados o redefinidos. Justifica tu respuesta.
- Corrige el código:

```
Imprimible docu = new Documento("Intemperie", fecha); // suponiendo
que fecha está inicializada
docu.imprimir();
```
- Corrige el código:

```
Documento docu = new Libro("ISBN10102030");
String codigo = docu.getIsbn();
```
- Rellena la siguiente tabla indicando el tipo estático y dinámico de la variable *docu* en las siguientes líneas de código:

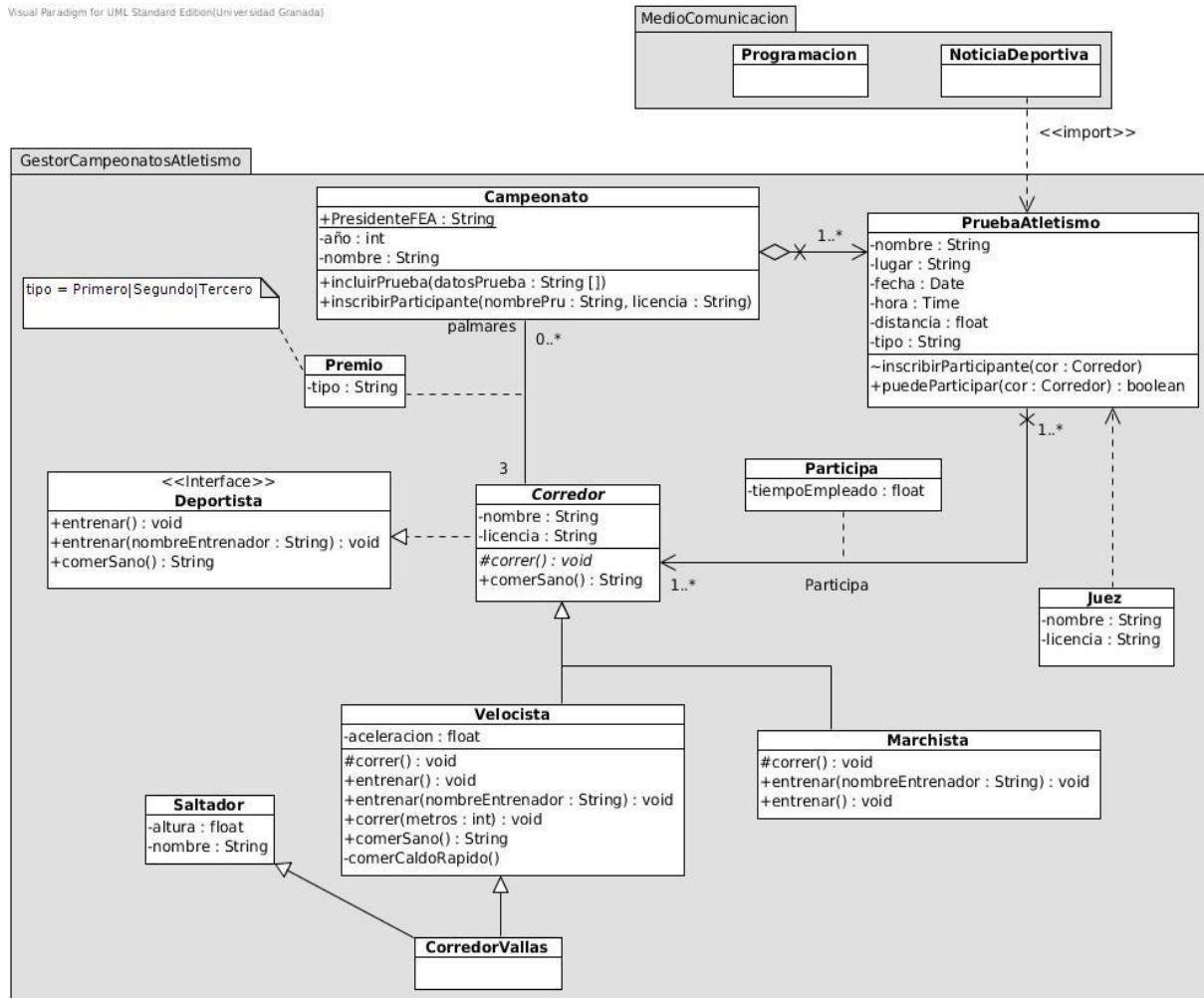
	Tipo estático	Tipo dinámico
Documento docu = new Articulo(titulo, fecha, pagIni, pagFin);		
docu.imprimir(pagIni, pagFin);		
docu = new Libro(titulo, fecha, isbn, pags);		
docu.imprimir();		

- Indica si hay errores de compilación o ejecución en el código anterior (suponiendo que las variables *titulo*, *fecha*, *pagIni*, *pagFin*, *isbn* y *pags* han sido declaradas e inicializadas convenientemente con anterioridad). Justifica tu respuesta.
- Rellena la siguiente tabla marcando con una “X” la situación que corresponda a cada una de las líneas del siguiente bloque de código (suponiendo que las variables *titulo*, *fecha*, *pagIni* y *pagFin* han sido declaradas e inicializadas convenientemente con anterioridad).

	Ningún error	Sólo error de compilación	Sólo error de ejecución
Imprimible imp = new Articulo(titulo, fecha, pagIni, pagFin) ;			
Documento docu = imp;			
imp.numeroPaginas();			
((Libro) docu).getIsbn();			

14. Partiendo del siguiente diagrama de clases:

Visual Paradigm for UML Standard Edition(Universidad Granada)



- Responde V (Verdadero) o F (Falso) a las siguientes cuestiones:

Desde la clase NoticiaDeportiva se puede acceder a todos los elementos públicos del paquete GestorCampeonatosAtletismo directamente	
Un CorredorVallas es un Velocista y Saltador	
El estado de un objeto de la clase Juez no viene determinado por el estado de un objeto de la clase PruebaAtletismo	
Una PruebaAtlettismo puede existir sin estar asociada a la clase Campeonato	
Un Velocista es un Corredor	
Un Deportista es un Corredor	
La clase Corredor tiene 4 métodos, 3 abstractos y 1 concreto	
La clase Marchista está mal representada en el diagrama, debe ser abstracta	
La clase CorredorVallas presenta un conflicto de nombres	
Todas las instancias de la clase Campeonato tienen una copia de la variable PresidenteFEA	

- Marca de los siguientes cuáles son métodos abstractos en Marchista:

correr()	
comerSano()	
entrenar()	
entrenar(nombreEntrenador:String)	

- Marca qué métodos están redefinidos y/o sobrecargados en la clase Velocista.

	redefinido	sobrecargado
correr()		
comerSano()		
entrenar()		

- Proporciona el tipo estático y dinámico de la variable que se indica en las siguientes sentencias Java.

	Variable	Tipo Estático	Tipo Dinámico
Deportista c= new Velocista();	c		
Corredor d= new Marchista();	d		
c=new Marchista();	c		
d = c;	d		

- En el siguiente código Java:

- Corrige los posibles errores de compilación.
- Una vez corregidos los errores de compilación, indica las líneas de código en las que habría error de ejecución.

	Corrección del error en Compilación	Error en ejecución
Corredor c= new Velocista();		
Deportista d= new Marchista();		
d.comerSano();		
Marchista m= d;		
d=c;		
d.correr(150);		
Velocista v = d;		
ArrayList<Corredor> corredores = new ArrayList();		
corredores.add(c);		
corredores.add(m);		
corredores.get(0).correr(10);		
corredores.get(1).correr(10);		
c= new Corredor();		

