

# Objetos

Dpto. Lenguajes y Sistemas Informáticos  
Universidad de Granada

Programación y Diseño Orientado a Objetos

(Curso 2023-2024)

# 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)

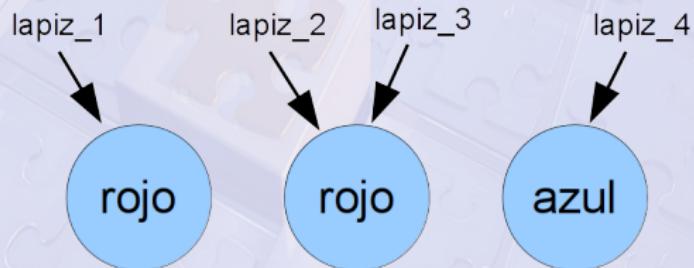
## Java: Instanciando clases, creando objetos

```
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 estado, 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?



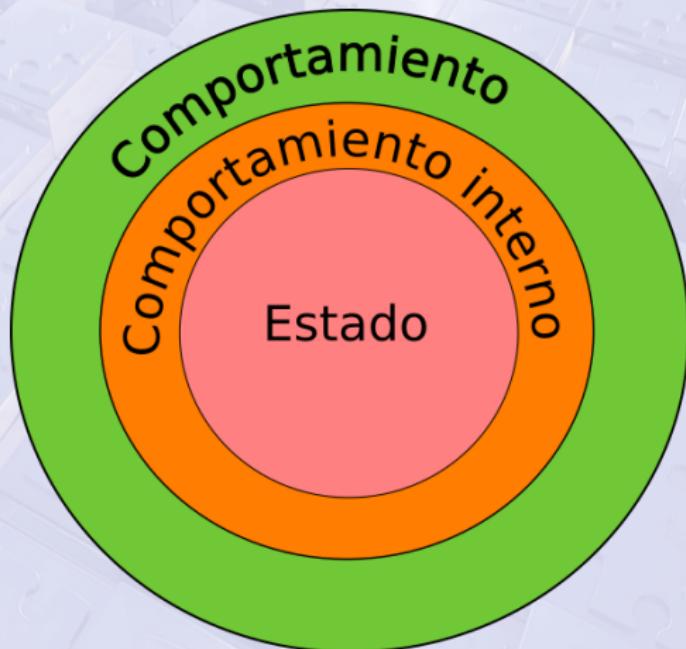
# Estado y Comportamiento

- 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**

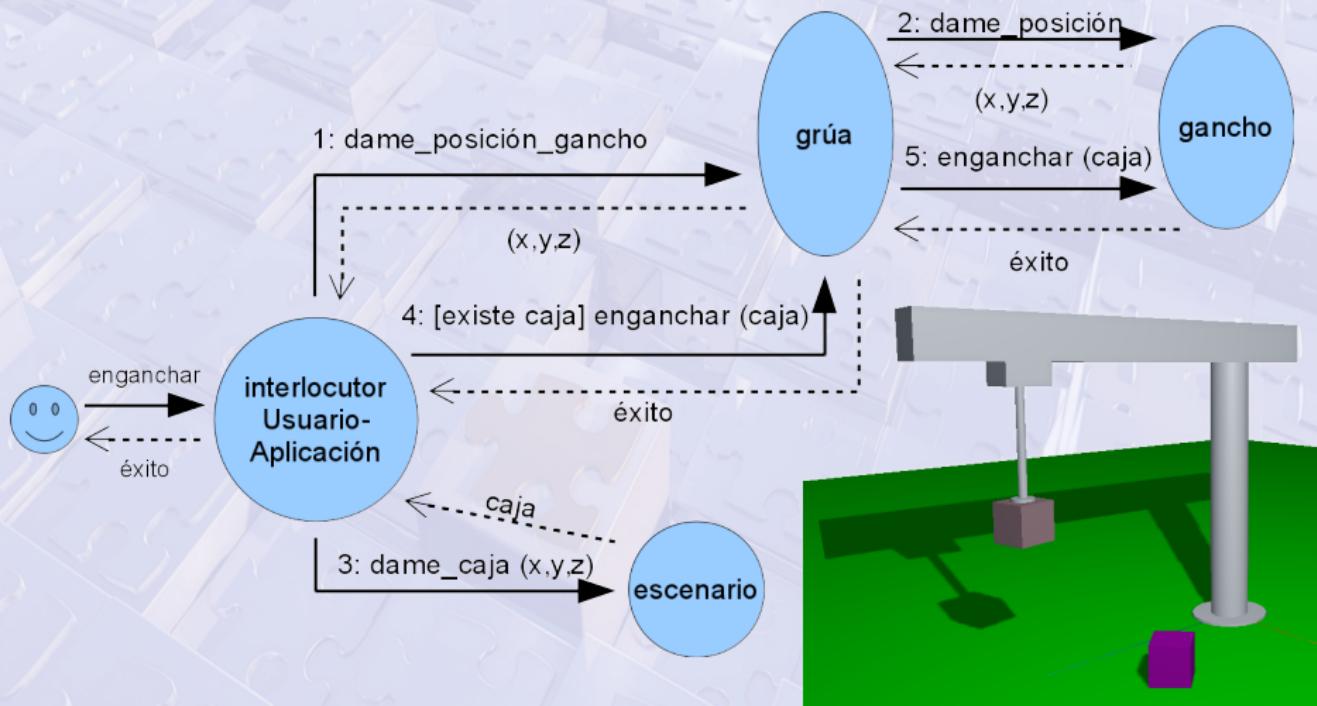
# Programación Orientada a Objetos (POO)

- 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

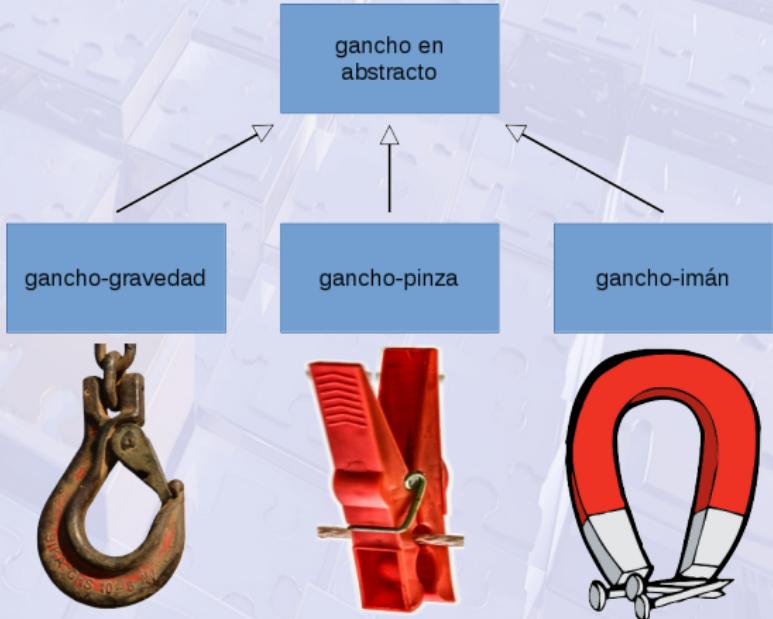


# POO

## Conceptos básicos

- El paradigma se basa en los siguientes conceptos:

- ▶ Clase
- ▶ Objeto o instancia
- ▶ Estado
- ▶ Identidad
- ▶ Mensaje
- ▶ Herencia
- ▶ Polimorfismo



# 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 >
```

- 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

Dpto. Lenguajes y Sistemas Informáticos  
Universidad de Granada

Programación y Diseño Orientado a Objetos

(Curso 2023-2024)

# Atributos y Métodos

Dpto. Lenguajes y Sistemas Informáticos  
Universidad de Granada

Programación y Diseño Orientado a Objetos

(Curso 2023-2024)

# 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 descripción 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   }

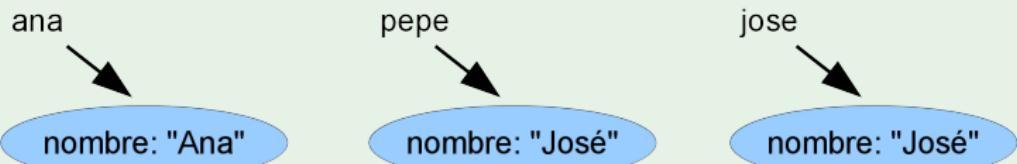
```

- 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    {
11         nombre = otroNombre;
12     }
13 }
```

## 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
```

ana

pepe

nombre: "Ana"

nombre: "José"

## Ejemplo: Contador de instancias

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

# Atributos de clase

→ **Diseño** ←

- ¿Cuándo usarlos?

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

- Ejemplos típicos:

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

★ ¿Quién sabría decirme qué es un número mágico?

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

# 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
10    puts @@variable
11    puts @variable
12  end
13
14  def self.muestraValores
15    puts @@variable
16    puts @variable
17  end
18 end
19
20 objeto = Clase.new
21 objeto.muestraValores
22 Clase.muestraValores
```

★ ¿Cuál es el resultado de ejecutar este programa?

→ Diseño ←

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

# Ejemplos

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

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

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

# 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 # En mayúscula sería constante
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?

# Ejemplos

## Ruby: Constantes

```
1 class Test
2   @ATTRIBUTO_INSTANCIA_CLASE_CTE=1
3   @@ATTRIBUTO_CLASE_CTE=2
4   CTE=3
5
6   def self.modifica
7     @ATTRIBUTO_INSTANCIA_CLASE_CTE=11 # en realidad no es una constante
8     @@ATTRIBUTO_CLASE_CTE=22 # en realidad no es una constante
9
10    CTE=33 # dynamic constant assignment. Error por ser constante
11    # Este error se notifica aunque no se ejecute este método
12
13  end
14
15  def self.muestra
16    puts @ATTRIBUTO_INSTANCIA_CLASE_CTE
17    puts @@ATTRIBUTO_CLASE_CTE
18    puts CTE
19  end
20
21 Test.muestra # 1 2 3
22 Test.modifica
23 Test.muestra # 11 22 3
24
25 # Las constantes no son privadas
26 puts Test::CTE
27
28 Test::CTE=333 # warning: already initialized constant Test::CTE
```

# 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 13
- ★ Otras formas de escribir la línea 13

## 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 to_s
17         return "Me llamo " + self.nombre
18     end
19 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 otro módulo

# 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

Dpto. Lenguajes y Sistemas Informáticos  
Universidad de Granada

Programación y Diseño Orientado a Objetos

(Curso 2023-2024)

# **Construcción de objetos**

Dpto. Lenguajes y Sistemas Informáticos  
Universidad de Granada

Programación y Diseño Orientado a Objetos

(Curso 2023-2024)

# 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 Mémoria 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

## ● 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
- Para construir un objeto se antepone la palabra reservada `new` al nombre de la clase
- Si no se crea ningún constructor existe uno por defecto sin parámetros

# 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
```

# Memoria dinámica y pila

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

# Memoria dinámica y pila: El lenguaje C++

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

## C++: Destructor

```

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

## C++: Pila y Heap

```

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



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



# **Construcción de objetos**

Dpto. Lenguajes y Sistemas Informáticos  
Universidad de Granada

Programación y Diseño Orientado a Objetos

(Curso 2023-2024)

# **Consultores y Modificadores**

Dpto. Lenguajes y Sistemas Informáticos  
Universidad de Granada

Programación y Diseño Orientado a Objetos

(Curso 2023-2024)

# 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  attr_reader :fecha_nacimiento      # consultor
12  attr_writer :fecha_nacimiento    # modificador
13  attr_accessor :fecha_nacimiento # consultor + modificador
14
15
16  def self.MAYORIA_EDAD=e      # modificador de clase
17    @@MAYORIA_EDAD = e
18  end
19 end
```

# 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   attr_reader :atr1
4   attr_accessor :atr2
5   attr_writer :atr3
6
7   def initialize (un, dos, tres)
8     @atr1 = un
9     @atr2 = dos
10    @atr3 = tres
11  end
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
```

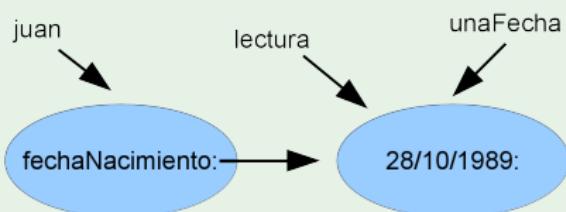
# Problemática de devolver (o asignar) referencias

## 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**

Dpto. Lenguajes y Sistemas Informáticos  
Universidad de Granada

Programación y Diseño Orientado a Objetos

(Curso 2023-2024)

## **Temporary page!**

$\text{\LaTeX}$  was unable to guess the total number of pages correctly.  
was some unprocessed data that should have been added to  
page this extra page has been added to receive it.  
If you rerun the document (without altering it) this surplus page  
away, because  $\text{\LaTeX}$  now knows how many pages to expect in  
document.

# Elementos de Agrupación

Dpto. Lenguajes y Sistemas Informáticos  
Universidad de Granada

Programación y Diseño Orientado a Objetos

(Curso 2023-2024)

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

# Contenidos

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

# 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

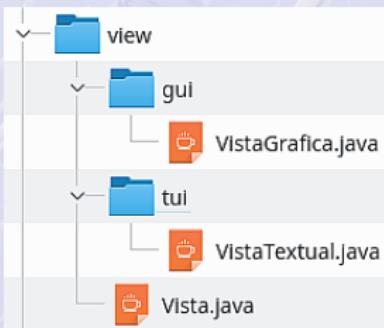
```
1 package miPrograma;  
2 // Los elementos que se definan en este archivo pertenecerán al paquete miPrograma  
3 import modelo.Fachada; // En este fichero se va a usar la clase Fachada del paquete modelo
```

# 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 public 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 puede quitar la palabra **public**?

# Módulos Ruby

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

# Módulos Ruby

## Ejemplo: Ruby

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

# Un proyecto Ruby definido en varios archivos

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

# Referenciando archivos Ruby

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

# Ejemplo

## : cosa.rb

