IIC 2143 – Ingeniería de Software

Polimorfismo y lookup

M. Trinidad Vargas
mtvargas1@uc.cl

Clases y objetos

Definimos una clase y creamos un objeto de esta misma clase

```ruby
class Person
end

p1 = Person.new
```

Métodos

Los métodos nos permiten definir comportamiento.

Necesitamos crear el objeto con new y llamamos al método de instancia

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

p1 = Person.new
puts p1.greet
puts Person.new.greet
```

Métodos de clase

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

```
class Developer
 def self.backend
   "I am backend developer"
 end

 def self.frontend
   "I am frontend developer"
 end
end

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

Métodos de clase

¿Para qué sirven?

Un ejemplo es la facilitación de creación de objetos

```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
```

```ruby
pedro = Person.create_male
maria = Person.create_female
```

Visibilidad de métodos

Los métodos privados solo pueden ser accedidos dentro del contexto del mismo objeto.

Los métodos protegidos pueden ser accedidos dentro de la misma clase o subclase.

```ruby
class Person
 private
 def secret_method
   puts "Este es el método secreto"
 end
end

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

Constructor

new es un método de clase que se usa para crear una nueva instancia

Ruby crea el objeto y luego llama al método initialize para configurarlo

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

Atributos

Los atributos empiezan con @

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

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

pedro = Person.new("Pedro")
puts pedro.greet("Maria")
```

Visibilidad de atributos

Por defecto, los atributos en Ruby son privados, es decir, solo pueden ser accedidos dentro de una misma instancia.

Para acceder desde otras clases o instancias es necesario definir accesores

attr_reader genera atributo

attr_writter genera atributo=(atributo)

attr_accessor genera ambos

Visibilidad de atributos

Por defecto, los atributos en Ruby son privados, es decir, solo pueden ser accedidos dentro de una misma instancia.

Para acceder desde otras clases o instancias es necesario definir accesores

```ruby
class Person
 def name
   @name
 end

 def name=(name)
   @name = name
 end
end
```

```ruby
class Person
 attr_accessor :name, :gender
 attr_reader :age

 def initialize(name, initial_age, gender)
   @name = name
   @age = initial_age
   @gender = gender
 end
end
```

Atributos de clase

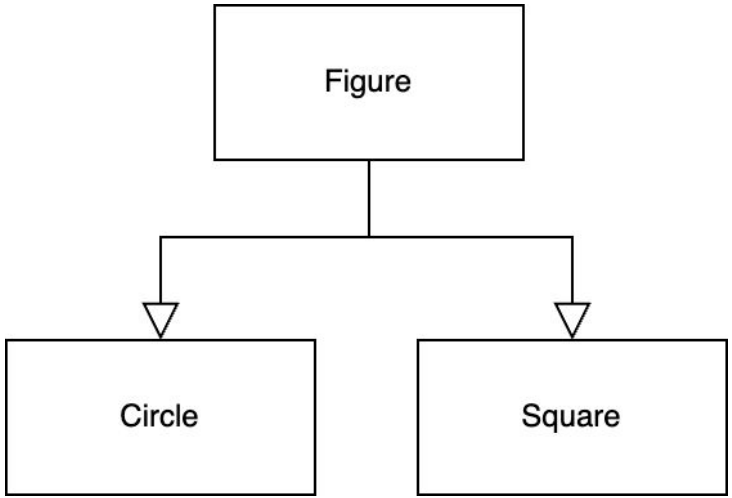
Los atributos de clase son compartidos por todas las instancias de esta clase

```
class Person
 @@people_count = 0

 def initialize
   @@people_count += 1
 end

 def self.people_count
   @@people_count
 end
end

Person.new
puts Person.people_count
```

En Ruby (casi) todo es objeto

Los strings, arrays y hasta los enteros son objetos por lo que es posible interactuar con ellos a través de sus métodos

Herencia



Atributos

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

class Circle < Figure
 attr_accessor :radius
end

c1 = circle.new
c1.fill = "red"
puts c1.fill
```

Polimorfismo

Es la capacidad de un objeto de tomar otras formas.

En lenguajes dinámicos (Ruby, Python) se conoce como duck typing

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

Polimorfismo

```ruby
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
```

```ruby
c1 = Circle.new

draw_figure(c1)
```

* Polimorfismo y herencia están relacionado pero pueden existir por si mismos

Jerarquía de clases

En este ejemplo hay una jerarquía de clase de tres niveles

```
class Figure
end

class Circle < Figure
 attr_accessor :radius
end

class Cylinder < Circle
 attr_accessor :length
end
```

Herencia y constructor

El método **`super`** permite acceder al método con el mismo nombre de la clase padre

```
class Parent
 def initialize
   puts "Este es el constructor de Parent"
 end
end

class Child < Parent
 def initialize
   super
 end
end
```

Sobre-escritura de métodos

Los métodos pueden ser redefinidos en las clases hijas teniendo la misma firma (nombre y argumentos) que un método existente de la clase padre

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

```
class Circle < Figure
 def to_s
   "Este circulo tiene radio #{@radius}"
 end
end
```

Sobre-escritura de métodos

```
class Figure
 def initialize(stoke, fill)
   @stroke = stroke
   @fill = fill
 end
end

class Circle < Figure
 attr_accessor :radius

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

Clases abstractas

Una clase abstracta es una clase incompleta donde falta la implementación de uno o más métodos.

Las clases hijas tienen la responsabilidad de implementar los métodos faltantes, de lo contrario Ruby lanzará un error

Clases abstractas

```
class Figure
 def print
   raise NotImplementedError
 end
end

class Square < Figure
end

f = Figure.new
f.print # lanza error
```

¿Cómo se relacionan la herencia y la generalización?

Búsqueda de métodos (Method lookup)

¿Qué método se ejecuta?

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?

```ruby
class A
 def foo
   1
 end

 def bar
   3
 end
end
```

```ruby
class B < A
 def bar
   foo + super
 end
end

puts B.new.bar
```

¿Qué imprime el siguiente código?

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

 def bar
   puts "S>>bar"
 end
end
```

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

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

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

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

Herencia y especialización

- La herencia busca reutilizar código
- La especialización busca modificar el comportamiento base
- La especialización utiliza herencia
- En Rails podemos tener herencia usando el mismo modelo o diferentes, es muy importante distinguir cuándo usar cada uno

Posibles errores

- Sobre utilizar herencia
  - Que se pueda usar no significa que sea la mejor solución

- Usar herencia cuando se debería usar composición
  - Diferenciar es-un de tiene-un

Ejercicio

Estás creando una aplicación que gestiona repartos a domicilio. En una primera instancia tienes repartidores con Auto y con Bicicleta que entregan los distintos pedidos.

Todos los repartidores pueden aceptar pedidos pero cada tipo tiene una capacidad y tiempo de reparto distinto

¿Cómo abordar el problema?

Preguntas claves

- ¿Cuáles son los actores del sistema? -> posibles clases
- ¿Qué hace cada uno? ¿Cuáles son sus responsabilidades? -> posibles métodos
- ¿Qué relaciones lógicas existen entre ellos? -> Herencia, composición, polimorfismo etc.

Preguntas

- ¿Qué tan fácil sería agregar un nuevo tipo de repartidor? Por ejemplo, un repartidor a pie
- ¿Qué cambiarías si tuvieras que permitir múltiples vehículos por repartidor?