- Completa el código de la siguiente clase parametrizada en Java, la cual representa a un club de corredores de cualquier tipo: velocistas o marchistas.

```

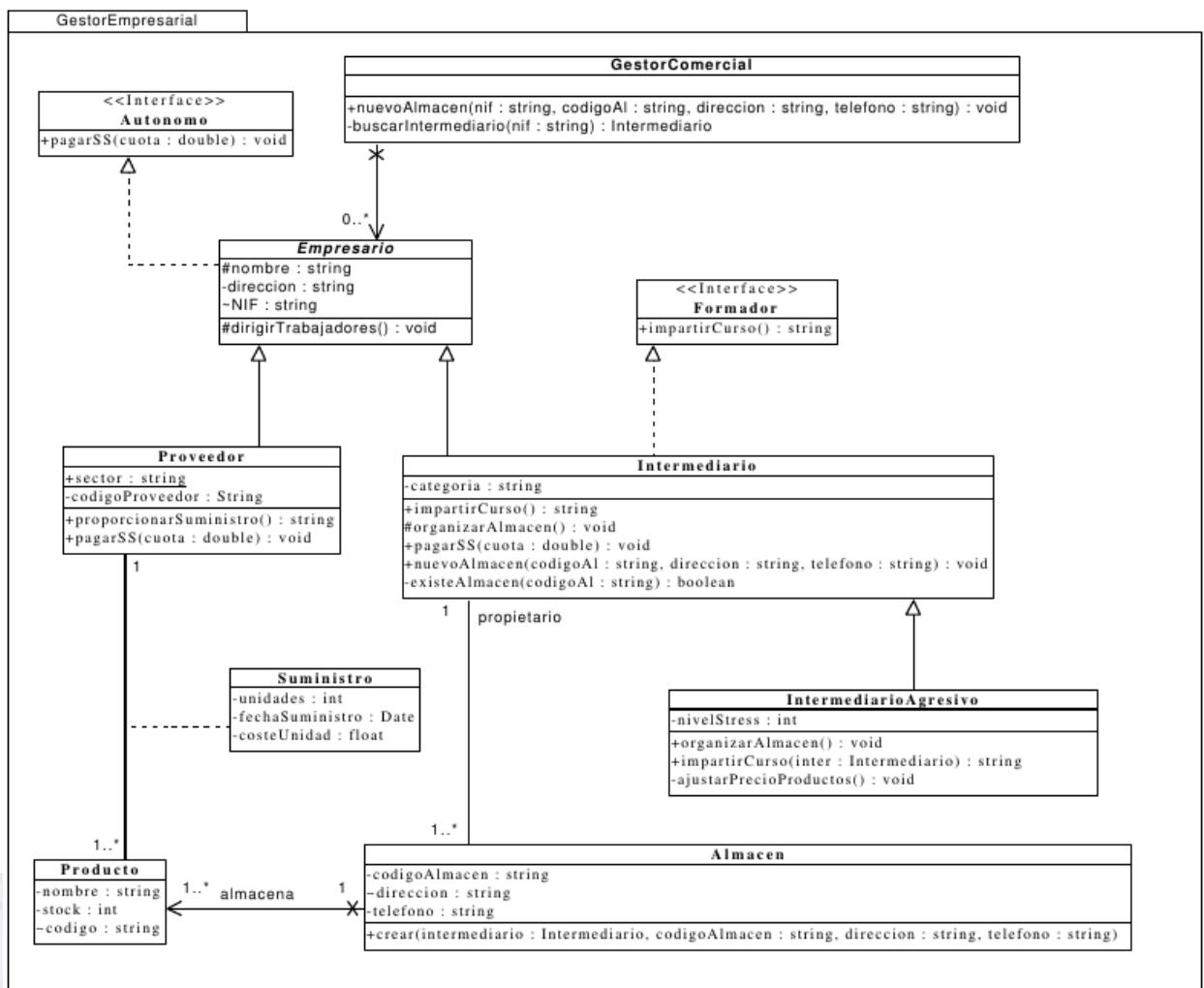
public class Club<.....> {
    private Map<String, .....> miembros; // key = número de licencia
    private .... lider;

    // Constructor....
    public Club(..... lider) { }
    // Consultor de un miembro del club a partir del numero de licencia
    public .... getMiembro(String numeroLicencia){ }
        // Incluir un nuevo miembro en el club
    public void incluirMiembro(.... miembro){ }
    // cambiar el líder
        public void setLider(..... lider) { }
        // Obtener el líder
        public .... getLider(){ }
}

```

- Escribe un pequeño main en el que se use la clase Club<T> como club de Velocistas

15. Dado el diagrama:



Indica si las siguientes líneas de código Java producen error de compilación, de ejecución, ambos o ninguno. Supón que están en un main en una clase nueva dentro del mismo paquete. Si hay error explica cómo lo arreglarías y si no hay error, indícalo explícitamente

<i>Considera para este ejercicio que todas las clases tienen un constructor válido que no recibe atributos.</i>	Error de compilación	Error de ejecución
Formador f = new Intermediario();		
f.pagarSS(23.4);		
Autonomo auto1 = f;		
Autonomo auto2 = (Proveedor) f;		
IntermediarioAgresivo emp1 = new IntermediarioAgresivo(); emp1.ajustarPrecioProductos();		
Empresario emp2 = new Empresario();		
ArrayList<Formador> formadores = new ArrayList(); formadores.add(f); formadores.add(emp1);		
formadores.get(1).impartirCurso();		
formadores.get(1).impartirCurso(f);		
((IntermediarioAgresivo)formadores.get(0)).impartirCurso(emp1);		

2.3.1. Soluciones

Índice

1. Ejercicio 1	4
1.1. Enunciado	4
1.2. Solución	4
2. Ejercicio 2	4
2.1. Enunciado	4
2.2. Solución	5
3. Ejercicio 3	6
3.1. Enunciado	6
3.2. Solución	6
4. Ejercicio 4	7
4.1. Enunciado	7
4.2. Solución	7
5. Ejercicio 5	7
5.1. Enunciado	7
5.2. Solución	7
6. Ejercicio 6	10
6.1. Enunciado	10
6.2. Solución	10
7. Ejercicio 7	11
7.1. Enunciado	11
7.2. Solución	11
7.2.1. Análisis línea por línea	11
8. Ejercicio 8	12
8.1. Enunciado	12
8.2. Solución	13
8.2.2. Tabla de diferencias	13
8.2.2. Solucion	14
9. Ejercicio 9	14
9.1. Enunciado	14
9.2. Solución	14
10. Ejercicio 10	16
11. Ejercicio 11	17
11.1. Enunciado	17
11.2. Solución	17

12. Ejercicio 12	19
12.1. Enunciado	19
12.2. Solución en Java	21
12.3. Solución en Ruby	25
13. Ejercicio 13	28
13.1. Enunciado Y Solución	29
14. Ejercicio 14	32
14.1. Enunciado	32
14.2. Solución	34

1 Ejercicio 1

1.1. Enunciado

Dada la siguiente clase en Java:

```

1 public class Vertebrado
2 {
3     public ArrayList<String> partesDelAbdomen();
4     public void desplazarse();
5     public String comunicarse(Vertebrado vertebrado);
6     protected Vertebrado obtenerCopia();
7 }
```

Indica con una cruz en la casilla correspondiente según si la declaración de los siguientes métodos de la clase Mamifero, subclase de Vertebrado, sobrecargan (*overloading*) o redefinen (*overriding*) a los de la clase Vertebrado.

1.2. Solución

Método	Sobrecarga	Redefinición
public ArrayList<String>partesDelAbdomen()		X
public void desplazarse(Modo m)	X	
public String comunicarse(Vertebrado vertebrado)		X
public String comunicarse(Mamifero mamifero)	X	
public Vertebrado obtenerCopia()		X
protected Mamifero obtenerCopia()		X

Nota: Al cambiar únicamente la visibilidad o el tipo hace que sea una redefinición (Últimos dos casos)

2 Ejercicio 2

2.1. Enunciado

A partir de las siguientes clases:

```

1 public class Persona {
2     protected String nombre;
3     public Persona(String nom) { this.setNombre(nom); }
4     protected String getNombre() { return this.nombre; }
5     public String hablar() { return "bla bla bla"; }
6 }
```

```

1 public class Estudiante extends Persona {
2     public String carrera;
3     public int curso;
4     public Estudiante(String nom, String carr, int cur) {
5         super(nom);
```

```

6     carrera = carr;
7     curso = cur;
8 }
9 public void estudiar() { System.out.println("Estudiando"); }
10}

```

Implementa la clase EstudianteInformatica que hereda de Estudiante. Debe tener:

- Una nueva variable de instancia que es una colección de String con los dispositivos que utiliza (por ejemplo, PC, tablet, smartphone).
- Métodos para consultar esa variable.
- Una redefinición del método estudiar() para que además de estudiar como los otros alumnos, diga que estudia con el último dispositivo de su colección.
- Un constructor para inicializar la clase.

Implementa este problema en Java y Ruby:

2.2. Solución

```

1 public class EstudianteInformatica extends Estudiante {
2     private ArrayList<String> dispositivos;
3     public EstudianteInformatica(string nombre, string curso){
4         super(nombre, "Informatica", curso);
5         dispositivos = new ArrayList<String>();
6     }
7
8     @Override
9     public void Estudiar(){
10        System.out.println("Informatica con " + dispositivos.get(
11            dispositivos.size() - 1));
12    }
12} // Fin de la clase EstudianteInformatica

```

Listing 1: Clase Estudiante de Informática

```

1 def EstudianteInformatica < Estudiante
2     def initialize (nombre, curso)
3         super(nombre, "Informatica", curso)
4         @dispositivos = [] # @dispositivos = Array.new
5     end
6
7     def Estudiar
8         super.Estudiar
9         puts "Estudiando Informatica con #{@dispositivos[-1]}"
10    end
11 end

```

Listing 2: Clase Estudiante de Informática

3 Ejercicio 3

3.1. Enunciado

Nota: En las diapositivas de la Relación hay un error, donde pone 3, no es el ejercicio 3 es la parte 2 del ejercicio 2, es decir, otra clase, el ejercicio 3 es el marcado con el número 4.