```

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

```

## : persona.rb

```

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

```

## : principal.rb

```

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

```

# Mal ejemplo

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



:cosa.rb

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

## Ejecución: Mensaje de error obtenido

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

★ Trazémoslo y averigüemos el porqué

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

# Elementos de Agrupación

Dpto. Lenguajes y Sistemas Informáticos  
Universidad de Granada

Programación y Diseño Orientado a Objetos

(Curso 2023-2024)

## **Temporary page!**

$\text{\LaTeX}$  was unable to guess the total number of pages correctly.  
was some unprocessed data that should have been added to  
page this extra page has been added to receive it.  
If you rerun the document (without altering it) this surplus page  
away, because  $\text{\LaTeX}$  now knows how many pages to expect in  
document.

# **UML: Diagramas Estructurales**

Dpto. Lenguajes y Sistemas Informáticos  
Universidad de Granada

Programación y Diseño Orientado a Objetos

(Curso 2023-2024)

# 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

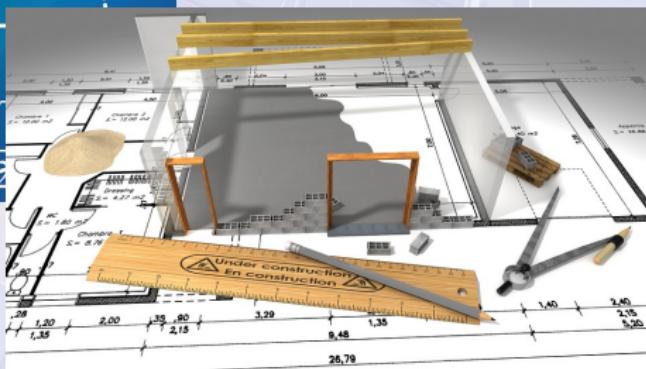
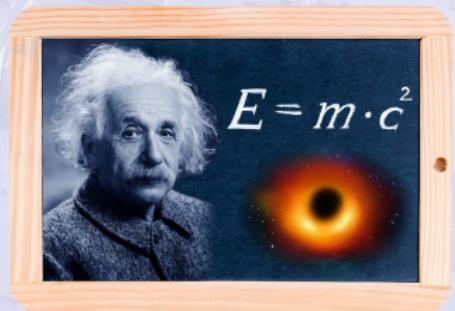
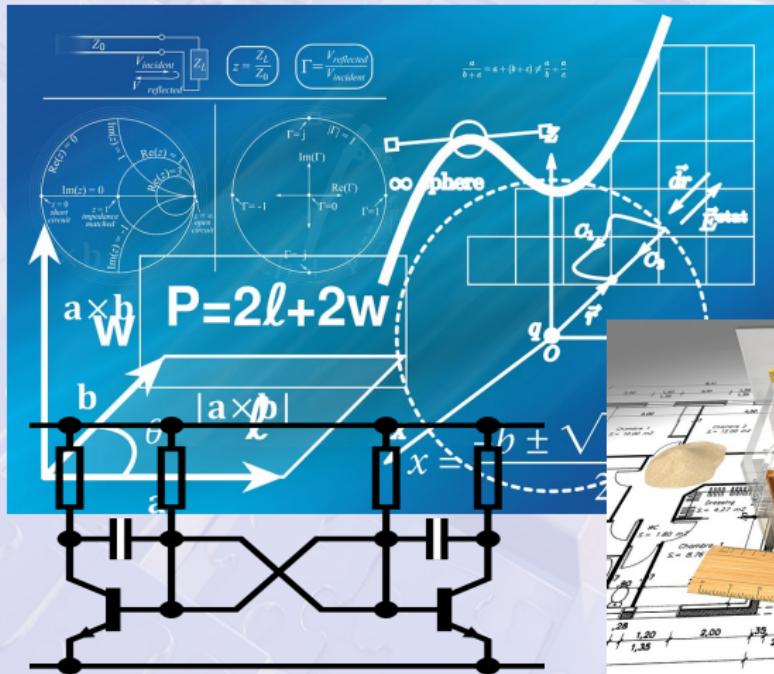
## 3 Relaciones entre clases

- 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 (versión 11/Universidad de 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

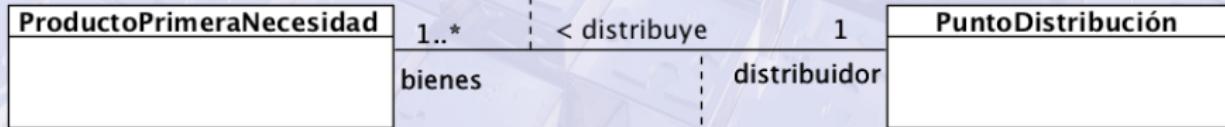


- ▶ Modela una **relación estructural fuerte y duradera en el tiempo**
- ▶ Las asociaciones **generan atributos de referencia**
- **Error MUY común:** No añadir atributos de referencia
- ▶ **Cardinalidad / multiplicidad:**
  - ★ Se representa con números (pueden definir un rango)
  - ★ Indica cuántas 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
- ▶ **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

# Ejemplo de asociación

Visual Paradigm Standard/Zoraida Callejas/Universidad Granada)

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



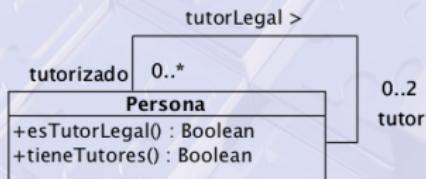
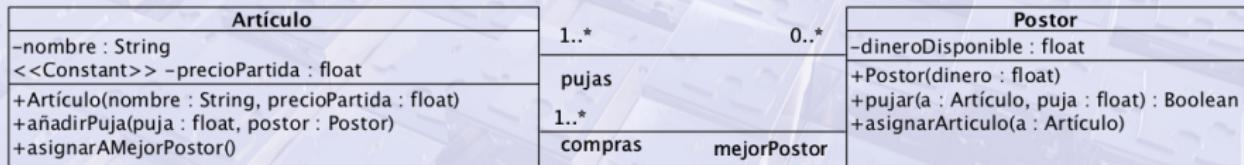
En cada extremo se puede indicar los nombres de los roles de las clases en la asociación

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

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

# Ejemplos de asociaciones

Visual Paradigm Standardizada (Universidad Granada)



# Clases asociación

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

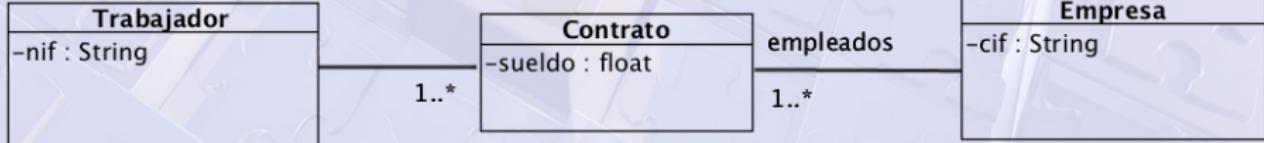
Visual Paradigm Standard (zoraida (Universidad Granada))



# Clases asociación

- Ambos diagramas son equivalentes

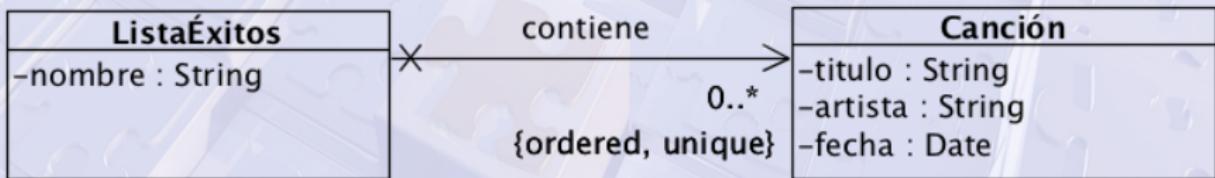
Visual Paradigm Standard (versión 10.0) (Universidad de Granada)



# Asociación: Propiedades de los extremos

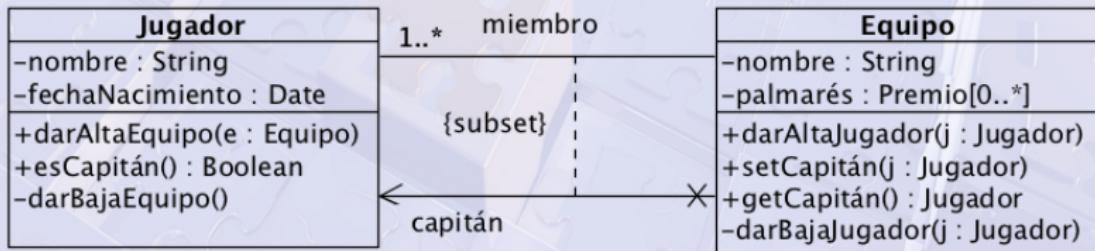
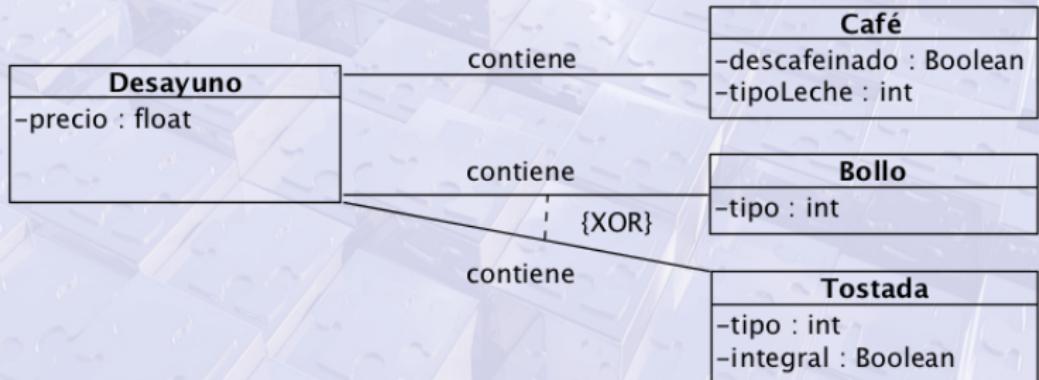
- En los extremos de las asociaciones se pueden indicar propiedades
- Las más comunes con multiplicidad mayor a 1 son:
  - {ordered} para indicar que se trata de una secuencia ordenada
  - {unique} para indicar que los elementos no se repiten

Visual Paradigm Standard(zoraida(Universidad Granada))



# Especificaciones de las asociaciones

Visual Paradigm Standard (Zoraida (Universidad Granada))

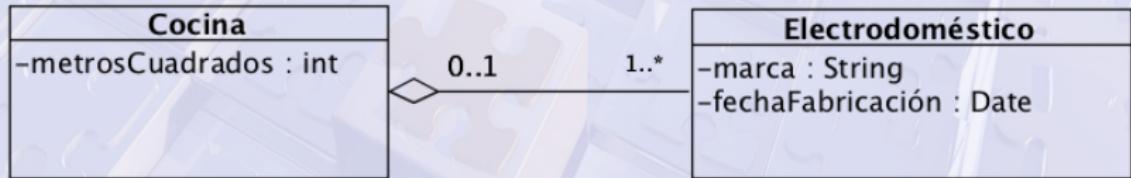


# Asociaciones especiales

## • Agregación



- ▶ Una de las clases representa el TODO y 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

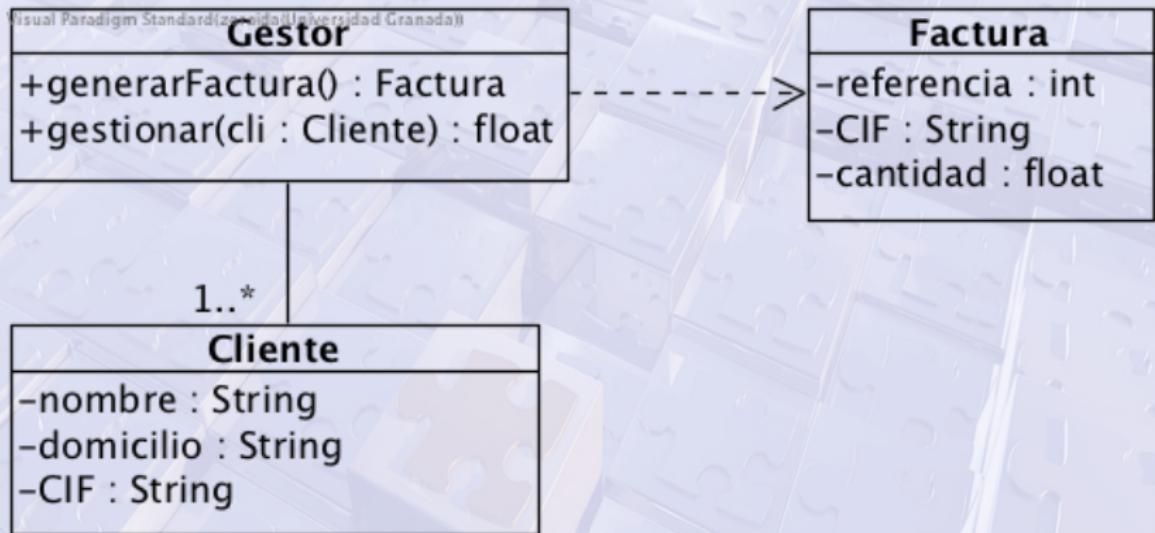


# Relaciones entre clases

## • Dependencia

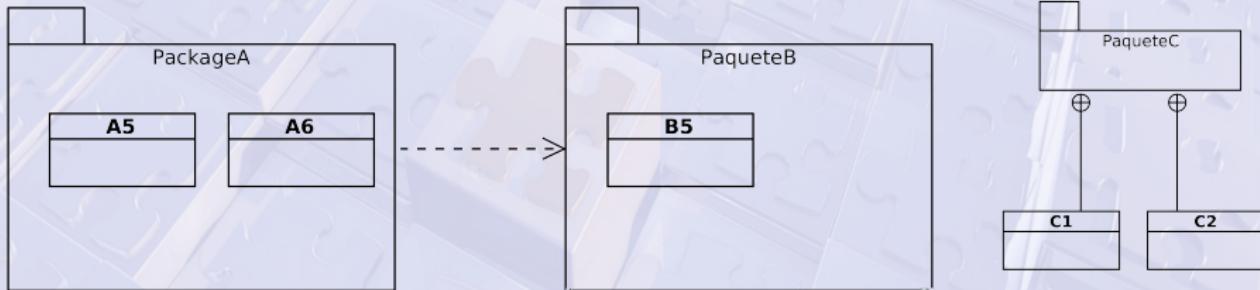
- ▶ Modela una relación débil y poco duradera en el tiempo
  - ▶ Cuando desde una clase *se utilizan* instancias de otra clase
  - ▶ Ejemplos
    - ★ Un método de una clase recibe como parámetros instancias de otra clase
    - ★ Un método de una clase devuelve una instancia de otra clase
  - ▶ Si se 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*

- 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**

Dpto. Lenguajes y Sistemas Informáticos  
Universidad de Granada

Programación y Diseño Orientado a Objetos

(Curso 2023-2024)

## **Temporary page!**

$\text{\LaTeX}$  was unable to guess the total number of pages correctly.  
