Object Oriented Programming

Juan Pablo Sandoval

Clases y Objetos

Juan Pablo Sandoval

Classes and Objects

```ruby
class Person
end

Person.new

p1 = Person.new
p2 = Person.new
p3 = Person.new

muchas_personas = []
10000.each do
  muchas_personas.push(Person.new)
end
```

El ejemplo anterior muestra cómo definir una clase y crear objetos de la misma en Ruby.

Comportamiento (métodos)

```ruby
class Person
  def greet
    "Hola"
  end
end

p1 = Person.new
puts p1.greet # imprime "Hola"
puts p1.greet # imprime "Hola"

# no es obligatorio guardar la persona en una variable, puedes hacer lo siguiente:
puts Person.new.greet # imprime "Hola"
```

El ejemplo define un método “greet”, crea un objeto utilizando “new” y llama a dicho método.
A este tipo de métodos se les conoce como ***métodos de instancia*** *ya que se ejecutan sobre una instancia de la clase.*

Métodos de clase

```ruby
class Developer

  def self.backend
    "I am backend developer"
  end

  def frontend
    "I am frontend developer"
  end

end

d = Developer.new
d.frontend
Developer.backend
```

El método backend es un método de clase, ya que este se ejecuta sobre la clase misma. No es necesario crear una instancia.

Para qué sirven los métodos de clase?

```ruby
class Person
  def initialize(name, gender)
    ...
  end

  def self.create_female(name)
    Person.new(name, :female)
  end

  def self.create_male(name)
    Person.new(name, :male)
  end
end

pedro = Person.create_male("Pedro")
maria = Person.create_female("Maria")
```

El código anterior crea dos métodos de clase que “facilitan” la creación de objetos.

Visibilidad de los métodos

```ruby
class Person
  ...

  private
    def secret_method
      puts "Este es el método secreto"
    end

    def another_secret_method
      puts "Este es otro método secreto"
    end
end

p1 = Person.new("Pedro")
p1.secret_method # genera un error!
```

Los métodos privados solo pueden ser accedido dentro de la misma clase. Por lo que el código anterior genera un error en la ultima linea. Existen otros tipos de visibilidad: private, public y protected.

El constructor

```ruby
class Person
    def initialize
    puts "creando nueva persona ..."
  end

  ...
end

Person.new # imprime "creando nueva persona ..."
```

Note que 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.

Attributos

```ruby
class Person
    def initialize(name)
    @name = name
  end

  def greet(other_person_name)
    "Hola #{other_person_name}, me llamo #{@name}"
  end
end

pedro = Person.new("Pedro")
puts pedro.greet("Juan")
# imprime "Hola Juan, me llamo Pedro"
```

Los atributos en ruby empiezan con una “@”, en este ejemplo la clase persona tiene el atributo “@name”.

Visibilidad de los Atributos

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

```ruby
class Person
    def initialize(name)
    @name = name
  end
  def greet(other_person_name)
    "Hola #{other_person_name}, me llamo #{@name}"
  end
  # Método para que @name pueda ser leído desde afuera
  def name
    @name
  end
  # Método para que @name pueda ser modificado desde afuera
  def name=(name)
    @name = name
  end
end

p1 = Person.new("Pedro")
puts p1.name # Imprime "Pedro"
# Supongamos que Pedro se cambio el nombre a Mary
p1.name=(“Mary”)
puts p1.name # Imprime "Mary"
```

Visibilidad de los Atributos

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

En este ejemplo “attr_reader” solo genera el método “age” y no asi el método “age=”.

```ruby
class Person
  attr_accessor :name, :gender
  attr_reader :age # No crea el método age=
  def initialize(name, initial_age, gender)
    @name = name
    @age = initial_age
    @gender = gender
  end
end

p1 = Person.new("Pedro", 30, :male)
p1.name= "Juan" # Cambia el nombre Juan
puts p1.name # Imprime Juan
puts p1.age # Imprime 30
p1.age = 40 # Genera error
```