Dada la siguiente clase abstracta en Java:

```
1 abstract class Transporte {  
2     protected String marca;  
3  
4     protected String getMarca() {  
5         return this.marca;  
6     }  
7  
8     protected void setMarca(String marc) {  
9         this.marca = marc;  
10    }  
11  
12    public abstract String hacerRuta(String origen, String destino);  
13}
```

Crea dos nuevas clases, Bicicleta y NaveEspacial, que hereden de Transporte e implementen el método hacerRuta de forma diferente:

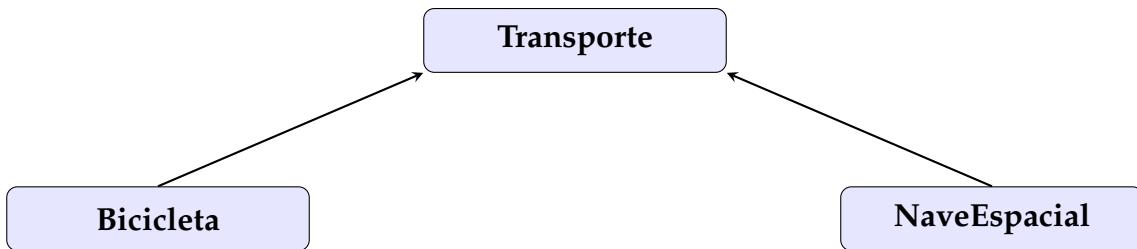
3.2. Solución

```
1 public class Bicicleta extends Transporte {  
2     @Override  
3     public String hacerRuta(String origen, String destino) {  
4         return "Pedaleando desde " + origen + " hasta " + destino;  
5     }  
6  
7  
8     public class NaveEspacial extends Transporte {  
9         @Override  
10        public String hacerRuta(String origen, String destino) {  
11            return "Volando desde " + origen + " hasta " + destino + " por  
12                el espacio";  
13        }  
14    }
```

Listing 3: Clase Bicicleta y Clase Nave Espacial

5 EJERCICIO 5

Diagrama UML



4 Ejercicio 4

4.1. Enunciado

Hacer el ejercicio anterior, pero en Ruby.

4.2. Solución

```
1 def Bicicleta < Transporte
2     def hacerRuta(origen, destino)
3         "Pedaleando desde #{origen} hasta #{destino}"
4     end
5 end
6
7 def NaveEspacial < Transporte
8     def hacerRuta(origen, destino)
9         "Volando desde #{origen} hasta #{destino} por el espacio"
10    end
11 end
```

Listing 4: Clase Bicicleta y Clase Nave Espacial

5 Ejercicio 5

5.1. Enunciado

Implementa en Java una clase paramétrica a partir de la cual se puedan definir clases de grupos de personas con un líder, como por ejemplo grupos de música con un cantante, o empresas con un jefe. Implementa también alguna de esas clases a partir de la clase paramétrica definida.

5.2. Solución

A continuación, se implementa una clase paramétrica en Java para definir grupos de personas con un líder. Posteriormente, se muestra un ejemplo de uso creando un grupo de música con un cantante.

```
1 // Clase paramétrica genérica
2 public class Grupo<T> {
3     private T lider;
4     private List<T> miembros;
5
6     public Grupo(T lider) {
7         this.lider = lider;
8         this.miembros = new ArrayList<>();
9     }
10
11    public T getLider() {
12        return lider;
13    }
14
15    public void setLider(T lider) {
16        this.lider = lider;
17    }
18
19    public void agregarMiembro(T miembro) {
20        this.miembros.add(miembro);
21    }
22
23    public List<T> getMiembros() {
24        return miembros;
25    }
26
27    @Override
28    public String toString() {
29        return "Líder: " + lider + ", Miembros: " + miembros;
30    }
31 }
32
33 // Ejemplo de uso con un grupo de música
34 public class Musico {
35     private String nombre;
36
37     public Musico(String nombre) {
38         this.nombre = nombre;
39     }
40
41     @Override
42     public String toString() {
43         return nombre;
44     }
45 }
46
47 public class Main {
48     public static void main(String[] args) {
49         Musico cantante = new Musico("Freddie Mercury");
50         Grupo<Musico> banda = new Grupo<>(cantante);
51
52         banda.agregarMiembro(new Musico("Brian May"));
53         banda.agregarMiembro(new Musico("Roger Taylor"));
54         banda.agregarMiembro(new Musico("John Deacon"));
55 }
```

```
56     System.out.println(banda);
57 }
58 }
```

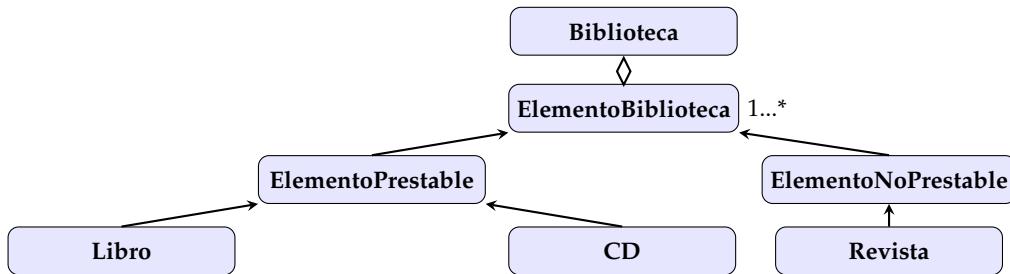
Si queremos implementar esta misma solución en Ruby, sería como sigue:

```
1 # Clase paramétrica genérica
2 class Grupo
3   attr_accessor :lider, :miembros
4
5   def initialize(lider)
6     @lider = lider
7     @miembros = []
8   end
9
10  def agregar_miembro(miembro)
11    @miembros << miembro
12    # Otras formas de añadir un miembro en Ruby
13    # @miembros.push(miembro)
14    # @miembros += [miembro]
15    # @miembros.concat([miembro])
16  end
17
18  def to_s
19    "Líder: #{@lider}, Miembros: #{@miembros.join(', ')}"
20  end
21 end
22
23 # Ejemplo de uso con un grupo de música
24 class Musico
25   attr_accessor :nombre
26
27   def initialize(nombre)
28     @nombre = nombre
29   end
30
31   def to_s
32     @nombre
33   end
34 end
35
36 # Crear grupo de música
37 cantante = Musico.new("Freddie Mercury")
38 banda = Grupo.new(cantante)
39
40 banda.agregar_miembro(Musico.new("Brian May"))
41 banda.agregar_miembro(Musico.new("Roger Taylor"))
42 banda.agregar_miembro(Musico.new("John Deacon"))
43
44 puts banda
```

6 Ejercicio 6

6.1. Enunciado

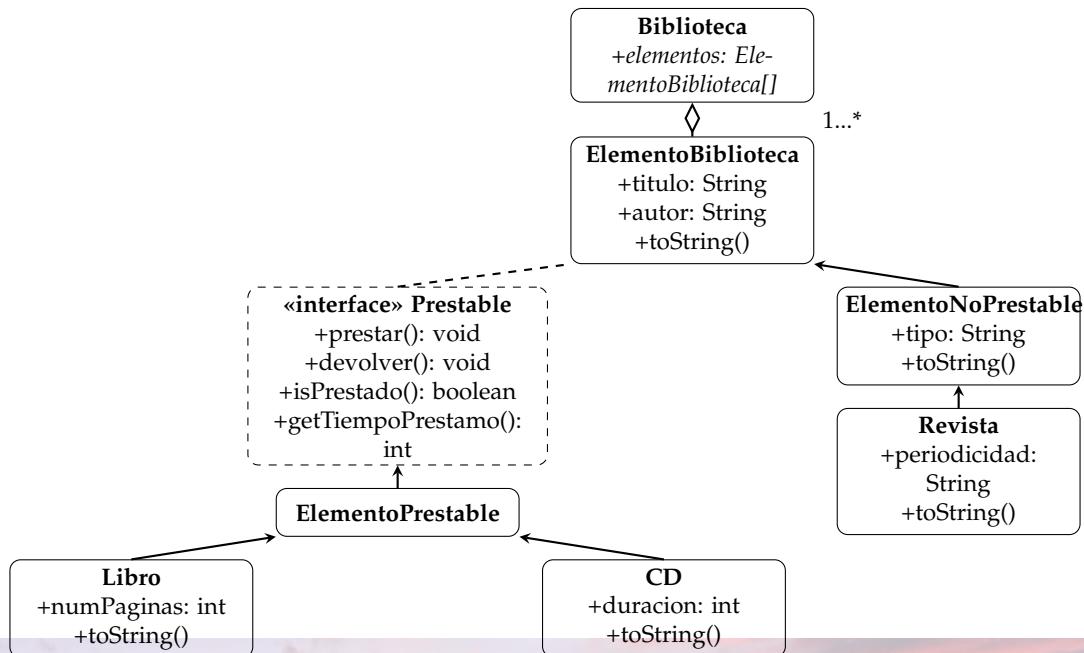
Partiendo del siguiente diagrama de clases, modifícalo añadiendo una interfaz Prestable implementada por la clase ElementoPrestable y sus subclases. Añade a este diagrama de clases los atributos y operaciones de las distintas clases y de la interfaz Prestable.



6.2. Solución

Detalles:

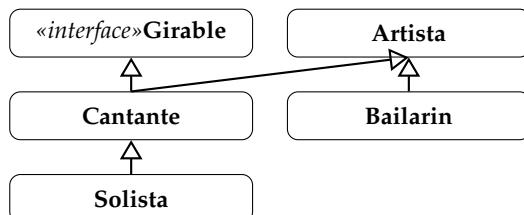
- Elemento Prestable hereda los métodos de la interfaz Prestable, por eso no se ponen en el diagrama.
- En el diagrama se sobreentiende el tipo de los `toString` que son de tipo String.