was some unprocessed data that should have been added to  
page this extra page has been added to receive it.  
If you rerun the document (without altering it) this surplus page  
away, because  $\text{\LaTeX}$  now knows how many pages to expect in  
document.

# UML: Diagramas de Interacción

Dpto. Lenguajes y Sistemas Informáticos  
Universidad de Granada

Programación y Diseño Orientado a Objetos

(Curso 2023-2024)

# 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

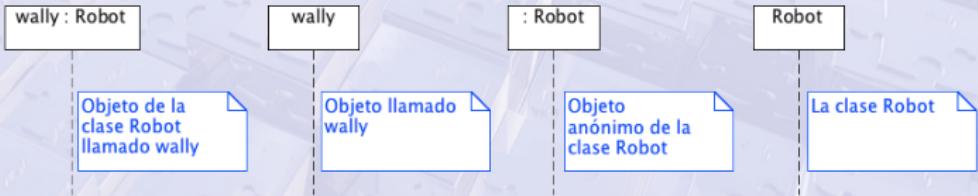
# Diagramas de interacción

- 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 colaboración:** Enfatizan la relación entre los objetos receptores y emisores de los mensajes
- Elementos:
  - ▶ Participantes: Objetos y clases que forman parte de la interacción
  - ▶ Mensajes: El flujo y su secuencia entre los participantes

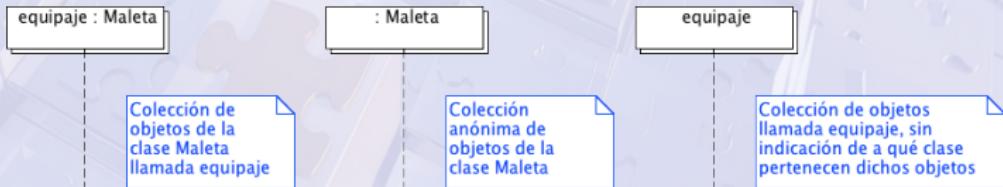
# Diagramas de secuencia

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

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

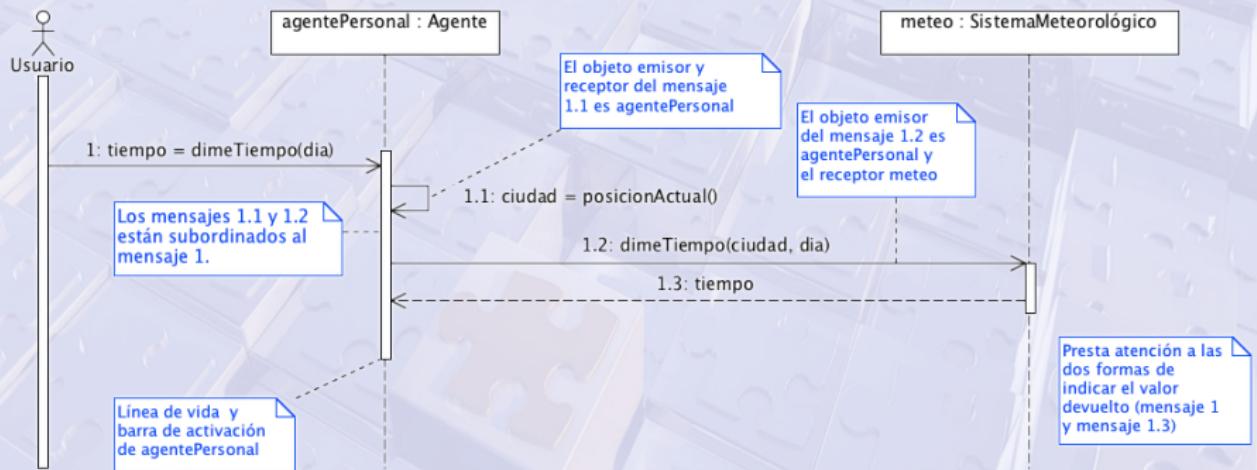


Los multiobjetos o colecciones de objetos se representan con un doble fondo



# Diagramas de secuencia

- **Mensajes:** Emisor y Receptor



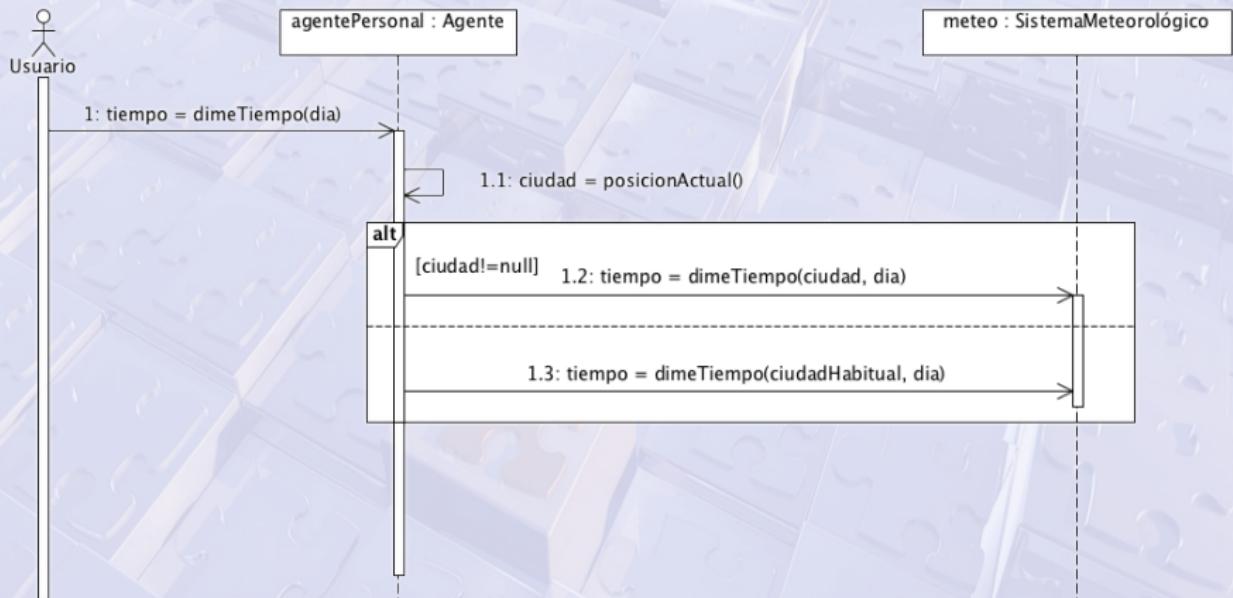
# Diagramas de secuencia

## Ruby: Implementación del diagrama anterior

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

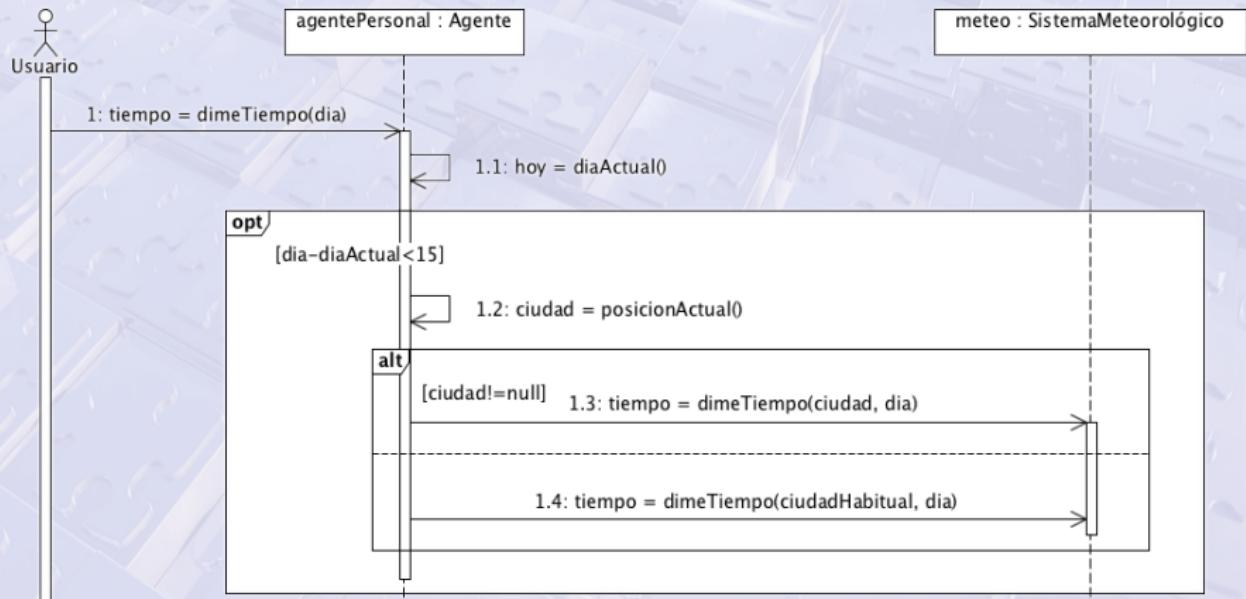
# Diagramas de secuencia

- Fragmentos: **Condicionales**



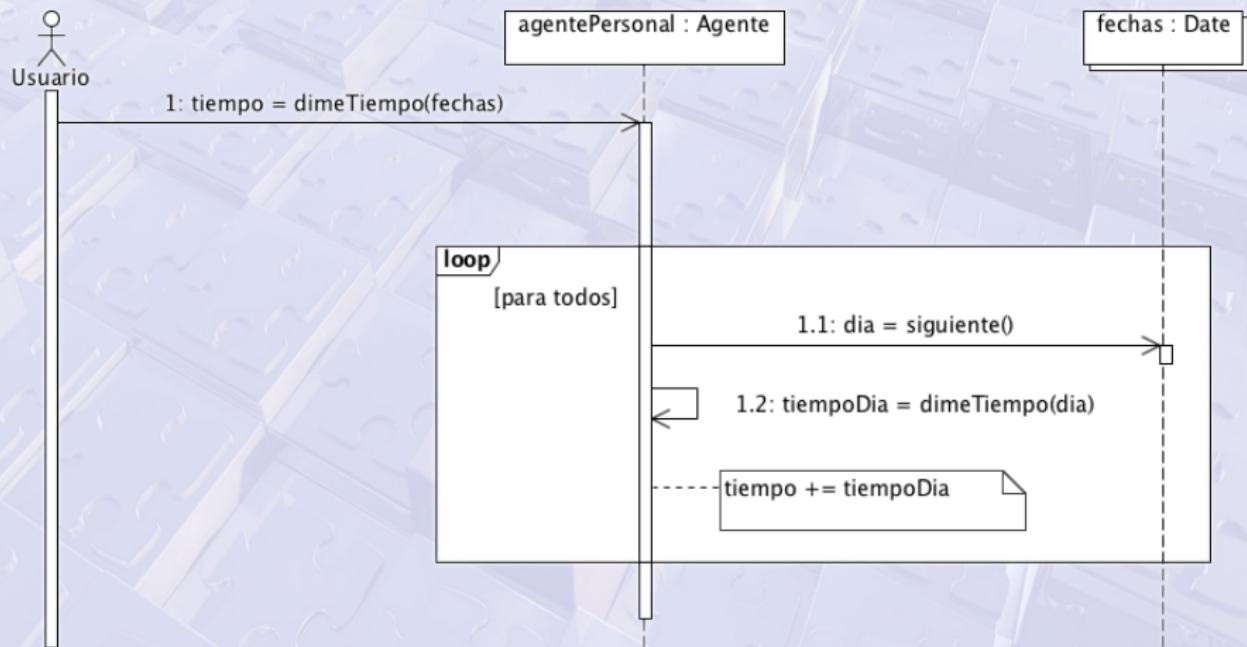
# Diagramas de secuencia

- Fragmentos: Condicionales



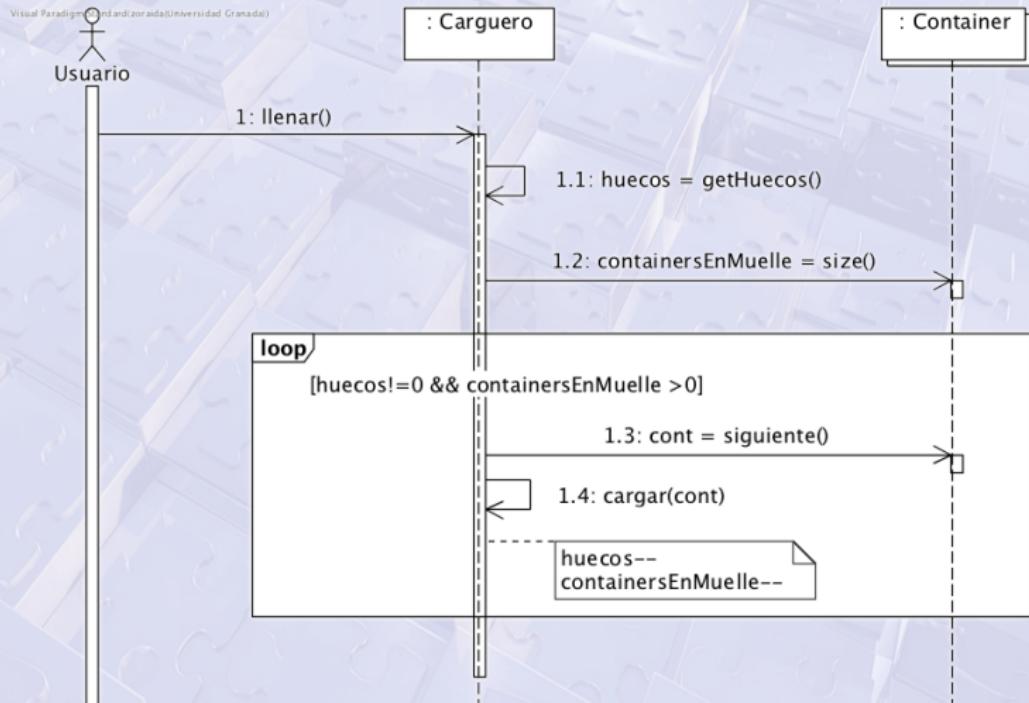
# Diagramas de secuencia

## • Fragmentos: Bucles



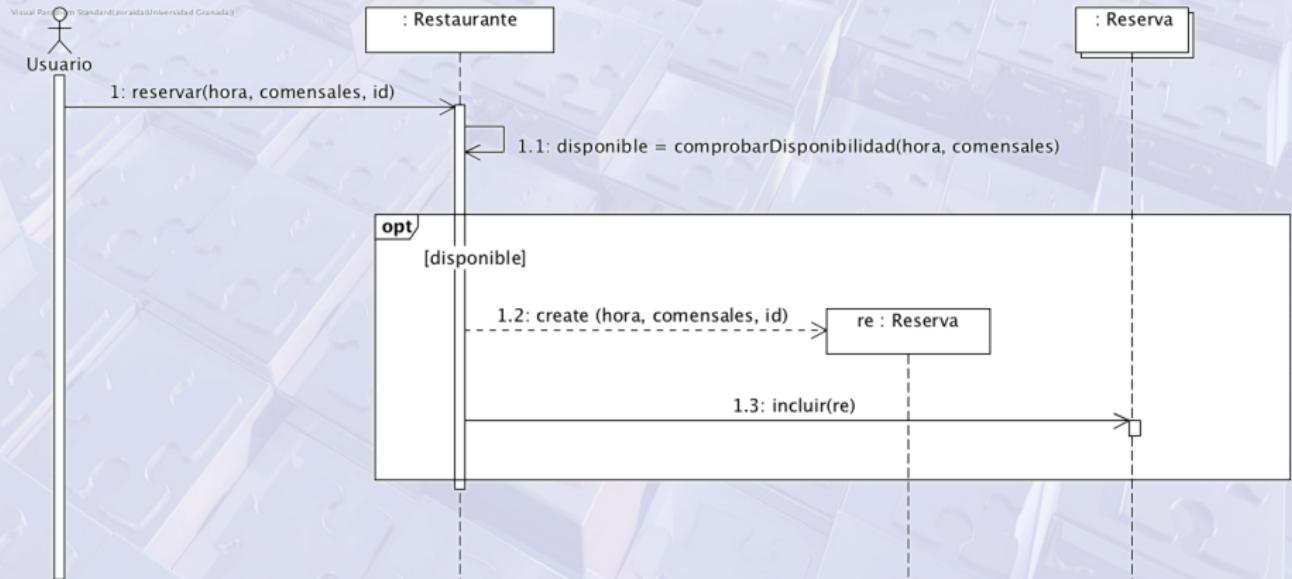
# Diagramas de secuencia

- Fragmentos: Bucles



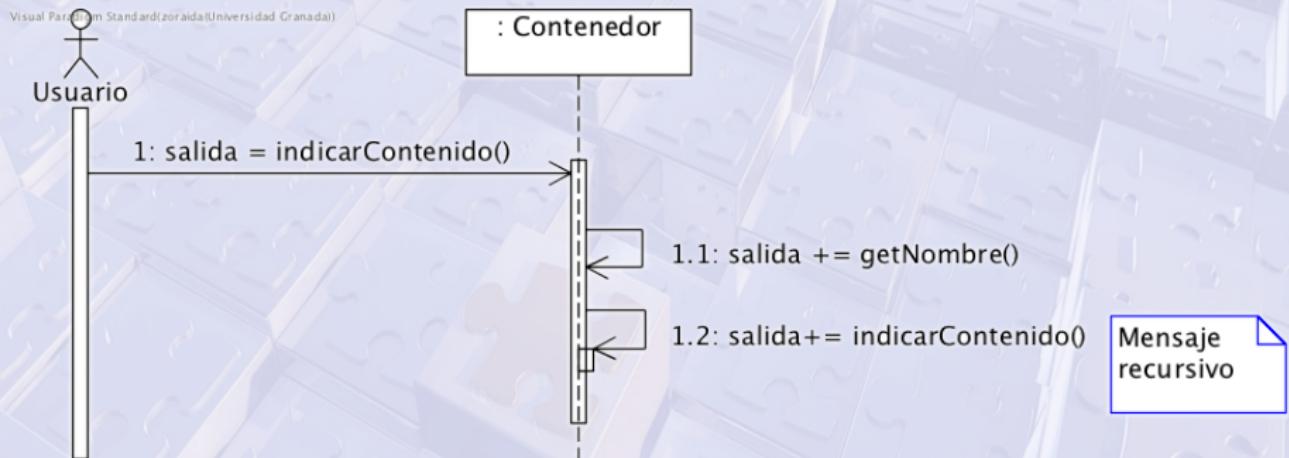
# Diagramas de secuencia

## ● Creación de instancias



# Diagramas de secuencia

## • Recursividad



# Diagramas de comunicación

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

# Diagramas de comunicación

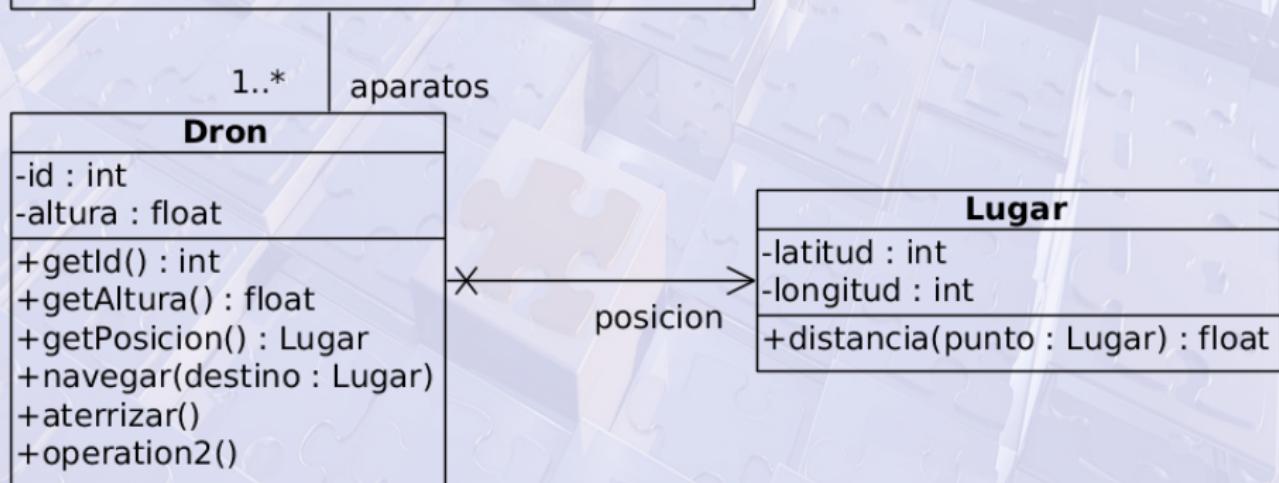
- Las vías de comunicación se representan mediante líneas que unen a los participantes
- Tipos de enlaces:
  - ▶ Global (G): Uno de los participantes pertenece a un ámbito superior. Ej: un atributo de clase
  - ▶ Asociación (A): Entre los participantes existe una asociación
  - ▶ Parámetro (P): Uno de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

# DC para los ejemplos siguientes

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

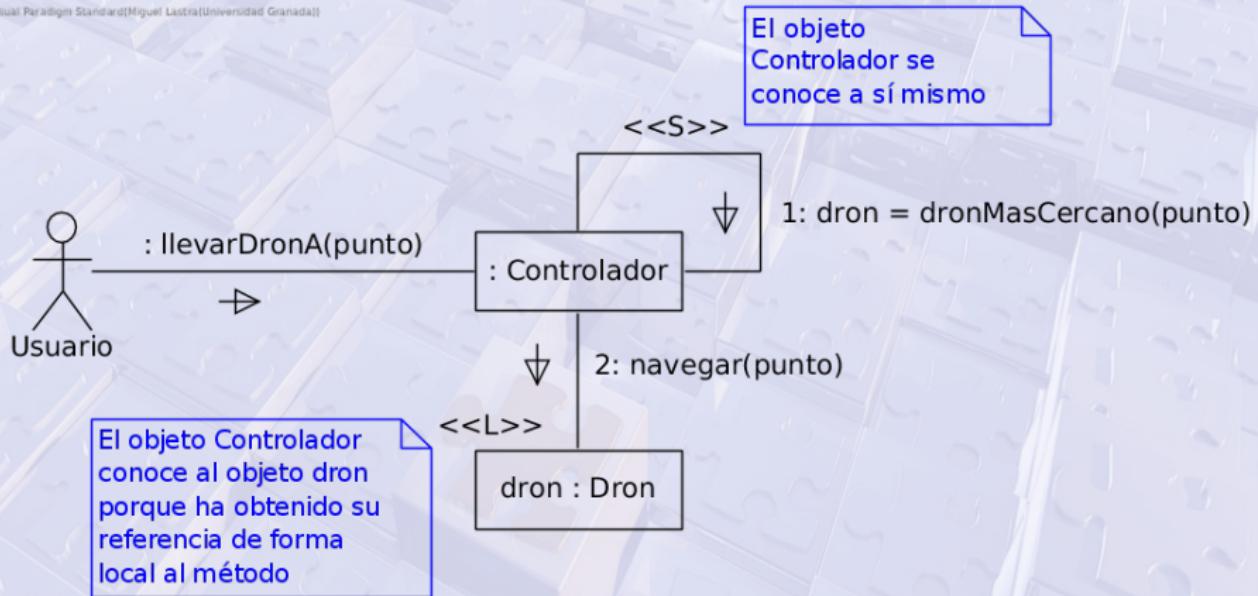
## Controlador

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



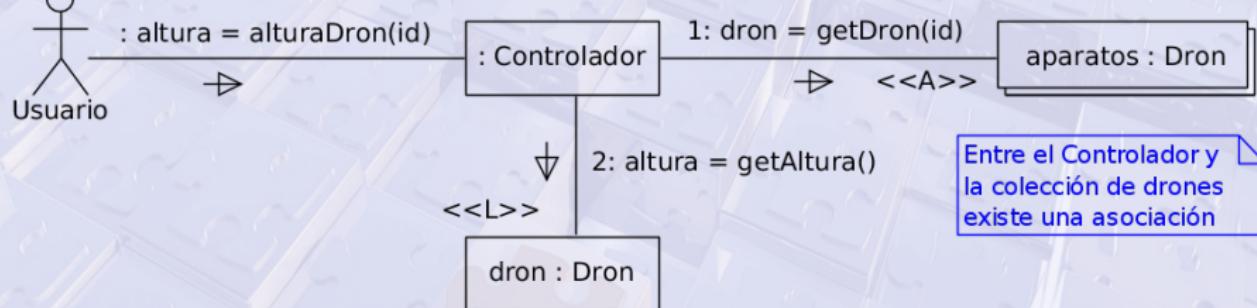
# Ejemplo 1

Visual Paradigm Standard (Miguel Lastra (Universidad Granada))



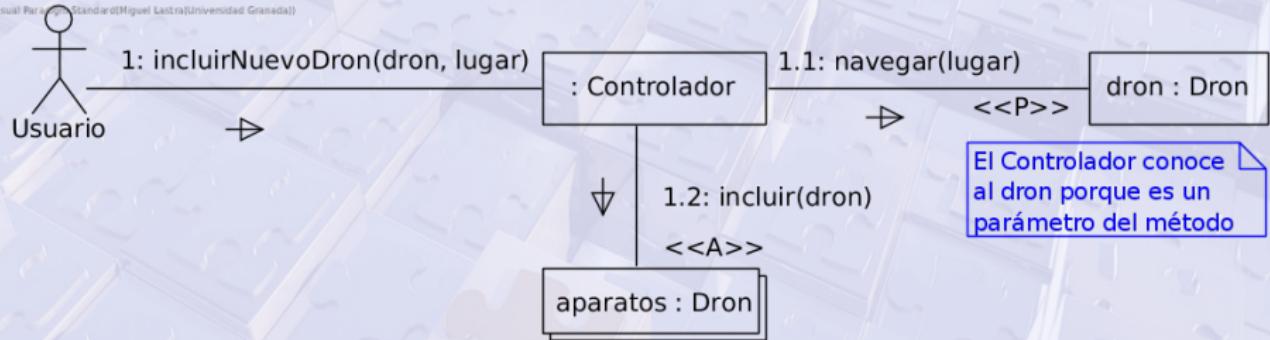
# Ejemplo 2

Visual Paragon Standard (Miguel Lastra (Universidad Granada))



# Ejemplo 3

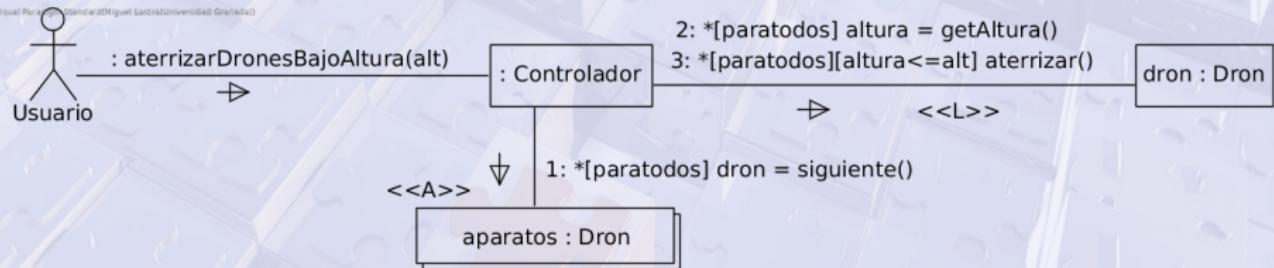
Visual Parámetros Standard[Miguel Lastra/Universidad Granada]



# Ejemplo 4

## • Condicionales y bucles

Miguel Peris (mperis@ugr.es) Miguel Peris (Universidad de Granada)



- 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

Dpto. Lenguajes y Sistemas Informáticos  
Universidad de Granada

Programación y Diseño Orientado a Objetos

(Curso 2023-2024)

## **Temporary page!**

$\text{\LaTeX}$  was unable to guess the total number of pages correctly.  
was some unprocessed data that should have been added to  
page this extra page has been added to receive it.  
If you rerun the document (without altering it) this surplus page  
away, because  $\text{\LaTeX}$  now knows how many pages to expect in  
document.

# **Herencia**

Dpto. Lenguajes y Sistemas Informáticos  
Universidad de Granada

**Programación y Diseño Orientado a Objetos**

**(Curso 2023-2024)**

# 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 Ejemplos

- Java
- Ruby

## 6 Diagramas de clases con herencia

# Introducción al concepto de herencia

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

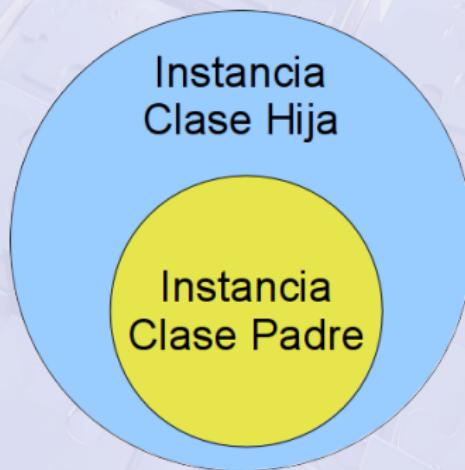


# La relación de herencia es una relación **es-un**

- 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?

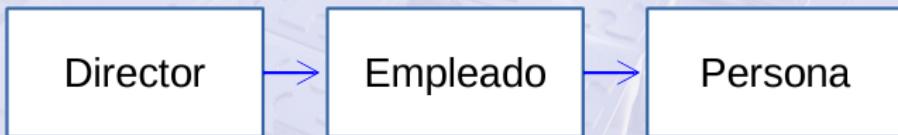
# ¿Qué se hereda?

- 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
```

