



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**

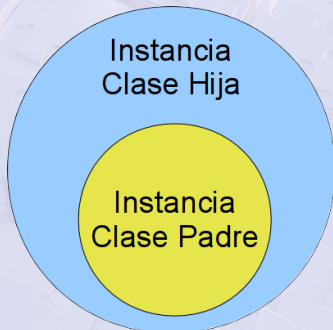
Si C hereda de B y B hereda de A, entonces C hereda de A

★ ¿Algún ejemplo de transitividad?



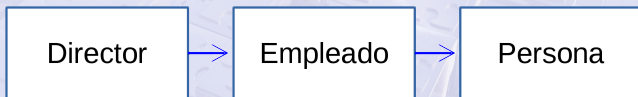
¿Qué se hereda?

- 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)

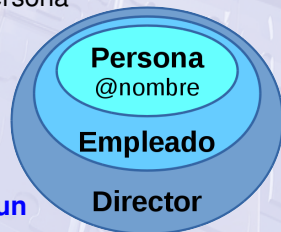
- (continuación)



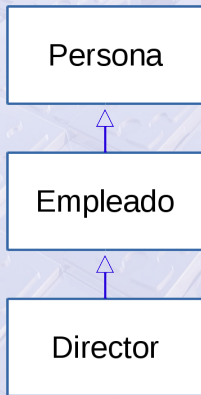
- ▶ Podría verse como que toda instancia de *Director* *tiene dentro* una instancia de *Empleado* y a su vez, toda instancia de *Empleado* *tiene dentro* una instancia de *Persona*

- 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

```

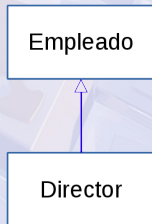
1 #encoding : utf -8
2
3 # Todas las clases están COMPLETAS
4 #   no se ha omitido ninguna línea
5
6 class Persona
7   attr_reader :nombre
8   def initialize (n)
9     @nombre = n
10  end
11 end
12 class Empleado < Persona
13 end
14 class Director < Empleado
15 end
16
17 el_dire = Director.new("Joaquín")
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   def hablar
6     "Hablo como una persona"
7   end
8 end
9 class Profesor < Persona
10  def hablar
11    "Hablo como un profesor"
12  end
13  def impartir_clase
14    "Impartiendo clase"
15  end
16 end
17 puts Persona.new.andar
18 puts Persona.new.hablar
19 puts Persona.new.impartir_clase
20 puts Profesor.new.andar
21 puts Profesor.new.hablar
22 puts Profesor.new.impartir_clase
```



★ ¿Alguna cosa os llama la atención?

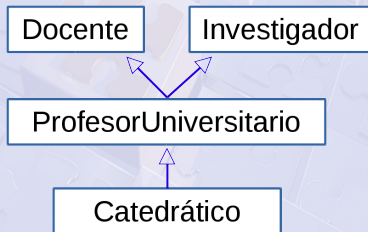
Tipos de herencia

• 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 línea
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", "PDOO")).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

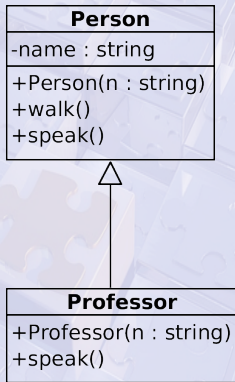
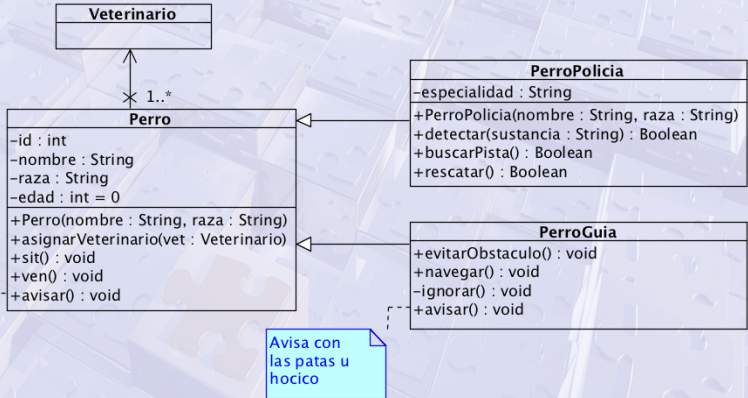


Diagrama de clases con herencia



- *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!

\LaTeX was unable to guess the total number of pages correctly. There was some unprocessed data that should have been added to the next page. This extra page has been added to receive it.

If you rerun the document (without altering it) this surplus page will be removed, because \LaTeX now knows how many pages to expect in the document.