



Ingeniería de Software

13 - Polimorfismo y Lookup

IIC2143-3

Josefa España

jpespana@uc.cl



Clases y Objetos



Clases y Objetos

Aquí mostramos cómo definir una clase y crear objetos de la misma en Ruby:

```
1 class Person
2   end
3
4   Person.new
5
6   p1 = Person.new
7   p2 = Person.new
8   p3 = Person.new
9
10  muchas_personas = []
11  10000.each do
12    muchas_personas.push(Person.new)
13  end
```



Métodos

Los métodos nos permiten definir comportamientos, por ejemplo acá se define un método greet. Para usarlo se crea un objeto usando “new”, y se llama al método de instancia.

```
1 class Person
2   def greet
3     "Hola"
4   end
5 end
6
7 p1 = Person.new
8 puts p1.greet # imprime "Hola"
9 puts p1.greet # imprime "Hola"
10
11 # no es obligatorio guardar la persona en una variable,
    puedes hacer lo siguiente:
12 puts Person.new.greet # imprime "Hola"
```



Métodos de clase

Los métodos de clase, como “backend” se ejecutan sobre la misma clase, no es necesario crear una instancia para llamar al método.

```
1 class Developer
2
3   def self.backend
4     "I am backend developer"
5   end
6
7   def frontend
8     "I am frontend developer"
9   end
10
11 end
12
13 d = Developer.new
14 d.frontend
15 Developer.backend
```



Métodos de clase

Los métodos de clase, como “backend” se ejecutan sobre la misma clase, no es necesario crear una instancia para llamar al método.

```
1 class Developer
2
3   def self.backend
4     "I am backend developer"
5   end
6
7   def frontend
8     "I am frontend developer"
9   end
10
11 end
12
13 d = Developer.new
14 d.frontend
15 Developer.backend
```



Métodos de clase

¿Para qué sirven?

Un ejemplo es la facilitación de creación de objetos:

```
1 class Person
2   def initialize(name, gender)
3     ...
4   end
5
6   def self.create_female(name)
7     Person.new(name, :female)
8   end
9
10  def self.create_male(name)
11    Person.new(name, :male)
12  end
13 end
14
15 pedro = Person.create_male("Pedro")
16 maria = Person.create_female("Maria")
```



Visibilidad de métodos

Los métodos privados solo pueden ser accedidos dentro de la misma clase, por lo que el código anterior genera un error en la última línea.

Como vimos anteriormente (clase 12), existen otros tipos de visibilidad: private, public y protected.

```
1 class Person
2   ...
3
4   private
5     def secret_method
6       puts "Este es el método secreto"
7     end
8
9     def another_secret_method
10      puts "Este es otro método secreto"
11    end
12  end
13
14  p1 = Person.new("Pedro")
15  p1.secret_method # genera un error!
```




Constructor

El “new” es un método de clase que se ejecuta sobre la clase Person. El mismo crea una instancia de la clase y posteriormente llama al método initialize sobre el objeto recién creado.

```
1 class Person
2   def initialize
3     puts "creando nueva persona ..."
4   end
5
6   ...
7 end
8
9 Person.new # imprime "creando nueva persona ..."
```



Atributos

Los atributos de Ruby empiezan con un “@”, en este ejemplo la clase Person tiene el atributo “@name”.

```
1 class Person
2   def initialize(name)
3     @name = name
4   end
5
6   def greet(other_person_name)
7     "Hola #{other_person_name}, me llamo #{@name}"
8   end
9 end
10
11 pedro = Person.new("Pedro")
12 puts pedro.greet("Juan")
13 # imprime "Hola Juan, me llamo Pedro"
```



Visibilidad de Atributos

Por defecto los atributos en Ruby son privados, es decir, solo pueden ser accedidos dentro de la misma clase. Para acceder a los atributos desde otras clases se deben hacer “accesores”.

```
1 class Person
2   def initialize(name)
3     @name = name
4   end
5   def greet(other_person_name)
6     "Hola #{other_person_name}, me llamo #{@name}"
7   end
8   # Método para que @name pueda ser leído desde afuera
9   def name
10    @name
11  end
12  # Método para que @name pueda ser modificado desde afuera
13  def name=(name)
14    @name = name
15  end
16 end
17
18 p1 = Person.new("Pedro")
19 puts p1.name # Imprime "Pedro"
20 # Supongamos que Pedro se cambio el nombre a Mary
21 p1.name="Mary"
22 puts p1.name # Imprime "Mary"
```



Visibilidad de Atributos

Es posible generar los accesores de los atributos usando el operador “attr_accessor”.