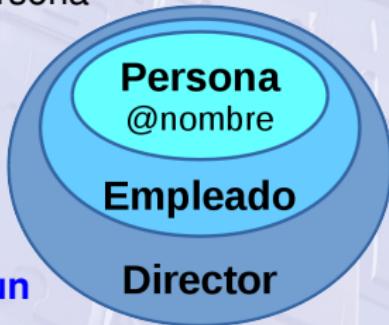
```
class Persona
  . . .
  def nombre
    @nombre
  end
end
```

# La herencia como composición (3)

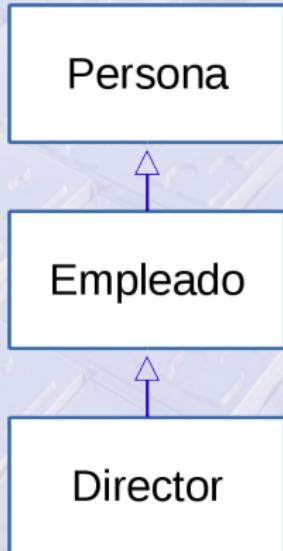
- (continuación)



- ▶ 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)



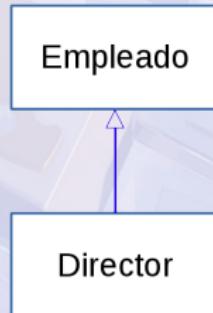
## Ruby: Implementación mediante herencia

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

★ ¿Cuál es el resultado de la ejecución?