7 Ejercicio 7

7.1. Enunciado

Teniendo el cuenta el siguiente diagrama de clases, señala en la segunda columna de la tabla aquellas líneas de código que contienen un error de compilación por incompatibilidad de tipos (ECIT) y escribe en la tercera columna la corrección necesaria para evitarlos.



Código	ECIT	Corrección
Girable arti=new Artista();		
Cantante cant1=new Cantante();		
Cantante sol=new Solista();		
Solista cant2=new Cantante();		
Bailarin bail= new Artista();		

Figura 1: Tabla de código y corrección

7.2. Solución

Código	ECIT	Corrección
Girable arti=new Artista();	X	Girable arti = new Cantante()
Cantante cant1=new Cantante();	✓	
Cantante sol=new Solista();	✓	
Solista cant2=new Cantante();	X	Cantante cant2=new Cantante();
Bailarin bail= new Artista();	X	Bailarin bail= new Bailarin();

Figura 2: Tabla de código y corrección

7.2.1. Análisis línea por línea

- Girable arti = new Artista();
 - **Problema:** La clase Artista no implementa la interfaz Girable. Esto genera un error de compilación por incompatibilidad de tipos.
 - **Corrección:** Usar una clase que implemente la interfaz Girable, como Cantante.
 - **Resultado corregido:** Girable arti = new Cantante();
- Cantante cant1 = new Cantante();

8 EJERCICIO 8

- **Problema:** Ninguno. La asignación es válida porque la referencia y la instancia son de la misma clase.
 - **Corrección:** No es necesario hacer ningún cambio.
 - **Resultado corregido:** (Sin cambios)
- Cantante sol = new Solista();
- **Problema:** Ninguno. Solista es una subclase de Cantante, por lo que esta asignación es válida.
 - **Corrección:** No es necesario hacer ningún cambio.
 - **Resultado corregido:** (Sin cambios)
- Solista cant2 = new Cantante();
- **Problema:** Incompatibilidad de tipos. Cantante es la clase base de Solista, y no se puede asignar una instancia de una clase base a una referencia de una subclase.
 - **Corrección:** Cambiar el tipo de referencia a Cantante.
 - **Resultado corregido:** Cantante cant2 = new Cantante();
- Bailarin bail = new Artista();
- **Problema:** Artista es una clase base abstracta y no se puede asignar directamente a una referencia de su subclase Bailarin.
 - **Corrección:** Crear una instancia de Bailarin en lugar de Artista.
 - **Resultado corregido:** Bailarin bail = new Bailarin();

8 Ejercicio 8

8.1. Enunciado

Teniendo en cuenta:

- El diagrama de clases del ejercicio anterior
- Que se han resuelto todos los errores del ejercicio
- Que en Java la primera posición de los contenedores es la 0
- Que todos los artistas actúan, pero sólo los cantantes cantan y sólo los solistas cantan solos

Marca las líneas donde se produce un error, indicando en la segunda columna si es de compilación (C) o de ejecución (E) y en la tercera el código correcto.

Código	Error (C/E)	Corrección
List<Artista>lista = new ArrayList();		
lista.add(arti);		
lista.add(cant1);		
lista.add(sol);		
lista.add(cant2);		
lista.add(bail);		
lista.get(1).canta();		
lista.get(0).actua();		
(Solista) lista.get(3).cantaSolo();		

Figura 3: Tabla de código y corrección

8.2. Solución

8.2.1. Tabla de diferencias

Criterio	Error de Compilación	Error de Ejecución
Cuándo ocurre	Durante la compilación.	Durante la ejecución del programa.
Tipo de problema	Problemas de sintaxis, tipos o reglas del lenguaje.	Problemas lógicos o de recursos.
Detección	Detectado por el compilador antes de ejecutar.	Ocurre mientras el programa está en ejecución.
Ejemplo común	Declaración de tipos incompatibles.	Dividir por cero o acceder a un índice inválido.

Cuadro 1: Principales diferencias entre errores de compilación y errores de ejecución

9 EJERCICIO 9

8.2.2. Solucion

Código	Error (C/E)	Corrección
List<Artista>lista = new ArrayList();	C	List<Artista>lista = new ArrayList<Artista>();
lista.add(arti);	No	Debido a que es un cantante y es una subclase de artista
lista.add(cant1);	No	Debido a que cantante esta por debajo en el diagrama de clases
lista.add(sol);	No	Debido a que es una subclase de Cantante y esta también lo es de Artista
lista.add(cant2);	No	"
lista.add(bail);	No	"
lista.get(1).canta();	No	Es un cantante
lista.get(0).actua();	E	Es un cantante
(Solista) lista.get(3).cantaSolo();	E	en la posición 3 es un cantante

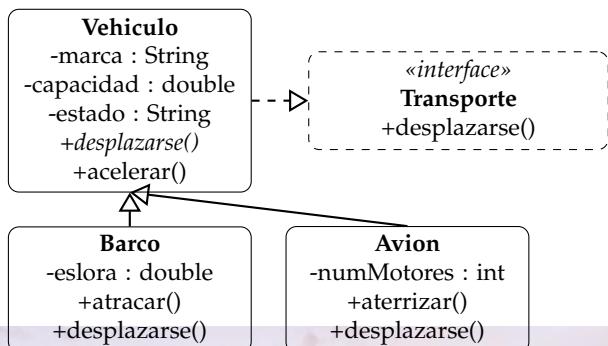
Figura 4: Tabla de código y corrección

9 Ejercicio 9

9.1. Enunciado

Dado el siguiente diagrama de UML, impleméntalo en Java. Presta atención a que el nombre de la clase Vehículo y su método desplazarse() aparecen en cursiva.

Nota: cuando en una clase no figura el tipo, es porque se trata de tipo void.



9.2. Solución

```

1 class Vehículo implements Transporte {
2     private String marca;
3     private double capacidad;
4     private String estado;
5
6     public void desplazarse() {
7         System.out.println("Desplazándose");
8     }
9
10    public void acelerar() {
11        System.out.println("Acelerando");
12    }
13}
14

```

Listing 5: Clase Vehiculo

```

1 class Barco extends Vehículo {
2     private double eslora;
3
4     public void atracar() {
5         System.out.println("Atracando");
6     }
7
8     @Override
9     public void desplazarse() {
10        System.out.println("Desplazándose por el agua");
11    }
12
13}
14

```

Listing 6: Clase Barco

```

1 class Avion extends Vehículo {
2     private int numMotores;
3
4     public void aterrizar() {
5         System.out.println("Aterrizando");
6     }
7
8     @Override
9     public void desplazarse() {
10        System.out.println("Desplazándose por el aire");
11    }
12
13}
14

```

Listing 7: Clase Avion

```

1 interface Transporte {
2     void desplazarse();
3
4 }

```

Listing 8: Interfaz Transporte

10 Ejercicio 10

Teniendo presente el diagrama anterior, indica cuáles de los siguientes trozos de código son correctos y cuáles darían error de compilación o de ejecución. En caso de que den error, indica por qué y cómo lo solucionarías.

Código	Error si procede	Corrección
Transporte x = new Barco(); x.atracar();	Error de compilación, ya que el tipo de la variable x es Transporte y no tiene el método atracar.	Transporte x = new Barco(); ((Barco) x).atracar();
Avion av = new Avion(); av.acelerar();	✓.	
Vehiculo v = new Vehiculo(); v.desplazarse();	✓.	
Vehiculo v2 = new Vehiculo(); v2.acelerar();	✓.	
Transporte t = new Avion(): Barco b= new Barco(); t = b;	✓.	
Avion a = new Avion(); String est = a.estado;	Error de compilación, ya que el atributo estado es privado en la clase Vehiculo.	Implementar un consultor para que devuelva el valor estado.
Vehiculo v3 = new Barco(); ((Barco) v3).atracar();	✓.	
List<Transporte>listaTransportes = newArrayList(); listaTransportes.add(new Barco()); listaTransportes.add(new Avion()); listaTransportes.add(new Barco()); for(Transporte tr: listaTransportes) tr.acelerar(); tr.atracar();	Error de compilación, ya que el método atracar() no está definido en la interfaz Transporte.	No usar esos métodos.
List<Transporte>otraLista = new ArrayList(); otraLista.add(new Barco()); otraLista.add(new Avion()); otraLista.add(new Barco()); for(Transporte tr: otraLista) ((Barco) tr).acelerar(); ((Barco) tr).atracar();	Error de ejecución, ya que no se puede hacer un cast de un objeto de tipo Avion a Barco.	No usar esos métodos.