En el ejemplo “attr_reader” solo genera el método “age”, y no así el método “age=”.

```
1 class Person
2   attr_accessor :name, :gender
3   attr_reader :age # No crea el método age=
4   def initialize(name, initial_age, gender)
5     @name = name
6     @age = initial_age
7     @gender = gender
8   end
9 end
10
11 p1 = Person.new("Pedro", 30, :male)
12 p1.name= "Juan" # Cambia el nombre Juan
13 puts p1.name # Imprime Juan
14 puts p1.age # Imprime 30
15 p1.age = 40 # Genera error
```



Atributos de clase

Los atributos de clase son compartidos por todas las instancias de esta clase.

En el ejemplo, contamos cuántos objetos de una clase fueron creados.

```
1 class Person
2   @@people_count = 0
3
4   def initialize
5     @@people_count += 1
6   end
7
8   def self.people_count
9     @@people_count
10  end
11 end
12
13 puts Person.people_count # Imprime 0
14 Person.new
15 puts Person.people_count # Imprime 1
16 Person.new
17 puts Person.people_count # Imprime 2
```



Ruby (casi) todo es Objeto

Las cadenas, arreglos, hasta los enteros son objetos. Por lo mismo es posible interactuar con ellos a través de métodos.

```
1 s = String.new( "Hola" ) # => "Hola"
2 s.length # => 4
3 a = Array.new # => []
4 a.push( "Hola" ) # => ["Hola"]
5 a.push( "Mundo" ) # => ["Hola", "Mundo"]
6 a.reverse # => ["Mundo", "Hola"]
7 2.even? # => true
```



Herencia



Atributos

En el ejemplo, el objeto de la clase “Circle” tiene el atributo radius, stroke, y fill por herencia.

```
1 # Clase Padre
2 class Figure
3   attr_accessor :stroke, :fill
4 end
5 # Hereda de Figure
6 class Circle < Figure
7   attr_accessor :radius
8 end
9 # Hereda de Figure
10 class Square < Figure
11   attr_accessor :side
12   ...
13 end
14 # Hereda de Figure
15 class Triangle < Figure
16   attr_accessor :base, :height
17 end
```

```
1 c1 = Circle.new
2 c1.fill = "red"
3 puts c1.fill
```




Polimorfismo

Es la capacidad de un objeto de tomar otras formas.

En este ejemplo, cualquier case que defina un método “draw” se puede usar como argumento de “draw_figura”.

Esto en Ruby se llama duck typing:

-- *"If it walks like a duck and it quacks like a duck, then it must be a duck"*

```
def draw_figure(figure, x, y)
  set_coordinates(x, y)
  figure.draw
end

class Circle
  attr_accessor :radius

  def draw
    ...
  end
end

class Triangle
  attr_accessor :base, :height

  def draw
    ...
  end
end
```



Jerarquía de clases

Aquí hay un ejemplo de una jerarquía de clases de 3 niveles:

```
1 class Figure
2   ...
3 end
4
5 class Circle < Figure
6   attr_accessor :radius
7 end
8
9 class Cylinder < Circle
10  attr_accessor :length
11 end
```



Herencia y constructor

El método “super” permite ejecutar el método de la clase padre con el mismo nombre. Aquí la línea 9 ejecuta el método “initialize” de la clase padre.

```
1 class Parent
2   def initialize
3     puts "Este es el constructor de Parent"
4   end
5 end
6
7 class Child
8   def initialize
9     super # con esta línea ejecutamos el constructor del padre
10  end
11 end
```



Herencia y constructor

En este ejemplo se ejecuta el constructor de la clase padre mandando los atributos necesarios:

```
1 class Figure
2   attr_accessor :stroke, :fill
3
4   def initialize(stroke, fill)
5     @stroke = stroke
6     @fill = fill
7   end
8 end
9
10 class Circle < Figure
11   attr_accessor :radius
12
13   def initialize(stroke, fill, radius)
14     super(stroke, fill)
15     @radius = radius
16   end
17 end
```



Sobre-escritura de métodos

El método “to_s” puede ser redefinido en la clase Circle.

La sobre-escritura es escribir un método en la clase hija que tiene la misma firma (nombre y argumentos) que un método existente en la clase padre.