# ¿Por qué se usa herencia?

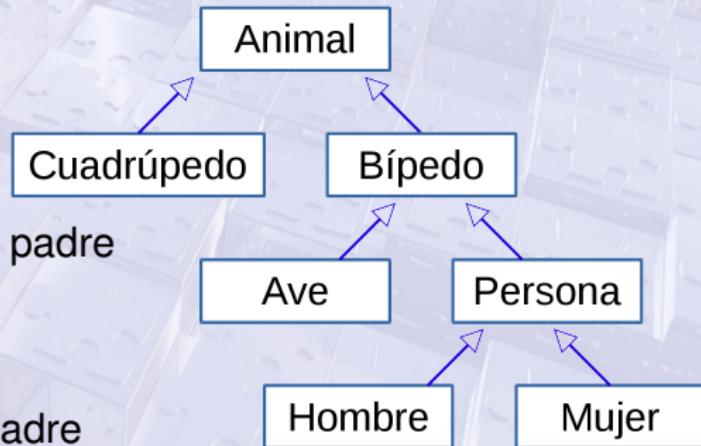
- Reutilización de código, normalmente
  - ▶ La clase derivada añade y/o modifica el comportamiento de la clase padre
- La clase padre puede verse como una generalización de sus descendientes
- Una clase hija puede verse como una especialización de la clase padre



# Criterios válidos

- **Especificación**

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



- **Especialización**

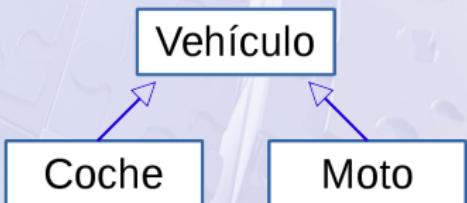
Las clases hijas **modifican** el comportamiento de la clase padre

- **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 otro módulo)



# 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

- Permite acceder a la implementación de cualquier método proporcionado por la clase padre
- Permite referenciar al constructor de la clase padre. **Debe aparecer en la primera línea del constructor** de la clase derivada

## Java: super en métodos

```

1 int metodo1 (int i, int j) {
2     int a = super.metodo1 (i,j);
3     // NO recomendable si se ha redefinido
        metodo2
4     // Totalmente innecesario si no se ha
        redefinido metodo2
5     int b = super.metodo2();
6     return (a+b);
7 }
```

No usarlo para llamar a métodos  
distintos del actual

## Java: super en constructor

```

1 class Persona {
2     private String nombre;
3     Persona (String nombre) {
4         this.nombre = nombre;
5     }
6 }
7 class Profesor extends Persona {
8     private String asignatura;
9     Profesor (String nomb, String asign) {
10        super (nomb);           // primera linea
11        asignatura = asign;
12    }
13 }
```

# 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
```

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

# Ejemplo en Java

## Java: Uso de super

```
1 class Persona {  
2     private String nombre;  
3     public Persona(String n) { // Constructor con parámetros  
4         nombre=n;  
5     }  
6     public String andar() { return ("Ando como una persona"); }  
7     public String hablar() {return ("Hablo como una persona"); }  
8 }  
9 class Profesor extends Persona {  
10    private String asignatura;  
11    public Profesor(String n, String a) {  
12        super(n);          // Esta llamada es obligatoria ¿por qué?  
13        // Además debe ser la primera línea  
14        asignatura = a;  
15    }  
16    @Override  
17    public String hablar() { return ("Estimados alumnos: \n" + super.hablar()); }  
18 }  
19  
20 // *****  
21  
22 public static void main(String[] args) {  
23     System.out.println((new Profesor("Jaime","PDDO")).hablar());  
24 }
```

# 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?

## Llamada a super en initialize

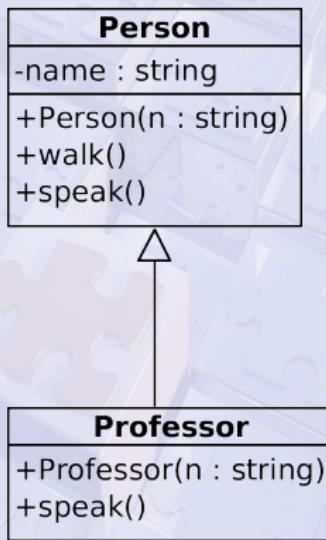
- La responsabilidad de llamar a super es nuestra
  - ▶ La llamada no se produce implícitamente
- No hacer la llamada explícita no produce un error por sí mismo
- Aunque puede *inducir* otros errores

### Ruby: Llamada a super en initialize

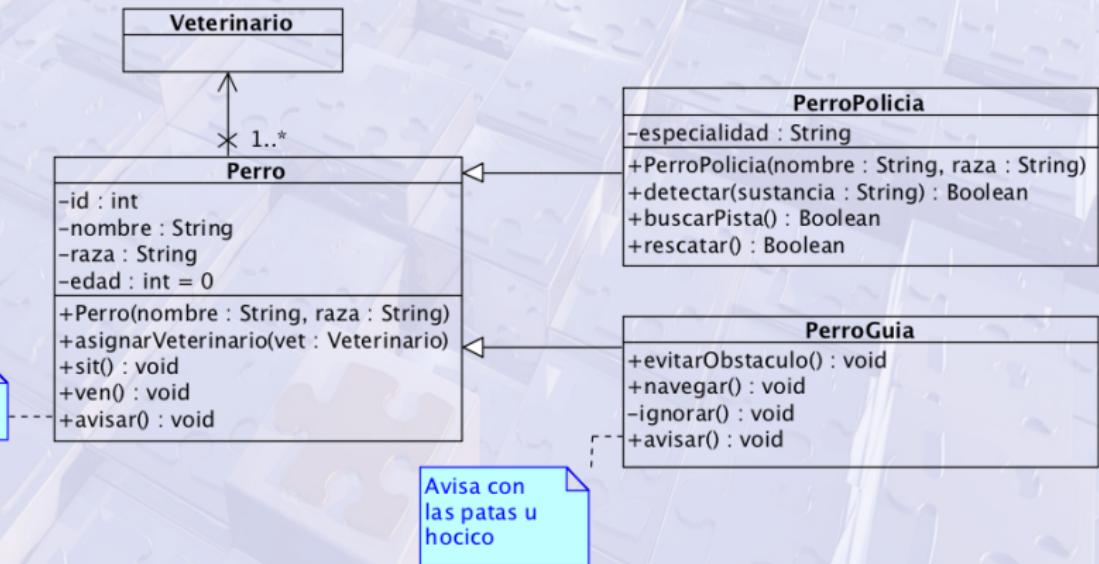
```
1 class Padre
2   def initialize
3     @atributo = 33
4   end
5
6   def metodo
7     puts @atributo + 1
8   end
9 end
10
11 class Hija < Padre
12   def initialize
13   end
14 end
15
16 Padre.new.metodo # ¿Qué ocurre aquí?
17 Hija.new.metodo # ¿Y aquí?
```

# 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

# **Herencia**

Dpto. Lenguajes y Sistemas Informáticos  
Universidad de Granada

**Programación y Diseño Orientado a Objetos**

**(Curso 2023-2024)**

## **Temporary page!**

$\text{\LaTeX}$  was unable to guess the total number of pages correctly.  
was some unprocessed data that should have been added to  
page this extra page has been added to receive it.  
If you rerun the document (without altering it) this surplus page  
away, because  $\text{\LaTeX}$  now knows how many pages to expect in  
document.

# Especificadores de acceso (Visibilidad)

Dpto. Lenguajes y Sistemas Informáticos  
Universidad de Granada

Programación y Diseño Orientado a Objetos

(Curso 2023-2024)

# 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

- Ejemplos

## 3 Especificadores de acceso en Ruby

- Ejemplos

# 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
  - ▶ Protegido
  - ▶ Público
- 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)

# 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

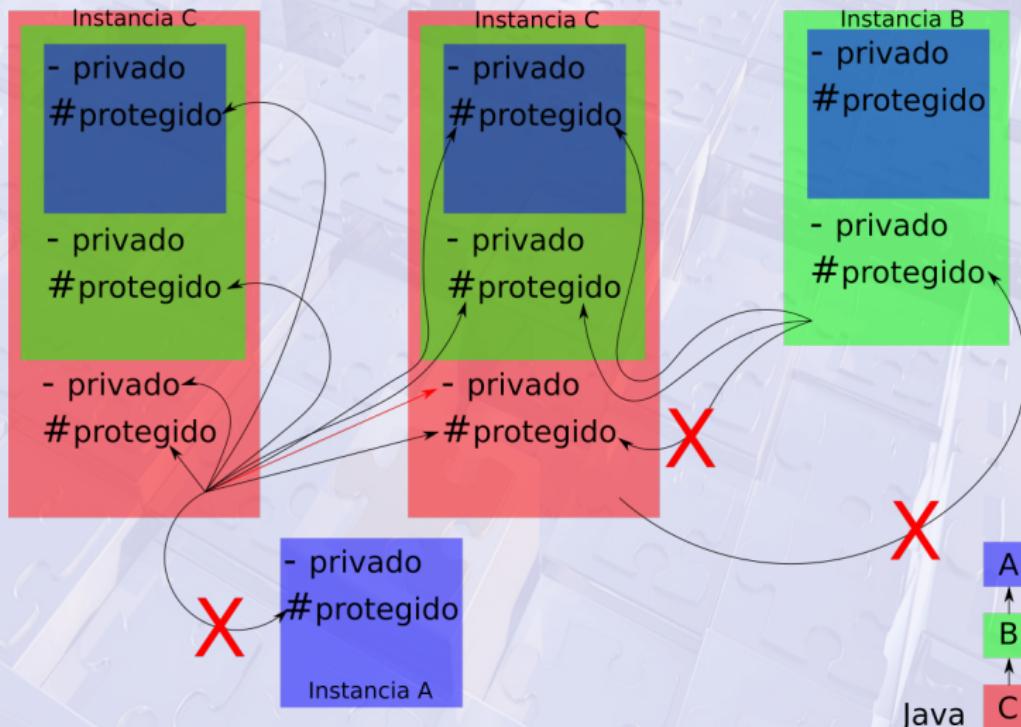
- Particularidades del especificador **protected** (continuación)
- Para poder acceder a  
**elementos protegidos de una instancia distinta:**  
(tanto desde ámbito de clase como de instancia)
  - ▶ Esa instancia tiene que ser de la misma clase que la propietaria del código desde el que se realiza el 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 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 (protectido);
10        System.out.println (o.protectido);
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.protectido);
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

- 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

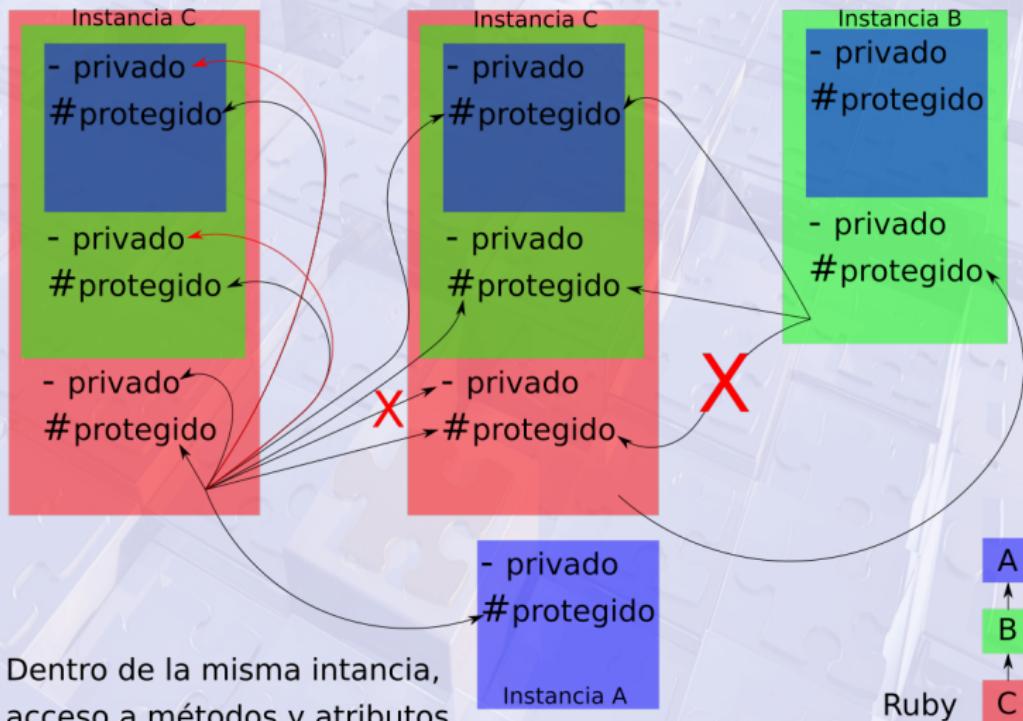
# Especificadores de acceso en Ruby

- Particularidades del especificador **protected**
  - ▶ Los métodos protegidos sí pueden ser invocados con un receptor de mensaje explícito
    - ★ La clase del código que invoca debe ser la misma, o una subclase, de la clase donde se declaró 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 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 # Receptor explícito 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 end
20
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
18 class D < C
19   protected
20   def protegidoD
21 end
22
23 C.new.test
24 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

→ **Diseño** ←

- 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

# Especificadores de acceso (Visibilidad)

Dpto. Lenguajes y Sistemas Informáticos  
Universidad de Granada

Programación y Diseño Orientado a Objetos

(Curso 2023-2024)

## **Temporary page!**

$\text{\LaTeX}$  was unable to guess the total number of pages correctly.  
was some unprocessed data that should have been added to  
page this extra page has been added to receive it.  
If you rerun the document (without altering it) this surplus page  
away, because  $\text{\LaTeX}$  now knows how many pages to expect in  
document.

# **Clases Abstractas e Interfaces**

Dpto. Lenguajes y Sistemas Informáticos  
Universidad de Granada

Programación y Diseño Orientado a Objetos

(Curso 2023-2024)

# 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

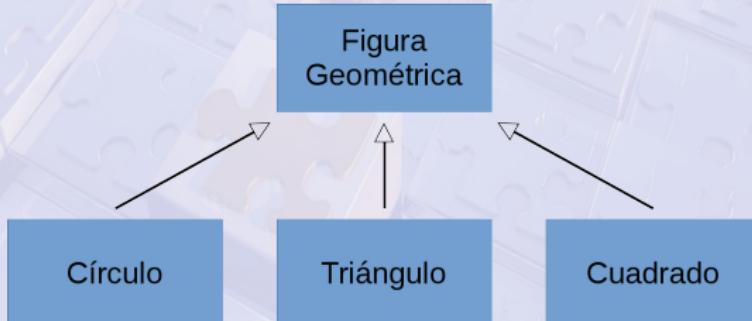
- Clases abstractas en Java
- Clases no instanciables en Ruby
- Clases abstractas en UML
- Ejemplos

## 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

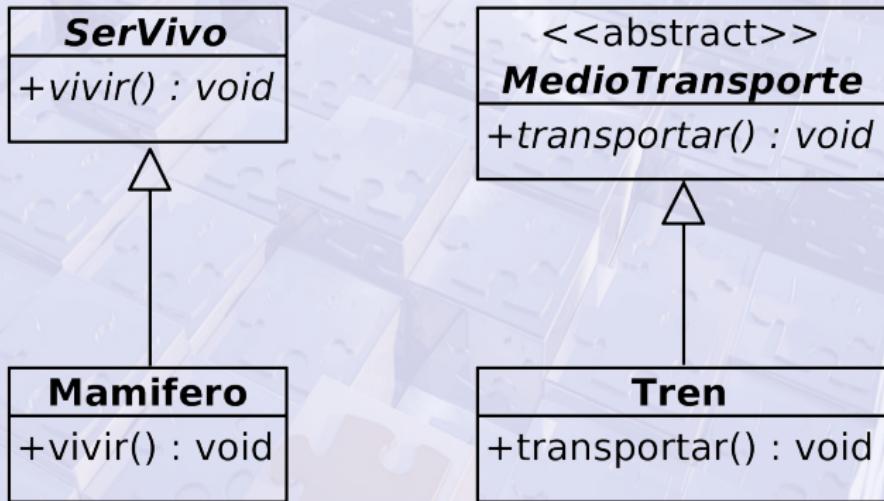
## • Solución:

- ▶ 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     # Atributos y métodos comunes
3     private_class_method :new
4 end
5
6 class Cuadrado < FiguraGeometrica
7     public_class_method :new
8     # 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";}  
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

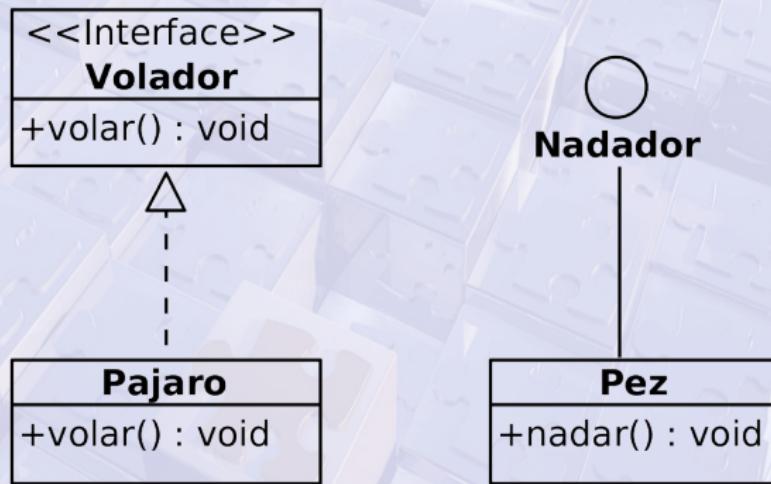
# Interfaces en Java

- 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 a otro mecanismo, polimorfismo, (que veremos más adelante) 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**

Dpto. Lenguajes y Sistemas Informáticos  
Universidad de Granada

Programación y Diseño Orientado a Objetos

(Curso 2023-2024)

## **Temporary page!**

$\text{\LaTeX}$  was unable to guess the total number of pages correctly.  
was some unprocessed data that should have been added to  
page this extra page has been added to receive it.  
If you rerun the document (without altering it) this surplus page  
away, because  $\text{\LaTeX}$  now knows how many pages to expect in  
document.

# **Clases Parametrizables**

Dpto. Lenguajes y Sistemas Informáticos  
Universidad de Granada

Programación y Diseño Orientado a Objetos

(Curso 2023-2024)

# 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
  - Algunas consideraciones de bajo nivel

# 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 (Vehiculo v) { ... } ... }
3 class ListaMascota { void insertar (Mascota m) { ... } ... }
```