11 Ejercicio 11

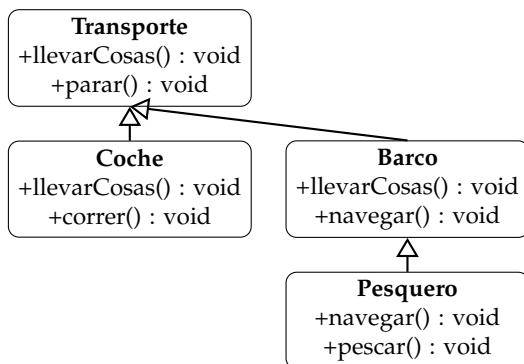
11.1. Enunciado

A partir del siguiente diagrama de clases y del código proporcionado, detecta errores de compilación y de ejecución. Corrígelos para que funcione correctamente.

```

1  class PruebaLigadura {
2      public static void main ( String args[]){
3          Coche c1 = new Coche();
4          c1.llevarCosas();
5          c1.correr();
6          Barco b = new Barco();
7          b.llevarCosas();
8          b.navegar();
9          b.correr();
10         b=c1;
11         Pesquero p = new Pesquero();
12         p.navegar();
13         p.pescar();
14         p.llevarCosas();
15         p=b;
16         b=p;
17         b.navegar();
18         b.pescar();
19         ArrayList v= new ArrayList();
20         v.add(c1);
21         v.add(p);
22         (v.get(1)).navegar();
23         (v.get(1)).llevarCosas();
24     }
25 }
```

Listing 9: Ejercicio del Enunciado



11.2. Solución

```

1  class PruebaLigadura {
2      public static void main ( String args[]){
3          Coche c1 = new Coche();
4          c1.llevarCosas();
5          c1.correr();
```

```
6     Barco b = new Barco();
7     b.llevarCosas();
8     b.navegar();
9     b.correr(); // Error: Barco no tiene el método correr
10    b=c1; // Error: No se puede asignar un Coche a un Barco
11    Pesquero p = new Pesquero();
12    p.navegar();
13    p.pescar();
14    p.llevarCosas();
15    p=b; // Error: No se puede asignar un Barco a un Pesquero
16    b=p;
17    b.navegar();
18    b.pescar(); // Error: Barco no tiene el método pescar
19    ArrayList<Transporte> v= new ArrayList<>();
20    v.add(c1);
21    v.add(p);
22    ((Barco) v.get(1)).navegar();
23    v.get(1).llevarCosas();
24 }
25 }
26
27 //-----Implementación de las clases-----
28
29 class Transporte {
30     public void llevarCosas() {
31         System.out.println("Llevando cosas");
32     }
33     public void parar() {
34         System.out.println("Parando");
35     }
36 }
37
38 class Coche extends Transporte {
39     public void correr() {
40         System.out.println("Corriendo");
41     }
42 }
43
44 class Barco extends Transporte {
45     public void navegar() {
46         System.out.println("Navegando");
47     }
48 }
49
50 class Pesquero extends Barco {
51     public void pescar() {
52         System.out.println("Pescando");
53     }
54 }
```

Listing 10: Ejercicio del Enunciado

12 Ejercicio 12

12.1. Enunciado

A partir de los siguientes diagramas de clases, resuelve las cuestiones que se plantea (siempre que sean de codificación hazlo en Java y en Ruby):

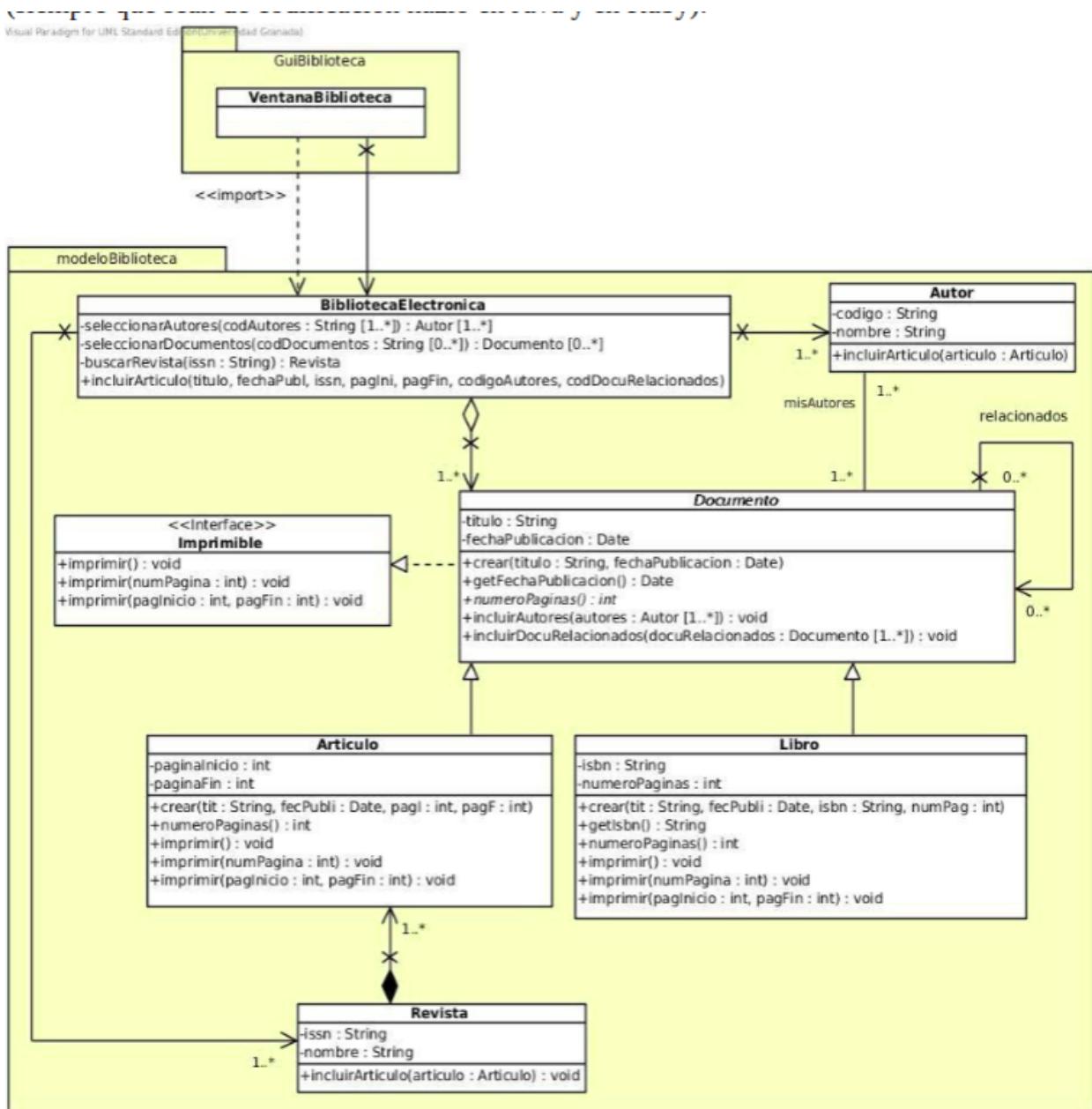


Figura 5: Diagrama de clases

- Define la clase Documento (cabecera, atributos y cabeceras de los métodos).
- Define la clase Articulo (cabecera, atributos y cabeceras de los métodos).

- Implementa el constructor que se indica en la clase Artículo.
- Define la interfaz Imprimible.
- Indica los atributos que definen el estado de la clase BibliotecaElectronica.
- La clase Documento figura en cursiva, lo que indica que es abstracta. Indica los dos motivos por los que lo es.
- En este modelo, ¿puede haber artículos que no estén ligados a una revista?
- Indica sobre las relaciones del diagrama de clases con cuál de los siguientes tipos se corresponden: asociación (AS), agregación (AG), composición (C), realización (R), herencia (H), dependencia (D).
- Escribe el contenido del fichero VentanaBiblioteca completo (pero sin añadir nada que no aparezca en el diagrama).
- En la clase Artículo, indica cuáles de sus métodos están sobrecargados o redefinidos. Justifica tu respuesta.
- Corrige el código:

```

1  Imprimible docu = new Documento("Intemperie", fecha); //
2      suponiendo que fecha está inicializada
docu.imprimir();

```

- Corrige el código:

```

1  Documento docu = new Libro("ISBN10102030");
2  String codigo = docu.getIsbn();

```

- Rellena la siguiente tabla indicando el tipo estático y dinámico de la variable docu en las siguientes líneas de código:

Código	Tipo estático	Tipo dinámico
Documento docu = new Artículo(título, fecha, pagIni, pagFin);		
docu.imprimir(pagIni, pagFin);		
docu = new Libro(título, fecha, isbn, pags);		
docu.imprimir();		

- Indica si hay errores de compilación o ejecución en el código anterior (suponiendo que las variables título, fecha, pagIni, pagFin, isbn y pags han sido declaradas e inicializadas convenientemente con anterioridad). Justifica tu respuesta.
- Rellena la siguiente tabla marcando con una "X" la situación que corresponda a cada una de las líneas del siguiente bloque de código (suponiendo que las variables título, fecha, pagIni y pagFin han sido declaradas e inicializadas convenientemente con anterioridad).