En Ruby, todas las clases por defecto heredan de Object, y “to_s” está definido en Object.

```
1 class Circle
2   def initialize(r)
3     @radius = r
4   end
5   def to_s
6     "Este es un círculo con radio #{@radius}"
7   end
8 end
9
10 c1 = Circle.new(5)
11 puts c1.to_s
12 # Imprime "Este es un círculo con radio 5"
```



Super y sobre-escritura

Con el keyword “super” se puede llamar al método de la clase padre que tiene el mismo nombre:

```
1 class Employee
2   def calculate_salary
3     # código complejo para calcular el salario
4   end
5 end
6
7 class Manager < Employee
8   def calculate_salary
9     base_salary = super
10    base_salary + @bonus
11  end
12 end
```



Clases abstractas

Una clase abstract es una clase incompleta, es decir, que falta la implementación de uno o más métodos.

Las clases hijas de la clase abstracta tienen la responsabilidad de implementar dicho método, de lo contrario Ruby lanzará un error.

```
1 class Figure
2   def print
3     raise NotImplementedError
4   end
5 end
6
7 class Square < Figure
8
9 end
10 f = Figure.new
11 f.print
12 # lanza error porque la clase Square
    no implementa el método print
```



Método Lookup



¿Qué método se ejecutará?

Para saber qué método se ejecutará usaremos un algoritmo básico de búsqueda, típico en la mayoría de los lenguajes de programación:

- Primero busca el método M en la lista de métodos de instancia dentro de la clase del objeto que recibe el mensaje.
- Si no lo encuentra, busca el método M en la clase padre recursivamente.
- Si luego de buscar en toda la jerarquía de clases el método no es encontrado, se invoca al método “method_missing”, el mismo que lanzó un error.

Ruby tiene algunos pasos adicionales, por ejemplo, cuando se utilizan módulos. Sin embargo, durante el curso, usaremos el algoritmo anterior por simplicidad.



Diferencia entre self y super

- **self:** Busca el método desde la clase del objeto que recibe el mensaje.
- **super:** Busca el método desde la clase padre de donde se encuentra la llamada a “super”.



¿Qué imprime el siguiente código?

```
1 class A
2   def foo
3     1
4   end
5   def bar
6     3
7   end
8 end
9
10 class B < A
11   def bar
12     foo + super
13   end
14 end
15
16 puts B.new.bar
```



¿Qué imprime el siguiente código?

```
1 class A
2   def foo
3     1
4   end
5   def bar
6     3
7   end
8 end
9
10 class B < A
11   def bar
12     foo + super
13   end
14 end
15
16 puts B.new.bar
```

Resultado: 4

Explicación:

- Primero se crea una instancia de B, y se llama al método bar.
- El método bar llama a foo, que al no estar definido en B, usa la definición de A, es decir, 1.
- Luego se suma “super”, es decir, la definición de “bar” de la herencia:
- Por lo tanto nos queda 1 + 3.



¿Qué imprime el siguiente código?

```
1 class S
2   def foo
3     self.bar
4     puts "S>>foo"
5   end
6   def bar
7     puts "S>>bar"
8   end
9 end
10
11 class A < S
12   def foo
13     super
14     puts "A>>foo"
15   end
16 end
```

```
18 class B < S
19   def bar
20     puts "B>>bar"
21   end
22 end
23
24 class C < S
25   def foo
26     puts "C>>foo"
27   end
28 end
29
30 S.new.foo
31 A.new.foo
32 B.new.foo
33 C.new.foo
```



¿Qué imprime el siguiente código?

```
1 class S
2   def foo
3     self.bar
4     puts "S>>foo"
5   end
6   def bar
7     puts "S>>bar"
8   end
9 end
10
11 class A < S
12   def foo
13     super
14     puts "A>>foo"
15   end
16 end
```

```
18 class B < S
19   def bar
20     puts "B>>bar"
21   end
22 end
23
24 class C < S
25   def foo
26     puts "C>>foo"
27   end
28 end
29
30 S.new.foo
31 A.new.foo
32 B.new.foo
33 C.new.foo
```

S.new.foo

- S>>bar
- S>>foo

A.new.foo

- S>>bar
- S>>foo
- A>>foo

B.new.foo

- B>>bar
- S>>foo

C.new.foo

- C>>foo



¿Consultas?



Ingeniería de Software

13 - Polimorfismo y Lookup

IIC2143-3

Josefa España

jpespana@uc.cl