# Clases parametrizables

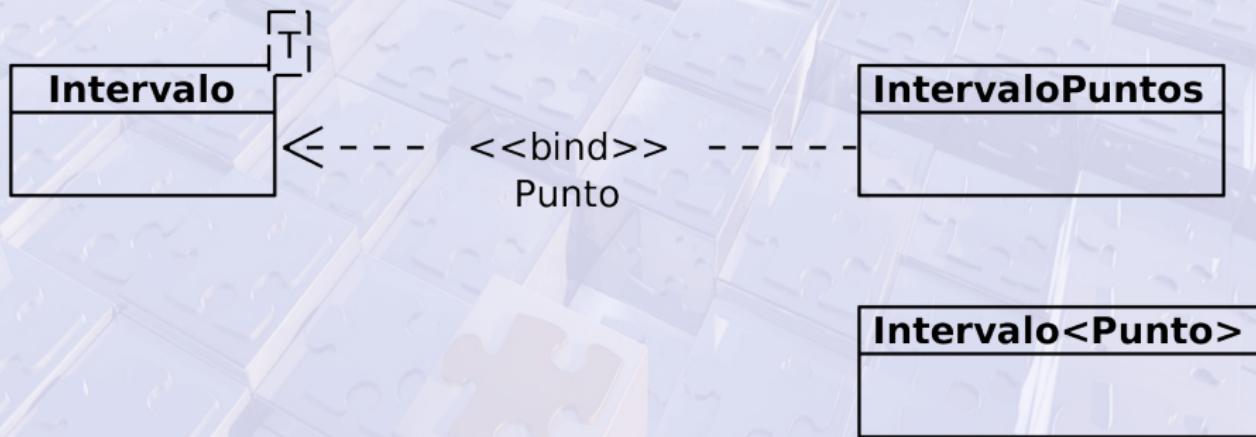
- Clases definidas en función de un tipo de dato (clase)
- Se generalizan un conjunto de operaciones válidas para diferentes tipos de datos
- El ejemplo clásico son los contenedores
  - ▶ Se puede definir una lista independientemente del tipo concreto de elementos que vaya a contener

## 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 class NodoOctal <T> {
2     private static final int NHIJOS = 8;
3     private NodoOctal <T> padre = null;
4     private final ArrayList <NodoOctal <T>> hijos = new ArrayList<>();
5     private ArrayList <T> contenido = new ArrayList<>();
6
7     NodoOctal () { padre = null; }
8     NodoOctal (NodoOctal <T> p) { padre = p; }
9
10    void divide () {
11        if (hijos.size() == 0) {
12            for (int i = 0; i < NHIJOS; i++) { hijos.add (new NodoOctal<> (this)); }
13        }
14    }
15
16    T obtenerElemento (int i) {
17        if ((i < contenido.size()) && (i >= 0)) { return contenido.get (i); }
18        else { return null; }
19    }
20
21    void insertarElemento (T e) { contenido.add (e); }
22
23    NodoOctal <T> hijo (int i) {
24        if ((i < hijos.size()) && (i >= 0)) { return hijos.get(i); }
25        else { return null; }
26    }
27 }
```

# 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 parametrizables en Java

## Algunas consideraciones de bajo nivel

- No se crea una instancia de cada clase paramétrica cada vez que se utiliza con un tipo concreto
- En tiempo de ejecución el parámetro pasa a ser `Object` o del tipo que se haya puesto como restricción
- Es un mecanismo para intentar evitar ciertos errores haciendo que sean detectables en tiempo de compilación

# Ejemplo

## Java: Clase parametrizable

```
1 class Clase1 {}  
2 class Clase2 {}  
3  
4 public class Parametrizable {  
5     public static void main(String [] args) {  
6         Clase1 c1;  
7         Clase2 c2;  
8         NodoOctal<Clase1> raiz1 = new NodoOctal<>(); // Se infiere el tipo  
9         raiz1.divide();  
10  
11        raiz1.insertarElemento (new Clase1());  
12        // raiz1.insertarElemento (new Object()); // Error compilación  
13  
14        c1 = raiz1.obtenerElemento (0);  
15        // c2 = raiz1.obtenerElemento (0); // Error compilación  
16        NodoOctal<Clase1> hijo1 = raiz1.hijo (3);  
17        hijo1.insertarElemento (new Clase1());  
18  
19        // Simulamos que no disponemos de clases parametrizables  
20        NodoOctal<Object> raiz = new NodoOctal<>();  
21        raiz.insertarElemento (new Clase1());  
22        c1 = (Clase1) raiz.obtenerElemento (0); // Cast necesario  
23        c2 = (Clase2) raiz.obtenerElemento (0); // Error ejecución  
24  
25        NodoOctal<Clase1> otro1 = new NodoOctal<> (raiz1);  
26        // NodoOctal<Clase2> otro2 = new NodoOctal<> (raiz1); // Error compilación  
27    }  
28 }
```

# Ejemplo ilamativo

Java: Ilustra la implementación de las clases parametrizables en Java

```
1 public static void main(String[] args) {  
2  
3     ArrayList<String> s = new ArrayList<>();  
4     s.add ("Sorpresa");  
5     // s.add (1); // Error de compilación  
6  
7     Object o = s;  
8     ArrayList<Integer> i = (ArrayList<Integer>) o; // Warning al compilar  
9     // Equivalente: ArrayList i=(ArrayList<Integer>) (Object) s;  
10    i.add (5);  
11    // i.add("Probando"); // Error de compilación  
12  
13    System.out.println (s); // [Sorpresa, 5]  
14  
15    for (Object c : s) {  
16        System.out.println (c.getClass ().getSimpleName () );  
17    }  
18    // String, Integer !!!  
19 }
```

# 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**

Dpto. Lenguajes y Sistemas Informáticos  
Universidad de Granada

Programación y Diseño Orientado a Objetos

(Curso 2023-2024)

## **Temporary page!**

$\text{\LaTeX}$  was unable to guess the total number of pages correctly.  
was some unprocessed data that should have been added to  
page this extra page has been added to receive it.  
If you rerun the document (without altering it) this surplus page  
away, because  $\text{\LaTeX}$  now knows how many pages to expect in  
document.

# **Polimorfismo y Ligadura Dinámica**

Dpto. Lenguajes y Sistemas Informáticos  
Universidad de Granada

Programación y Diseño Orientado a Objetos

(Curso 2023-2024)

# 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

- ¿Recordáis el ejemplo de las figuras geométricas?

## Java: Introducción a polimorfismo y ligadura dinámica

```
1 class FiguraGeometrica { // entre otras cosas ...
2     float area() { return 0.0f; }
3 }
4 class Circulo extends FiguraGeometrica { // entre otras cosas ...
5     float area() {
6         return Math.PI * radio * radio;
7     }
8 }
9 class Rectangulo extends FiguraGeometrica { // entre otras cosas ...
10    float area() {
11        return lado1*lado2;
12    }
13 }
14
15 // En algún otro sitio ...
16 ArrayList<FiguraGeometrica> colecciónDeFiguras = new ArrayList<>();
17 colecciónDeFiguras.add (new Circulo (radio));
18 colecciónDeFiguras.add (new Rectangulo (lado1, lado2));
19
20 float suma = 0.0f;
21 for (FiguraGeometrica unaFigura : colecciónDeFiguras) {
22     suma += unaFigura.area ();
23 }
```

# 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 el objeto referenciado en un momento determinado por una variable

## Java: Tipo estático y dinámico

```
1 ArrayList<FiguraGeometrica> colecciónDeFiguras = new ArrayList<>();
2 colecciónDeFiguras.add (new Circulo (radio));
3 colecciónDeFiguras.add (new Rectangulo (lado1 , lado2));
4
5 float suma = 0.0f;
6 for (FiguraGeometrica unaFigura : colecciónDeFiguras) {
7   suma += unaFigura.area ();
8 }
```

★ ¿Cuál es el tipo estático de unaFigura?

★ ¿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 FiguraGeometrica { // entre otras cosas ...
2     float area() { return 0.0f; }
3 }
4 class Circulo extends FiguraGeometrica { // entre otras cosas ...
5     float area() { return Math.PI * radio * radio; }
6 }
7 class Rectangulo extends FiguraGeometrica { // entre otras cosas ...
8     float area() { return lado1*lado2; }
9 }
10
11 // En algún otro sitio ...
12 for (FiguraGeometrica unaFigura : colecciónDeFiguras) {
13     suma += unaFigura.area();                                // ¿Qué implementación de area se va a ejecutar?
14 }
```

# 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) ((Object) new Profesor()));  
17  
18 }
```

# Comprobaciones explícitas de tipos

- Deben evitarse las comprobaciones explícitas de tipos

## Java: Ejemplo sobre comprobaciones explícitas de tipos

```
1 class Persona {  
2     private String nombre;  
3  
4     public Persona(String n) {  
5         nombre=n;  
6     }  
7  
8     public void setNombre(String n) {  
9         nombre=n;  
10    }  
11 }  
12  
13 class Profesor extends Persona {  
14  
15     public Profesor(String n) {  
16         super(n);  
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     Persona p;  
4     Random r = new Random();  
5  
6     int dado = r.nextInt(6)+1;  
7  
8     if (dado<=3) {  
9         p=new Persona("Pepe");  
10    }  
11    else {  
12        p=new Profesor("Pepe");  
13    }  
14  
15    // Nada recomendable  
16    // Mal diseño  
17    // No lo hagáis  
18    // Lo digo en serio, no hagáis este tipo de diseños  
19    if (p instanceof Profesor) {  
20        p.setNombre("Prof. Pepe");  
21    }  
22    else {  
23        p.setNombre("Don/Dña Pepe");  
24    }  
25 }
```

# Comprobaciones explícitas de tipos

- Deben evitarse las comprobaciones explícitas de tipos

## Java: Forma correcta de proceder

```
1 class Persona {  
2     private String nombre;  
3  
4     public Persona(String n) { nombre=n; }  
5  
6     protected void setNombre(String n) { nombre=n; }  
7  
8     public void cambiarNombre(String n) {  
9         setNombre("Don/Dña "+n);  
10    }  
11 }  
12 }  
13  
14 class Profesor extends Persona {  
15     public Profesor(String n) {  
16         super(n);  
17     }  
18  
19     @Override  
20     public void cambiarNombre(String n) {  
21         setNombre("Prof. "+n);  
22     }  
23 }
```

# 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     Persona p;
4     Random r=new Random();
5
6     int dado=r.nextInt(6)+1;
7
8     if (dado<=3) {
9         p=new Persona("Pepe");
10    }
11    else {
12        p=new Profesor("Pepe");
13    }
14
15 // Tenemos el comportamiento correcto de forma automática
16
17 p.cambiarNombre("Pepe");
18
19 // También será válido si se añaden nuevos descendientes
20 // de Persona/Profesor simplemente redefiniendo el método
21
22 }
```

# 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 19 a 24 de la transparencia 16 con la línea 17 de la transparencia 18  
(¡y solo hay involucradas 2 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**

Dpto. Lenguajes y Sistemas Informáticos  
Universidad de Granada

Programación y Diseño Orientado a Objetos

(Curso 2023-2024)

## **Temporary page!**

$\text{\LaTeX}$  was unable to guess the total number of pages correctly.  
was some unprocessed data that should have been added to  
page this extra page has been added to receive it.  
If you rerun the document (without altering it) this surplus page  
away, because  $\text{\LaTeX}$  now knows how many pages to expect in  
document.

# **Herencia en el Ámbito de Clase**

Dpto. Lenguajes y Sistemas Informáticos  
Universidad de Granada

**Programación y Diseño Orientado a Objetos**

**(Curso 2023-2024)**

# 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**

Dpto. Lenguajes y Sistemas Informáticos  
Universidad de Granada

**Programación y Diseño Orientado a Objetos**

**(Curso 2023-2024)**

## **Temporary page!**

$\text{\LaTeX}$  was unable to guess the total number of pages correctly.  
was some unprocessed data that should have been added to  
page this extra page has been added to receive it.  
If you rerun the document (without altering it) this surplus page  
away, because  $\text{\LaTeX}$  now knows how many pages to expect in  
document.