Código	Ningún error	Sólo error de compilación	Sólo error de ejecución
Imprimible imp = new Articulo(titulo, fecha, pagIni, pagFin);			
Documento docu = imp;			
imp.numeroPaginas();			
((Libro) docu).getIsbn();			

12.2. Solución en Java

```

1 abstract class Documento implements Imprimible{
2     private String titulo;
3     private Date fechaPublicacion;
4     private Autor misAutores[];
5     private Documento relacionados[];
6
7
8     public Documento(String titulo, Date fechaPublicacion);
9
10    public Date getFechaPublicacion();
11
12    public abstract int numeroPaginas();
13
14    public void incluirAutores(Autor[] autores);
15
16    public void incluirDocuRelacionados(Documento[] docuRelacionados);
17
18
19 }
```

Listing 11: clase Documento (cabecera, atributos y cabeceras de los métodos)

```

1 class Articulo extends Documento{
2     private int paginaInicio;
3     private int paginaFin;
4
5     public Articulo(String tit, Date fecPubli, int pagI, int pagF){
6         super(tit, fecPubli);
7         this.paginaInicio = pagI;
8         this.paginaFin = pagF;
9     }
10    public int numeroPaginas();
11    public void imprimir();
12    public void imprimir(int numeroPagina);
13    public void imprimir(int pagInicio, int pagFin);
14 }
```

Listing 12: clase Articulo (cabecera, atributos y cabeceras de los métodos), constructor de la clase Artículo

```
1 interface Imprimible {  
2     void imprimir();  
3     void imprimir(int numeroPagina);  
4     void imprimir(int pagInicio, int pagFin);  
5 }
```

Listing 13: Intefaz Imprimible

```
1 class BibliotecaElectronica{  
2     private Autor autores[];  
3     private Documento documentos[];  
4     private Revista revistas[];  
5 }
```

Listing 14: Atributos que definen el estado de la clase BibliotecaElectronica

Motivos por los que la clase Documento es abstracta

1. Sirve como clase plantilla para las clases Libro y Revista que son clases hijas según el diagrama de clases.
2. Contiene al menos un método abstracto, que en este caso es el método numeroPaginas().

¿Puede haber artículos que no estén ligados a una revista?

En este modelo si, lo que no puede haber son revistas que no estén ligadas a un Artículo debido al diagrama de clases y la lógica del rombo de la flecha de composición de la clase Artículo (parte de esta).

Pintar en el diagrama de clases las uniones de: asociación (AS), agregación (AG), composición (C), realización (R), herencia (H), dependencia (D)

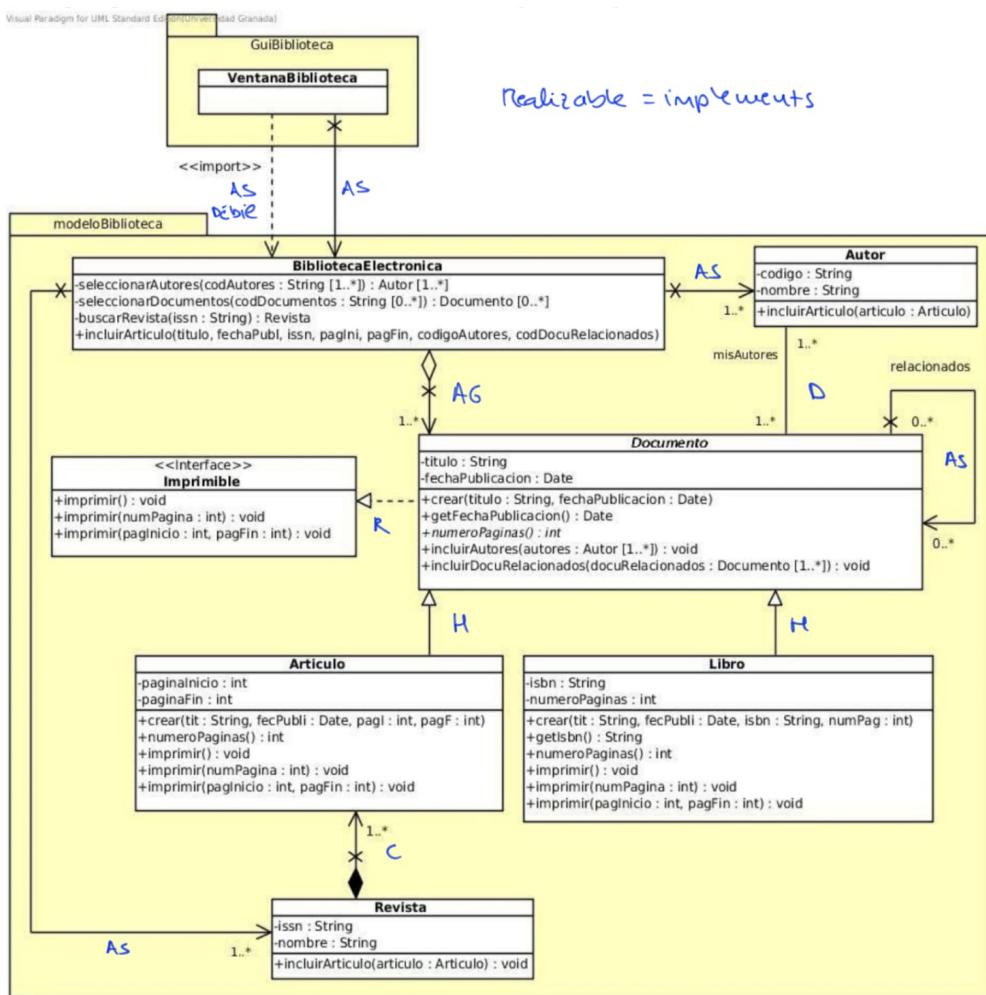


Figura 6: Diagrama de clases

Contenido del fichero VentanaBiblioteca completo (pero sin añadir nada que no aparezca en el diagrama)

```

1 class VentanaBiblioteca{
2     private BibliotecaElectronica biblioteca;
3
4     public VentanaBiblioteca(BibliotecaElectronica biblioteca){
5         this.biblioteca = biblioteca;
6     }
7 }
```

Listing 15: VentanaBiblioteca, el constructor se puede obviar, ya que usaríamos el por defecto

En la clase Articulo, indica cuáles de sus métodos están sobrecargados o redefinidos. Justifica tu respuesta.

- El método imprimir() está redefinido porque aparece en la clase Documento, el cual lo implementa de la interfaz Imprimible.
- El método imprimir(int numeroPagina) está redefinido porque aparece en la clase Documento, el cual lo implementa de la interfaz Imprimible.
- El método imprimir(int pagInicio, int pagFin) está redefinido porque aparece en la clase Documento, el cual lo implementa de la interfaz Imprimible.
- El método numeroPaginas() está redefinido porque sobrescribe el método de la clase Documento. Aunque sea abstracto, al cambiar únicamente el tipo se considera una redefinición¹.

Corrige el código

```
1  Imprimible docu = new Documento("Intemperie", fecha); // suponiendo que
   fecha está inicializada
2  docu.imprimir();
```

Listing 16: Código a corregir

Corrección:

```
1  Documento docu = new Articulo("Intemperie", fecha, 1, 10);
2  docu.imprimir();
```

Listing 17: Corrección del código

Corrige el código:

```
1  Documento docu = new Libro("ISBN10102030");
2  String codigo = docu.getIsbn();
```

Listing 18: Código a corregir

Corrección:

```
1  Libro docu = new Libro("ISBN10102030");
2  String codigo = docu.getIsbn();
```

Listing 19: Corrección del código

¹Mirar ejercicio 1.

Rellena la siguiente tabla indicando el tipo estático y dinámico de la variable docu en las siguientes líneas de código:

Código	Tipo estático	Tipo dinámico
Documento docu = new Articulo(titulo, fecha, pagIni, pagFin);	Documento	Articulo
docu.imprimir(pagIni, pagFin);	Documento	Articulo
docu = new Libro(titulo, fecha, isbn, pags);	Documento	Libro
docu.imprimir();	Documento	Libro

Indica si hay errores de compilación o ejecución en el código anterior (suponiendo que las variables titulo, fecha, pagIni, pagFin, isbn y pags han sido declaradas e inicializadas convenientemente con anterioridad). Justifica tu respuesta.

- En la primera línea de código no hay errores de compilación ni de ejecución.
- Si, debido a que no puedes usar imprimir de Documento, ya que aunque derive de la interfaz que lo tenga, no está implementado en Documento. Vemos que está en Artículo, pero para usarlo debemos de realizar un cast: ((Articulo)docu).imprimir(); *Nota: en java solo se puedes usar las funciones de los tipos estáticos, por eso en este caso da error de compilación y lo tenemos que hacer de la manera indicada.*
- En la tercera línea de código no hay errores de compilación ni de ejecución.
- Pasa lo mismo que en la 2º explicación de este.

Rellena la siguiente tabla marcando con una “X” la situación que corresponda a cada una de las líneas del siguiente bloque de código (suponiendo que las variables titulo, fecha, pagIni y pagFin han sido declaradas e inicializadas convenientemente con anterioridad).