Atributos de clase

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

El ejemplo permite contar cuántos objetos de una clase fueron creados.

```ruby
class Person
  @@people_count = 0

  def initialize
    @@people_count += 1
  end

  def self.people_count
    @@people_count
  end
end

puts Person.people_count # Imprime 0
Person.new
puts Person.people_count # Imprime 1
Person.new
puts Person.people_count # Imprime 2
```

En ruby (casi) todo es objeto

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

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

Herencia

Juan Pablo Sandoval

Attributos

```ruby
# Clase Padre
class Figure
  attr_accessor: :stroke, :fill
end
# Hereda de Figure
class Circle < Figure
  attr_accessor :radius
end
# Hereda de Figure
class Square < Figure
  attr_accessor :side
  ...
end
# Hereda de Figure
class Triangle < Figure
  attr_accessor :base, :height
end
```

```ruby
c1 = Circle.new
c1.fill = "red"
puts c1.fill
```

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

Jerarquía de clases

```ruby
class Figure
  ...
end

class Circle < Figure
  attr_accessor :radius
end

class Cylinder < Circle
  attr_accessor :length
end
```

Este ejemplo muestra una jerarquía de clases de 3 niveles.

Herencia y el constructor

```
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
```

Super permite ejecutar el método de la clase padre con el mismo nombre. En este ejemplo la linea 9 ejecuta el método “initialize” de la clase padre.

Herencia y el constructor

```ruby
class Figure
  attr_accessor: :stroke, :fill

  def initialize(stroke, fill)
    @stroke = stroke
    @fill = fill
  end
end

class Circle < Figure
  attr_accessor :radius

  def initialize(stroke, fill, radius)
    super(stroke, fill)
    @radius = radius
  end
end
```

En este ejemplo se ejecuta el constructor de la clase padre mandándole los atributos necesarios.

Sobre-escritura de métodos

```ruby
class Circle
  def initialize(r)
    @radious = r
  end
  def to_s
    "Este es un círculo con radio #{@radius}"
  end
end

c1 = Circle.new(5)
puts c1.to_s
# Imprime "Este es un círculo con radio 5"
```

En este ejemplo el método “to_s” es re-definido 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 heredan de Object, y el método “to_s” esta definido en la clase Object.

super y sobre-escritura

```ruby
class Employee
  def calculate_salary
    # código complejo para calular el salario
  end
end

class Manager < Employee
  def calculate_salary
    base_salary = super
    base_salary + @bonus
  end
end
```

Con el keyword super se puede llamar al método de la clase padre con el mismo nombre.

Clases abstractas

```ruby
class Figure
  def print
    raise NotImplementedError
  end
end

class Square < Figure

end
f = Figure.new
f.print
# lanza error porque la clase Square no implementa el método print
```

Una clase abstract es una clase incompleta, es decir que le falta la implementación de uno o más métodos. Las clases hijas de la clase abstracta tienen que implementar dicho método. De lo contrarío Ruby lanzara error.

El método Lookup

Juan Pablo Sandoval

Como saber que método se ejecutara

Algoritmo básico de búsqueda 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 utiliza módulos. Sin embargo, durante el curso, utilizaremos el algoritmo anterior.

diferencia entre *self* y *super*

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

Ejemplo 1: Que imprime el siguiente código?

```ruby
class A
  def foo
    1
  end
  def bar
    3
  end
end

class B < A
  def bar
    foo + super
  end
end

puts B.new.bar
```

Ejemplo 2: Que imprime el siguiente código?

```ruby
class S
  def foo
    self.bar
    puts "S>>foo"
  end
  def bar
    puts "S>>bar"
  end
end

class A < S
  def foo
    super
    puts "A>>foo"
  end
end
```

```ruby
class B < S
  def bar
    puts "B>>bar"
  end
end

class C < S
  def foo
    puts "C>>foo"
  end
end

S.new.foo
A.new.foo
B.new.foo
C.new.foo
```

Ejercicio en clase

- *Ingrese a la sección de evaluaciones en canvas y resuelva los ejercicios de "POO - Quick Test".*
- *Una vez iniciado solo tiene 10 minutos.*
- *Una vez todos hayan finalizado, el profesor resolverá los ejercicios en pizarra.*
- *Se una decima extra para la I2 a los que tengan >=3 respuestas buenas.*