# **Herencia Múltiple**

Dpto. Lenguajes y Sistemas Informáticos  
Universidad de Granada

**Programación y Diseño Orientado a Objetos**

**(Curso 2023-2024)**

# Créditos

- Las siguientes imágenes e ilustraciones son libres y se han obtenido de:
  - ▶ Emojis, <https://pixabay.com/images/id-2074153/>
  - ▶  [https://medias.maisonsdumonde.com/image/upload/q\\_auto,f\\_auto/w\\_2000/img/estanteria-de-metal-negro-y-abeto-con-reloj-1000-0-31-188836\\_1.jpg](https://medias.maisonsdumonde.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

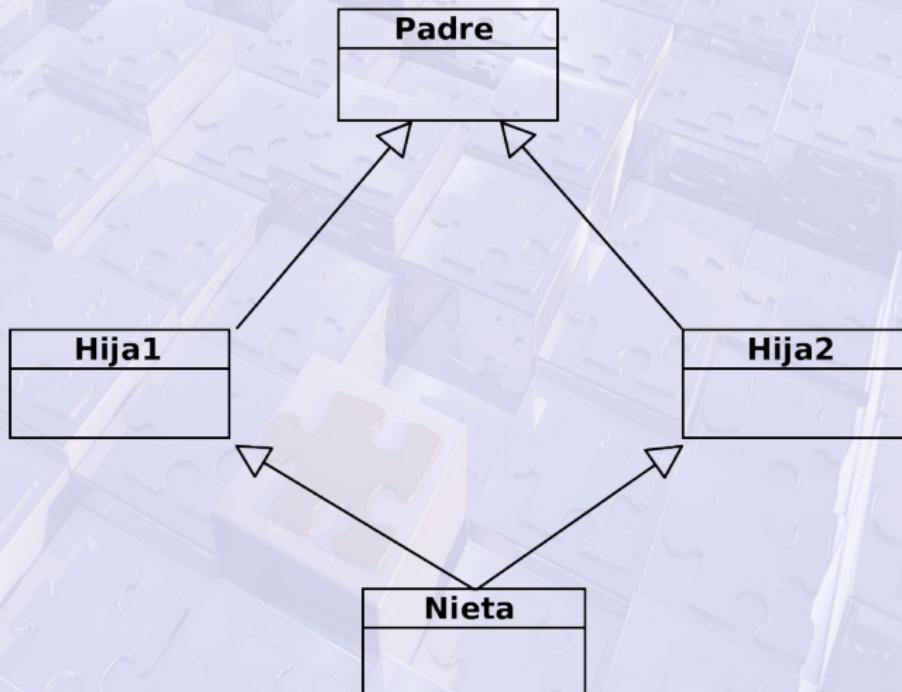
# Herencia múltiple

- 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 A {
2     private:
3         int a;
4     public:
5         A () {std::cout << "Creado A SIN INICIALIZAR" << std::endl;}
6         A (int i) {std::cout << "Creado A e inicializado" << std::endl; a = i;}
7         int getA () {return a;}
8         int setA (int i) {a = i;}
9     };
10
11 class B : public A {
12     public:
13         B (int a) : A(a) {}
14         void useAFromB () {std::cout << "Uso desde B " << getA() << std::endl;}
15     };
16
17 class C : public A {
18     public:
19         C (int a) : A(a) {}
20         void useAFromC () {std::cout << "Uso desde C " << getA() << std::endl;}
21     };
```

# Ejemplo del problema del diamante

## C++: Herencia múltiple. Problema del diamante

```
1 class D : public B, public C {
2     public:
3         D (int a) : B(a), C(a) {}
4         void useAFromD() {useAFromB(); useAFromC();}
5         // void modifyA (int a) {setA(a);} // Error: Hay ambigüedad
6         void modifyA (int a) {C::setA(a);} // evitamos que la referencia sea ambigua
7
8 };
9
10 int main(int argc,char **argv) {
11     std::cout << "Problema del diamante v0" << std::endl;
12     D *d = new D(33);
13     // Creando A e inicializando
14     // Creando A e inicializando
15     d->useAFromD ();
16     // Uso desde B 33
17     // Uso desde C 33
18
19     d->modifyA (66);
20
21     d->useAFromD ();
22     // Uso desde B 33
23     // Uso desde C 66 // Hay dos versiones del atributo A::a
24 }
```

# Ejemplo del problema del diamante

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

```
1 class A {
2     private:
3         int a;
4     public:
5         A () {std::cout << "Creado A SIN INICIALIZAR" << std::endl;}
6         A (int i) {std::cout << "Creado A e inicializado" << std::endl; a = i;}
7         int getA () {return a;}
8         int setA (int i) {a = i;}
9     };
10
11 class B : virtual public A {
12     public:
13         B (int a) : A(a) {}
14         void useAFromB () {std::cout << "Uso desde B " << getA() << std::endl;}
15     };
16
17 class C : virtual public A {
18     public:
19         C (int a) : A(a) {}
20         void useAFromC () {std::cout << "Uso desde C " << getA() << std::endl;}
21     };
```

# Ejemplo del problema del diamante

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

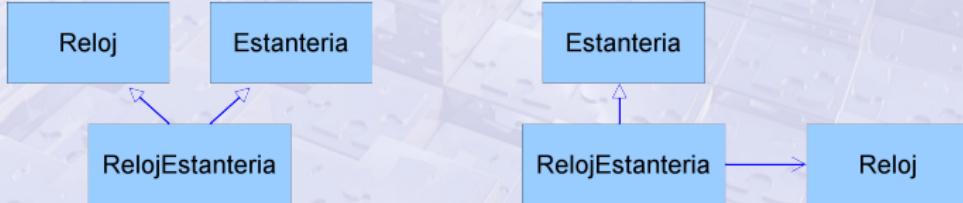
```

1 class D : public B, public C {
2     public:
3         D (int a) : A(a), B(a), C(a) {}    // Debemos llamar al constructor de A
4                                         // Si no, se llama al constructor por defecto
5         void useAFromD() {useAFromB(); useAFromC();}
6         // void modifyA (int a) {setA(a);} // Error: Hay ambigüedad
7         void modifyA (int a) {C::setA(a);} // evitamos que la referencia sea ambigua
8
9 };
10
11 int main(int argc,char **argv) {
12     std::cout << "Problema del diamante v0" << std::endl;
13     D *d = new D(33);
14     // Creando A e inicializando
15     d->useAFromD ();
16     // Uso desde B 33
17     // Uso desde C 33
18
19     d->modifyA (66);
20
21     d->useAFromD ();
22     // Uso desde B 66
23     // Uso desde C 66 // Solo hay una versión del atributo A::a
24
25     B *b = new B(77);
26     b->useAFromB();
27     // Uso desde B 77 // Las instancias de B también tienen el atributo A::a
28 }
```

# 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

# Ejemplo de composición



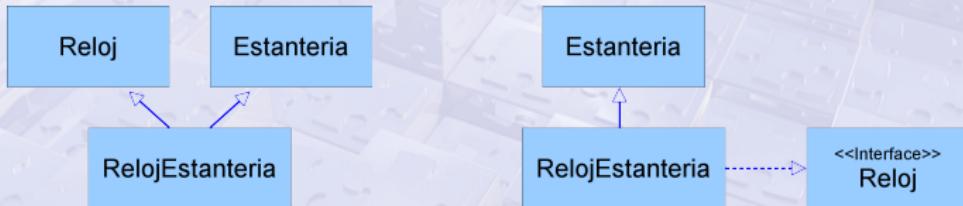
## Java: Ejemplo de composición

```

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**

Dpto. Lenguajes y Sistemas Informáticos  
Universidad de Granada

**Programación y Diseño Orientado a Objetos**

**(Curso 2023-2024)**

## **Temporary page!**

$\text{\LaTeX}$  was unable to guess the total number of pages correctly.  
was some unprocessed data that should have been added to  
page this extra page has been added to receive it.  
If you rerun the document (without altering it) this surplus page  
away, because  $\text{\LaTeX}$  now knows how many pages to expect in  
document.

# **Modelo Vista Controlador**

Dpto. Lenguajes y Sistemas Informáticos  
Universidad de Granada

Programación y Diseño Orientado a Objetos

(Curso 2023-2024)

# 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



# 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

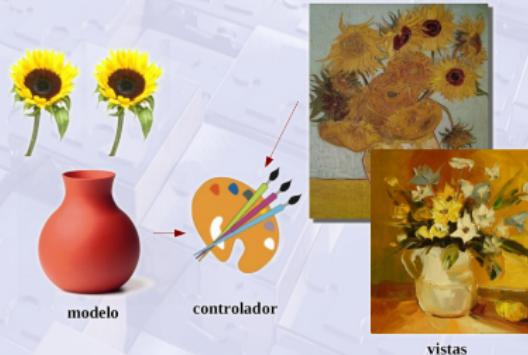
**Vista:** Una representación visual de los datos del modelo para mostrarlos al usuario

- ▶ 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

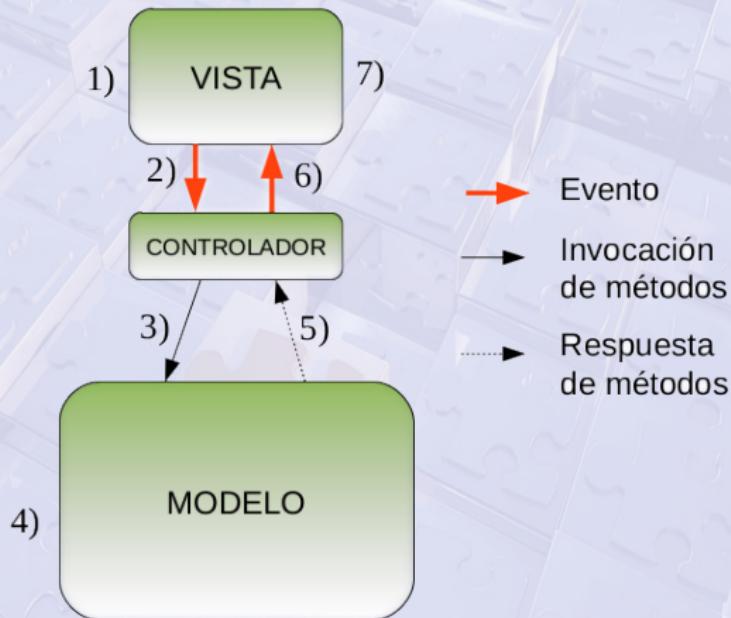


# La responsabilidad del Controlador

- 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*.

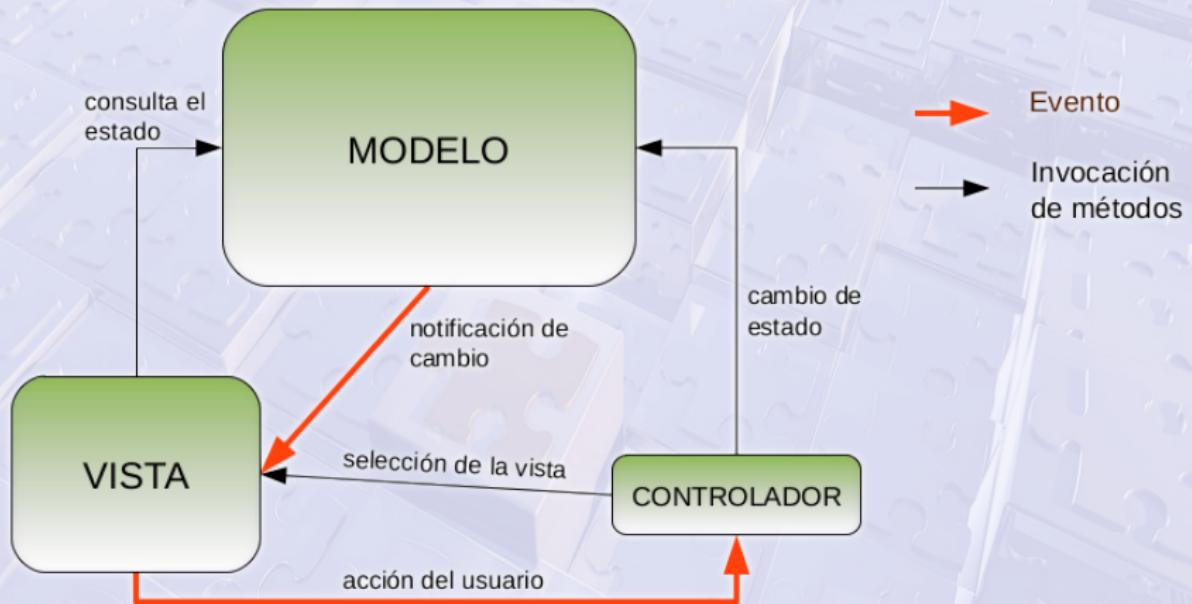
# Esquema básico de MVC

- 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**

Dpto. Lenguajes y Sistemas Informáticos  
Universidad de Granada

Programación y Diseño Orientado a Objetos

(Curso 2023-2024)

## **Temporary page!**

$\text{\LaTeX}$  was unable to guess the total number of pages correctly.  
was some unprocessed data that should have been added to  
page this extra page has been added to receive it.  
If you rerun the document (without altering it) this surplus page  
away, because  $\text{\LaTeX}$  now knows how many pages to expect in  
document.