Código	Ningún error	Sólo error de compilación	Sólo error de ejecución
Imprimible imp = new Articulo(titulo, fecha, pagIni, pagFin);	X		
Documento docu = imp;		X	
imp.numeroPaginas();		X	
((Libro) docu).getIsbn();			X

12.3. Solución en Ruby

```

1 class Documento
2   attr_reader :titulo, :fecha_publicacion
3
4   def initialize(titulo, fecha_publicacion)
5     @titulo = titulo
6     @fecha_publicacion = fecha_publicacion
7   end
8
9   def numero_paginas
10    raise NotImplementedError, 'Debe ser implementado en una
11      subclase'
12  end
13
14  def incluir_autores(autores)
15    # Implementación
16  end
17
18  def incluir_documentos_relacionados(documentos)
19    # Implementación
20  end
21
22 end

```

Listing 20: Clase Documento (cabecera, atributos y cabeceras de los métodos)

```

1 class Articulo < Documento
2   attr_reader :pagina_inicio, :pagina_fin
3
4   def initialize(titulo, fecha_publicacion, pagina_inicio, pagina_fin)
5     super(titulo, fecha_publicacion)
6     @pagina_inicio = pagina_inicio
7     @pagina_fin = pagina_fin
8   end
9
10  def numero_paginas
11    @pagina_fin - @pagina_inicio + 1
12  end
13
14  def imprimir
15    # Implementación
16  end
17
18  def imprimir_por_página(numero_página)
19    # Implementación
20  end
21
22  def imprimir_por_rango(página_inicio, página_fin)
23    # Implementación
24  end
25 end

```

Listing 21: Clase Articulo (cabecera, atributos y cabeceras de los métodos), constructor de la clase Artículo

```
1 module Imprimible
```

```
2     def imprimir
3         # Implementación
4     end
5
6     def imprimir_por_pagina(numero_pagina)
7         # Implementación
8     end
9
10    def imprimir_por_rango(pagina_inicio, pagina_fin)
11        # Implementación
12    end
13
```

Listing 22: Interfaz Imprimible

```
1 class BibliotecaElectronica
2     attr_accessor :autores, :documentos, :revistas
3 
```

Listing 23: Atributos que definen el estado de la clase BibliotecaElectronica

13 Ejercicio 13

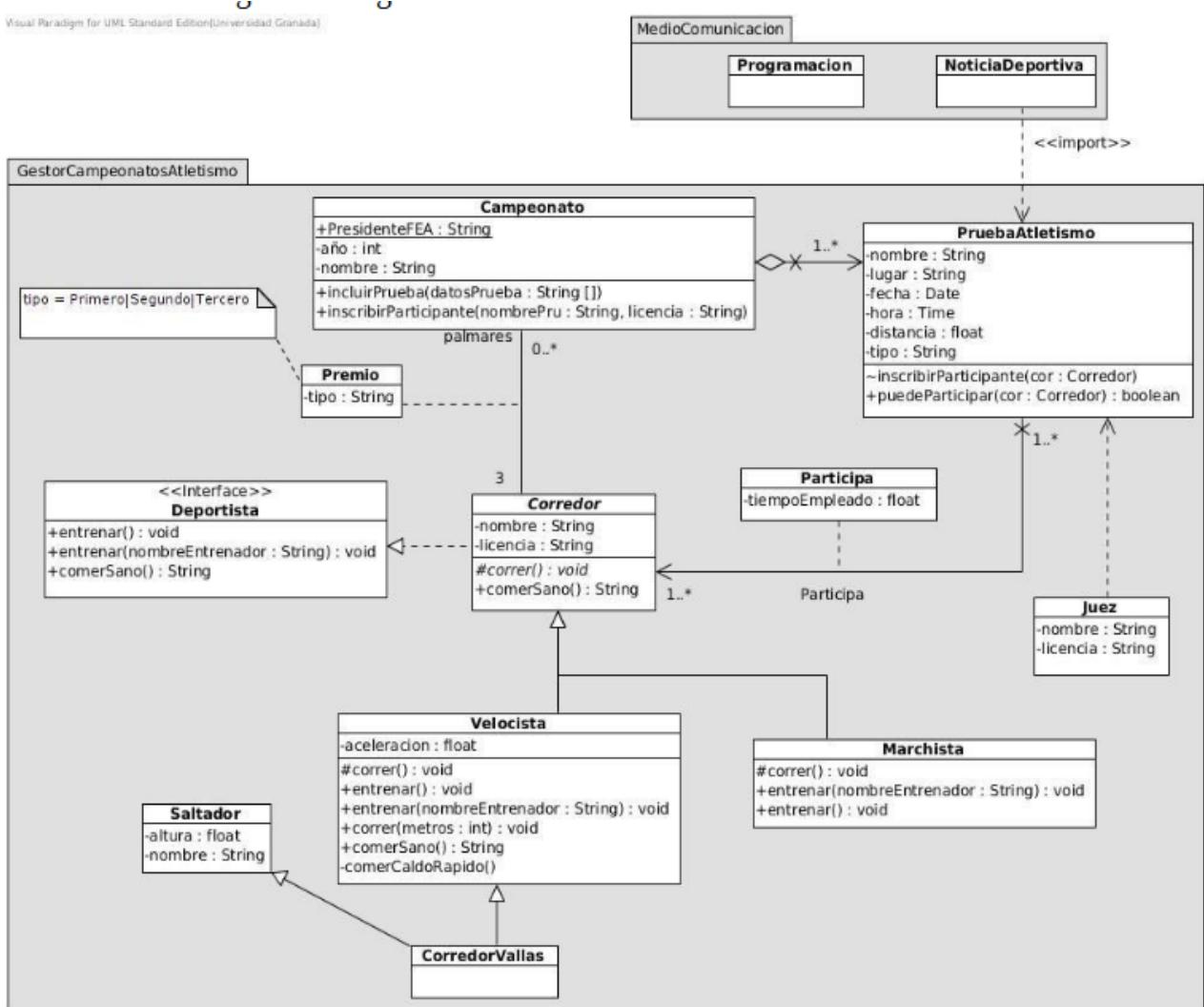


Figura 7: Diagrama de clases

13.1. Enunciado Y Solución

Cuestión	Respuesta
Desde la clase NoticiaDeportiva se puede acceder a todos los elementos públicos del paquete GestorCampeonatosAtletismo directamente	F
Un CorredorVallas es un Velocista y Saltador	V
El estado de un objeto de la clase Juez no viene determinado por el estado de un objeto de la clase PruebaAtletismo	V
Una PruebaAtletismo puede existir sin estar asociada a la clase Campeonato	V
Un Velocista es un Corredor	V
Un Deportista es un Corredor	F
La clase Corredor tiene 4 métodos, 3 abstractos y 1 concreto	F
La clase Marchista está mal representada en el diagrama, debe ser abstracta	F
La clase CorredorVallas presenta un conflicto de nombres	F
Todas las instancias de la clase Campeonato tienen una copia de la variable PresidenteFEA	V

Cuadro 2: Cuestiones Verdaderas o Falsas

Método	Abstracto
correr()	No, porque es una redefinición de Corredor
comerSano()	No, es una redefinición de la clase Corredor
entrenar()	No, porque es una redefinición de la interfaz Deportista
entrenar(nombreEntrenador:String)	No, porque es una redefinición de la interfaz Deportista

Cuadro 3: Métodos abstractos en Marchista

Método	Redefinido	Sobrecargado
correr()	X, porque cambia la cabecera	
comerSano()	X	
entrenar()	X, aunque sea de la interfaz	

Cuadro 4: Métodos redefinidos o sobrecargados en Velocista

	Variable	Tipo Estático	Tipo Dinámico
Deportista c = new Velocista();	c	Deportista	Velocista
Corredor d = new Marchista();	d	Corredor	Marchista
c = new Marchista();	c	Deportista	Marchista
d = c;	d	Corredor	Marchista

Cuadro 5: Tipo estático y dinámico de variables

Código	Corrección del error en Compilación	Error en ejecución
Corredor c = new Velocista();	Ninguno	Ninguno
Deportista d = new Marchista();	Ninguno	Ninguno
d.comerSano();	Ninguno	Ninguno
Marchista m = (Marchista) d;	Ninguno	Ninguno
d = c;	Ninguno	Ninguno
d.correr(150);	El método correr(int) no está definido en la interfaz Deportista	Ninguno
Velocista v = (Velocista) d;	Ninguno	Error en ejecución si d no es una instancia de Velocista
ArrayList<Corredor>corredores = new ArrayList<>();	Ninguno	Ninguno
corredores.add(c);	Ninguno	Ninguno
corredores.add(m);	Ninguno	Ninguno
corredores.get(0).correr(10);	El método correr(int) no está definido en Corredor	Ninguno
corredores.get(1).correr(10);	El método correr(int) no está definido en Corredor	Ninguno
c = new Corredor();	No se puede instanciar una clase abstracta	Ninguno

Cuadro 6: Corrección de errores de compilación y ejecución

Para una explicación más detallada pincha aquí.

```

1  public class Club<T> {
2      private Map<String, T> miembros; // key = número de licencia
3      private T lider;
4
5      // Constructor
6      public Club(T lider) {
7          this.lider = lider;
8          this.miembros = new HashMap<>();
9      }
10
11     // Consultor de un miembro del club a partir del numero de licencia

```

```

12 public T getMiembro(String numeroLicencia) {
13     return miembros.get(numeroLicencia);
14 }
15
16 // Incluir un nuevo miembro en el club
17 public void incluirMiembro(String numeroLicencia, T miembro) {
18     miembros.put(numeroLicencia, miembro);
19 }
20
21 // Cambiar el líder
22 public void setLider(T lider) {
23     this.lider = lider;
24 }
25
26 // Obtener el líder
27 public T getLider() {
28     return lider;
29 }
30

```

Listing 24: Código en java

```

1 import java.util.HashMap;
2 import java.util.Map;
3
4 public class Main {
5     public static void main(String[] args) {
6         // Creamos una instancia de Velocista
7         Velocista lider = new Velocista("Usain Bolt", "001", 9.58f);
8
9         // Creamos un club de Velocistas, con Usain Bolt como líder inicial
10        Club<Velocista> clubDeVelocistas = new Club<>(lider);
11
12        // Creamos más miembros del club
13        Velocista miembro1 = new Velocista("Carl Lewis", "002", 9.86f);
14        Velocista miembro2 = new Velocista("Tyson Gay", "003", 9.69f);
15
16        // Añadimos miembros al club
17        clubDeVelocistas.incluirMiembro("002", miembro1);
18        clubDeVelocistas.incluirMiembro("003", miembro2);
19
20        // Consultamos los miembros
21        System.out.println("Líder del club: " + clubDeVelocistas.getLider()
22                           .getNombre());
23        System.out.println("Miembro con licencia 002: " + clubDeVelocistas.
24                           getMiembro("002").getNombre());
25        System.out.println("Miembro con licencia 003: " + clubDeVelocistas.
26                           getMiembro("003").getNombre());
27
28        // Cambiamos el líder del club
29        clubDeVelocistas.setLider(miembro1);
30        System.out.println("Nuevo líder del club: " + clubDeVelocistas.
31                           getLider().getNombre());
32    }
33 }

```

14 EJERCICIO 14

```
31 // Clase Velocista
32 class Velocista {
33     private String nombre;
34     private String licencia;
35     private float mejorMarca;
36
37     // Constructor
38     public Velocista(String nombre, String licencia, float mejorMarca) {
39         this.nombre = nombre;
40         this.licencia = licencia;
41         this.mejorMarca = mejorMarca;
42     }
43
44     // Getters
45     public String getNombre() {
46         return nombre;
47     }
48
49     public String getLicencia() {
50         return licencia;
51     }
52
53     public float getMejorMarca() {
54         return mejorMarca;
55     }
56 }
```

Listing 25: Main

Listing 26: Salida del Main

```
Líder del club: Usain Bolt
Miembro con licencia 002: Carl Lewis
Miembro con licencia 003: Tyson Gay
Nuevo líder del club: Carl Lewis
```

14 Ejercicio 14

14.1. Enunciado

Dado el siguiente diagrama de clases:

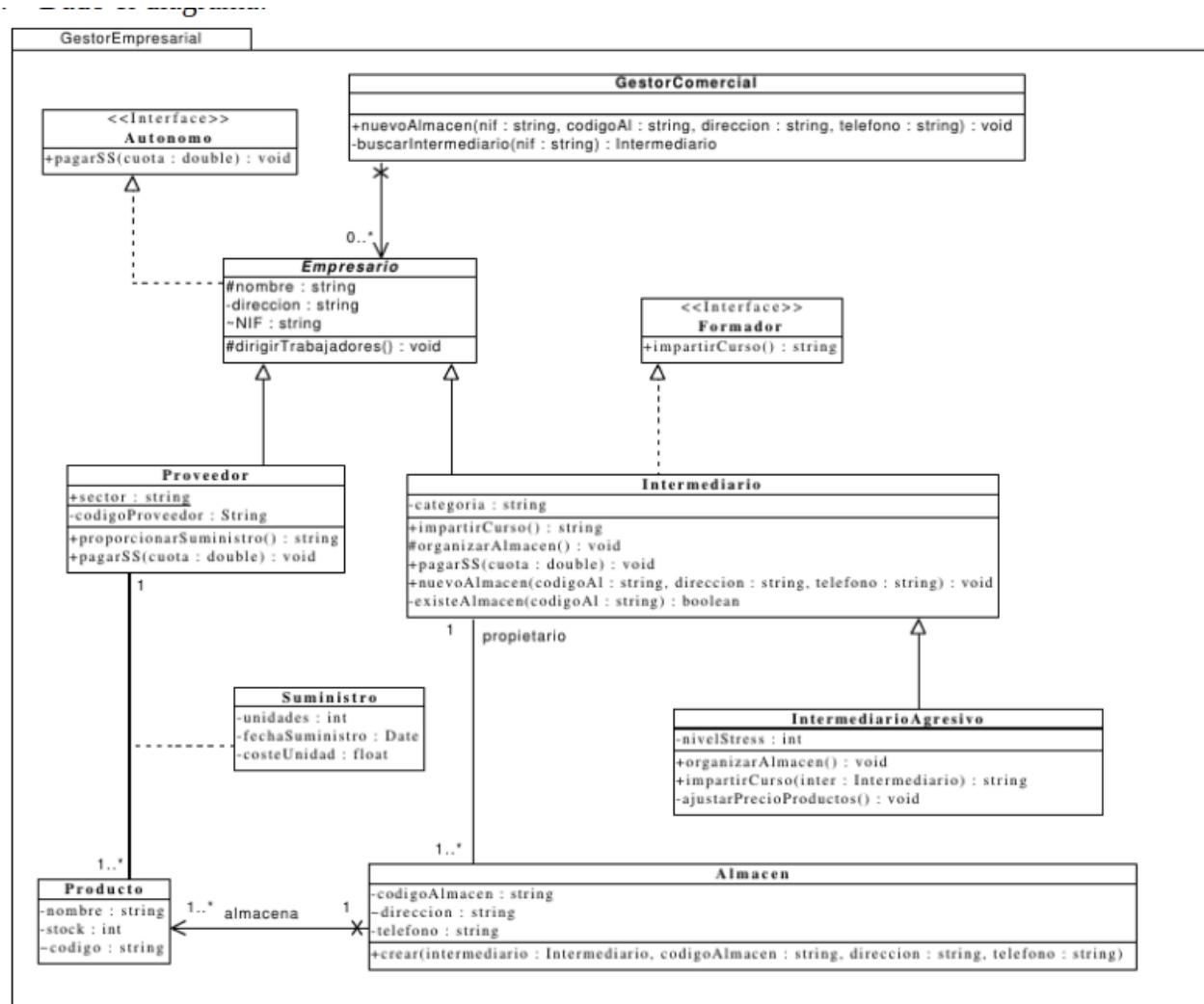


Figura 8: Diagrama de clases

Indica si las siguientes líneas de código Java producen error de compilación, de ejecución, ambos o ninguno. Supón que están en un `main` en una clase nueva dentro del mismo paquete. Si hay error explica cómo lo arreglarías y si no hay error, indícalo explícitamente. Considera para este ejercicio que todas las clases tienen un constructor válido que no recibe atributos.

Código	Error de compilación	Error de ejecución
Formador f = new Intermediario();	Ninguno	Ninguno
f.pagarSS(23.4);	Ninguno	Ninguno
Autonomo auto1 = f;	Sí	Ninguno
Autonomo auto2 = (Proveedor) f;	Sí	Ninguno
IntermediarioAgresivo emp1 = new IntermediarioAgresivo();	Ninguno	Ninguno
emp1.ajustarPrecioProductos();	Ninguno	Ninguno
Empresario emp2 = new Empresario();	Ninguno	Ninguno
ArrayList<Formador> formadores = new ArrayList();	Sí	Ninguno
formadores.add(f);	Ninguno	Ninguno
formadores.add(emp1);	Ninguno	Ninguno
formadores.get(1).impartirCurso();	Ninguno	Ninguno
formadores.get(1).impartirCurso(f);	Sí	Ninguno
((IntermediarioAgresivo)formadores.get(0)).impartirCurso(emp1);	Ninguno	Ninguno

14.2. Solución

Código	Error de compilación	Error de ejecución
Formador f = new Intermediario();	Ninguno	Ninguno
f.pagarSS(23.4);	Ninguno	Ninguno
Autonomo auto1 = f;	Sí	Ninguno
Autonomo auto2 = (Proveedor) f;	Sí	Ninguno
IntermediarioAgresivo emp1 = new IntermediarioAgresivo();	Ninguno	Ninguno
emp1.ajustarPrecioProductos();	Ninguno	Ninguno
Empresario emp2 = new Empresario();	Ninguno	Ninguno
ArrayList<Formador> formadores = new ArrayList();	Sí	Ninguno
formadores.add(f);	Ninguno	Ninguno
formadores.add(emp1);	Ninguno	Ninguno
formadores.get(1).impartirCurso();	Ninguno	Ninguno
formadores.get(1).impartirCurso(f);	Sí	Ninguno
((IntermediarioAgresivo)formadores.get(0)).impartirCurso(emp1);	Ninguno	Ninguno

Para una explicación más detallada pincha aquí.

3 Exámenes

3.1. Examen 1

PDOO. Parcial de Teoría hasta la Tema 2

Ismael Sallami Moreno

-
- **Asignatura:** PDOO
 - **Grado:** Doble Grado en Ingeniería Informática y ADE .
 - **Descripción:** Parcial de Teoría 1 y 2 de PDOO.
-

1. Preguntas

- Definir cabeceras de métodos en Java y saber si los atributos son privados o públicos en función de un diagrama de clases.
- Que significa included
- Usar métodos self para incrementar variables de clase
- Pasar a código un diagrama de secuencia