# Copia de Objetos

Dpto. Lenguajes y Sistemas Informáticos  
Universidad de Granada

Programación y Diseño Orientado a Objetos

(Curso 2023-2024)

# 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

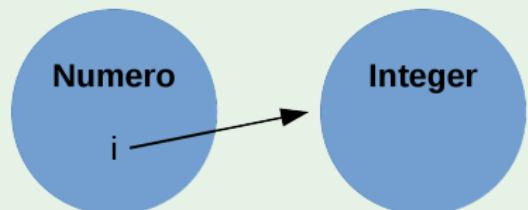
- Aparecen dos conceptos
    - ▶ Profundidad de la copia
      - ★ Hasta que nivel se van a realizar copias de estado en vez de identidad
    - ▶ Inmutabilidad de los objetos
      - ★ Un objeto es inmutable si no dispone de métodos que modifiquen su estado
  - Hay que tener cuidado con los objetos que referenciaan 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

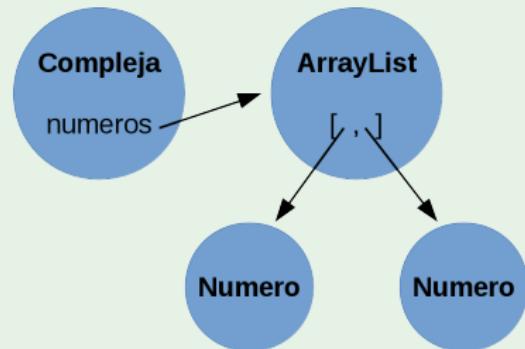
```
1 class Numero{  
2     private Integer i;  
3  
4     public Numero(Integer a) {  
5         i = a;  
6     }  
7  
8     public void inc() {  
9         // Este método modifica el estado del objeto  
10        // La clase es mutable  
11        i++;  
12    }  
13  
14    @Override  
15    public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



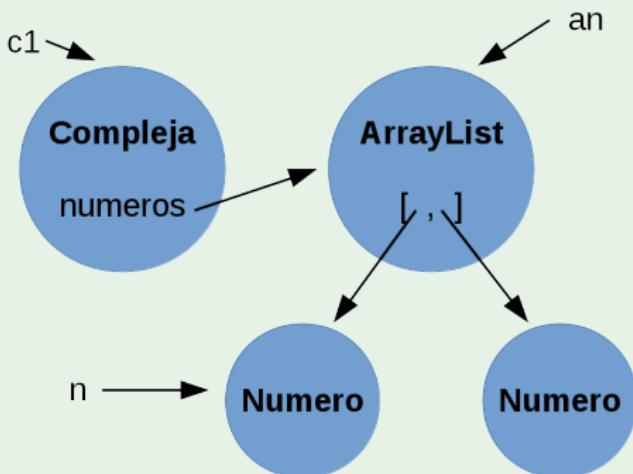
# Ejemplo

## 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 }

```



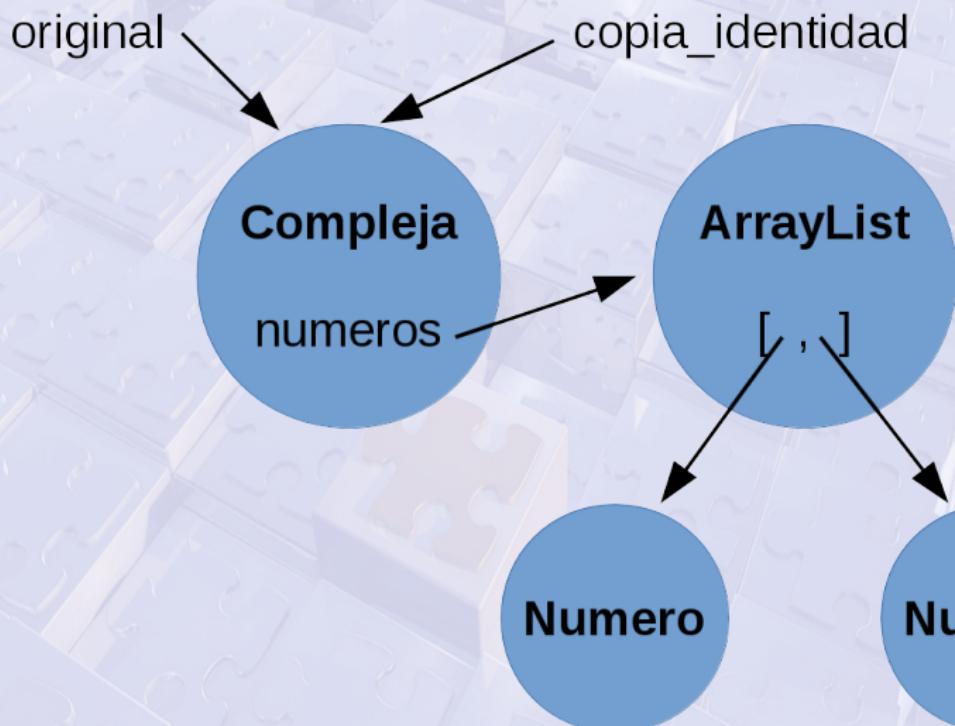
- Se expone:
  - ▶ El estado de los objetos **Compleja**
  - ▶ El de los objetos **Numero** referenciados por los objetos **Compleja**

# 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()`

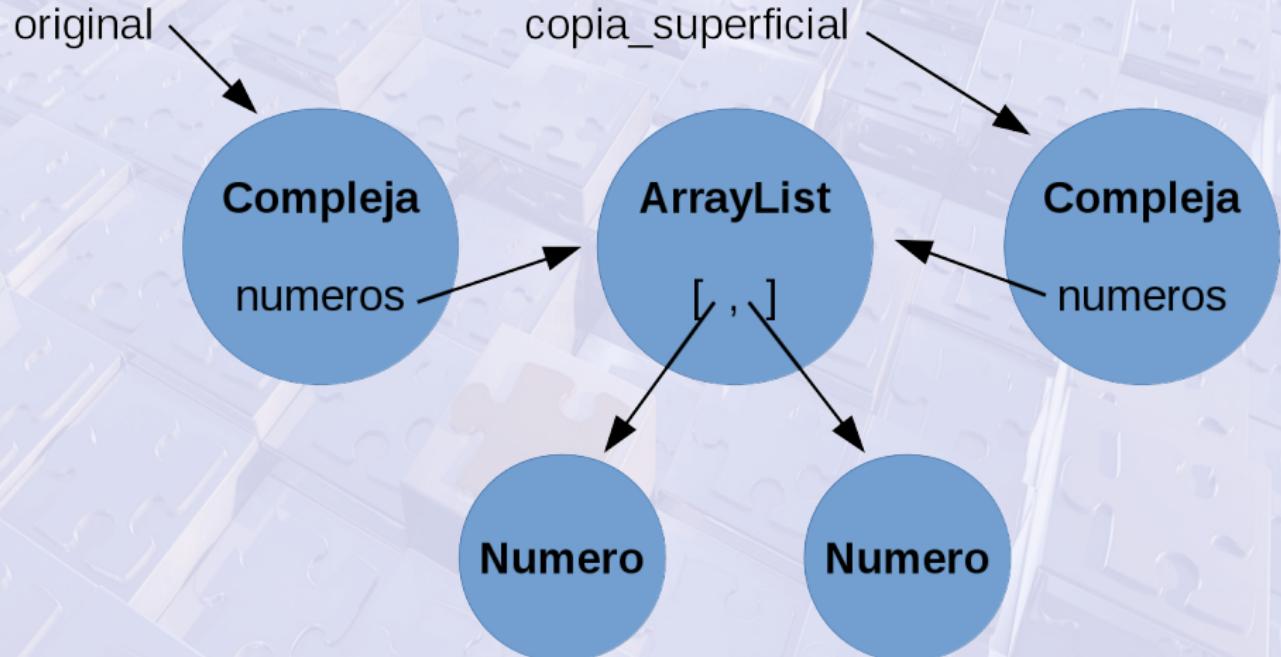
# Tipos de copia

## Copia de identidad



# Tipos de copia

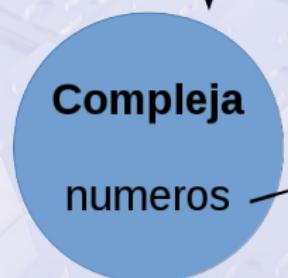
## Copia superficial



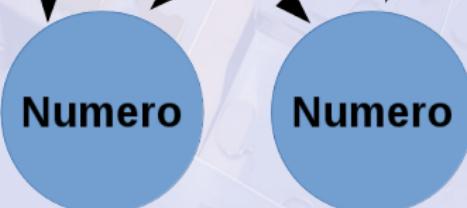
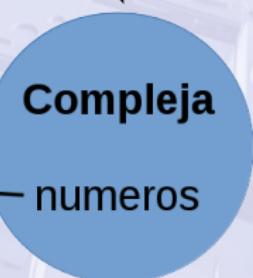
# Tipos de copia

## Copia superficial

original



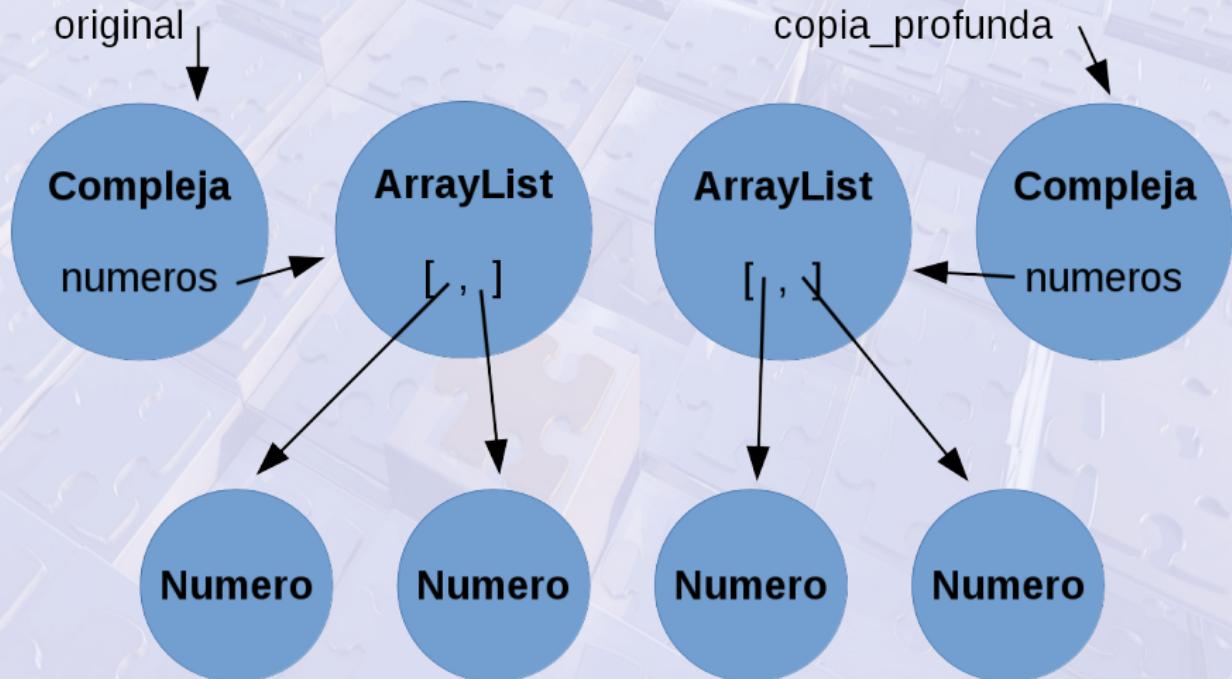
copia\_superficial\_también



# 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 }
```

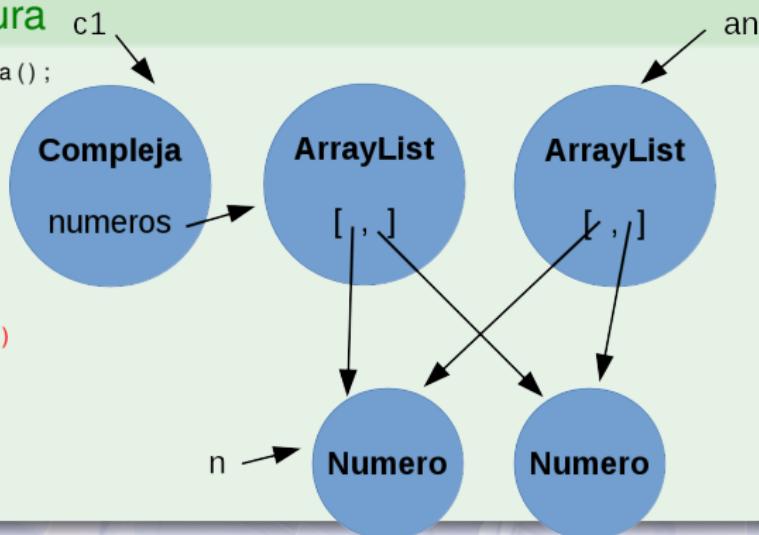
# Ejemplo

## Java: Usando ComplejaSegura

```

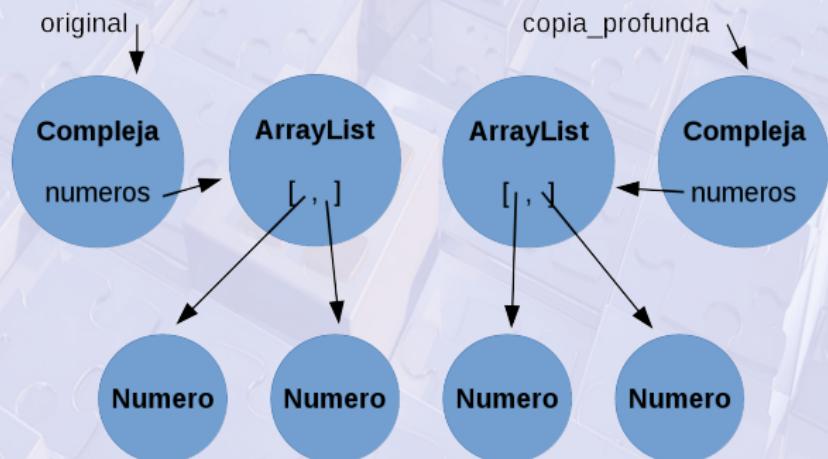
1 ComplejaSegura c1 = new ComplejaSegura();
2
3 c1.add (new Numero(3));
4 c1.add (new Numero(2));
5
6 c1.getNumeros().clear();
7 for(Numero i : c1.getNumeros()) {
8     System.out.println(i);
9 } // R --> 3 2    OK!   :-(

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         Número n = null;
6         for (Número 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 (Número 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 A {  
2     private int x;  
3     private int y;  
4  
5     public A (int a,int b) {  
6         x = a;  
7         y = b;  
8     }  
9  
10    public A (A 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 B extends A {  
2     private int z;  
3  
4     public B (int a, int b, int c) {  
5         super (a, b);  
6         z = c;  
7     }  
8  
9     public B (B 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 A a1 = new A(11, 22);
2 A a2 = new B(111, 222, 333);
3 A 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 A a4 = null;
12
13 // ¿ a3 es un A o un B ?
14
15 // a4 = new A(a3);
16 a4 = new B((B) a3);
17
18 System.out.print(a4);
```

# Ejemplo

## Java: Usamos reflexión para evitar el problema anterior

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

# 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 del supuesto de las prácticas)
    - ★ 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

Dpto. Lenguajes y Sistemas Informáticos  
Universidad de Granada

Programación y Diseño Orientado a Objetos

(Curso 2023-2024)

## **Temporary page!**

$\text{\LaTeX}$  was unable to guess the total number of pages correctly.  
was some unprocessed data that should have been added to  
page this extra page has been added to receive it.  
If you rerun the document (without altering it) this surplus page  
away, because  $\text{\LaTeX}$  now knows how many pages to expect in  
document.

# Reflexión

Dpto. Lenguajes y Sistemas Informáticos  
Universidad de Granada

Programación y Diseño Orientado a Objetos

(Curso 2023-2024)

# 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 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 def publicacion(añopublicacion)
11   @añoPublicacion = añopublicacion
12 end
13 end
14
15 libro1.publicacion(1997) # Se invoca el nuevo método
16 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 def autor(autor)
3                         @autor = autor
4                     end
5 end
6
7 libro1.autor("J. R. R. Tolkien")
8 puts libro1.inspect
9
10 libro2 = Libro.new("Cien años de soledad")
11 libro2.autor("G. García Márquez") # 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

Dpto. Lenguajes y Sistemas Informáticos  
Universidad de Granada

Programación y Diseño Orientado a Objetos

(Curso 2023-2024)

## **Temporary page!**

$\text{\LaTeX}$  was unable to guess the total number of pages correctly.  
was some unprocessed data that should have been added to  
page this extra page has been added to receive it.  
If you rerun the document (without altering it) this surplus page  
away, because  $\text{\LaTeX}$  now knows how many pages to expect in  
document.