POO, Módulos, Mixins y Rack _

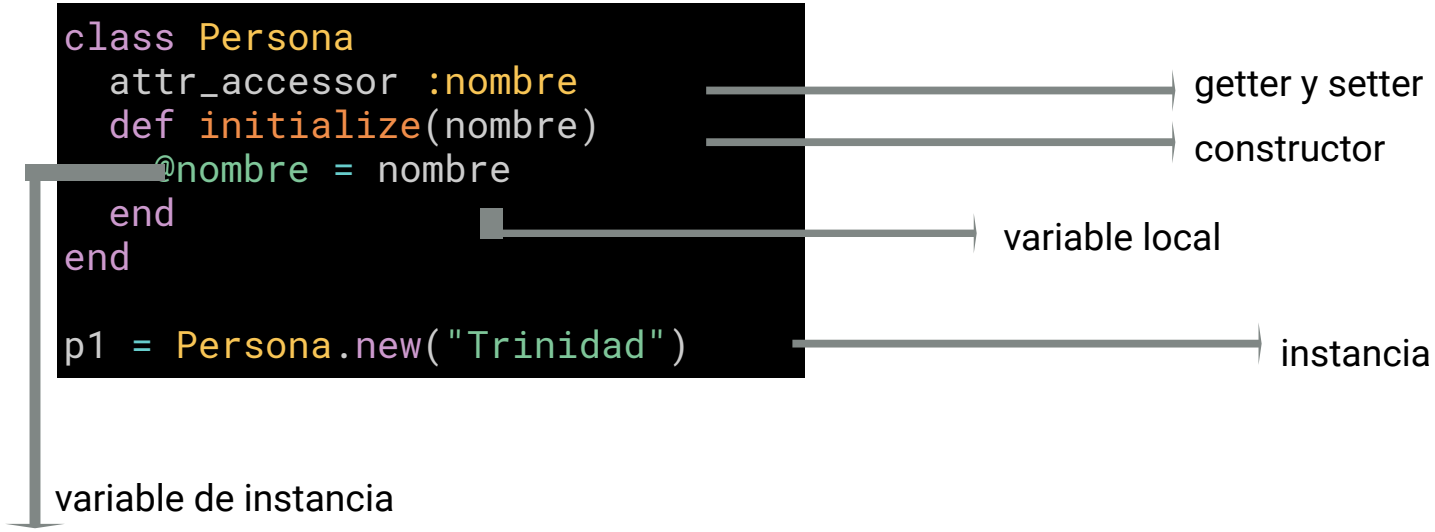
Repaso

- Los objetos son parte importante de Ruby
- Casi todo es un objeto.
- Los objetos nos ayudan a ordenar y mantener nuestro código.
- Los objetos también suelen llamarse instancias.
- Para crear un objeto necesitamos instanciar una clase

Requisitos para esta unidad

- clase
- instancia
- constructor (initialize)
- getter
- setter
- attr_accessor

Repaso


class Persona
attr_accessor :nombre
def initialize(nombre)
@nombre = nombre
end
end
p1 = Persona.new("Trinidad")
getter y setter
constructor
variable local
instancia
variable de instancia

¿Qué aprenderemos?

- Mutabilidad y como crear métodos inmutables.
- Atributos y métodos de clase.
- Reutilización de código con herencia.
- Reutilización de código con módulos.

Conceptos claves de esta clase

- Métodos de clase
- Variables de clase
- self
- Herencia
- super
- Módulos
- Mixins

Identidad y mutabilidad

Identidad

Aquí aprenderemos a distinguir cuándo dos objetos son iguales

Identidad

```ruby
class Persona
  attr_accessor :nombre
  def initialize(nombre)
    @nombre = nombre
  end
end

p1 = Persona.new("Trinidad")
p2 = Persona.new("Trinidad")
```

¿p1 == p2?

¿Cuántos kilómetros ha caminado la persona2?

```ruby
class Persona
  def initialize(nombre, caminado = 0)
    @nombre = nombre
    @caminado = caminado
  end

  def caminar(km = 1)
    @caminado += km
  end

  def caminado
    @caminado
  end
end

p1 = Persona.new("Javiera")
p1.caminar(5)
p1.caminar

p2 = Persona.new("Javiera")
p2.caminar(10)

puts p1.caminado
puts p2.caminado
```

Identidad

Toda instancia tiene un identificador único, podemos saber cual es preguntando por object_id.

```
a = [4,3,2,1]

a.object_id #70326026069520
```

2 instancias distintas tienen identificadores distintos

```
a = [4,3,2,1]

b = [4,3,2,1]

puts a.object_id

puts b.object_id
```

Si dos variables almacenan la misma instancia entonces ambas nos mostrarán el mismo object_id

```
a = [4,3,2,1]

b = a

puts a.object_id

puts b.object_id

puts a.object_id == b.object_id #false
```

Identidad

Toda instancia tiene un identificador único, podemos saber cual es preguntando por object_id.

```
a = [4,3,2,1]

a.object_id #70326026069520
```

Misma instancia mismo object_id

```
a = [4,3,2,1]

b = a

puts a.object_id

puts b.object_id

puts a.object_id == b.object_id
```

Métodos de clase

En ruby las clases también pueden tener atributos y métodos, aunque su uso es distinto.

Una clase con método de instancia y de clase

```ruby
class MiClase
    def de_instancia
        puts "Soy un método de instancia"
    end

    def self.de_clase
        puts "Soy un método de clase"
    end
end

MiClase.new.de_instancia
MiClase.de_clase
```

¿Por qué tenemos que aprender esta diferencia?

- Porque tienen usos distintos.
- Además cuando trabajemos con modelos en rails veremos que tienen ambos tipos de métodos y aprender bien la diferencia nos ayudará a disminuir la curva de aprendizaje.

Clase vs objetos

Tanto la clase como objeto pueden tener comportamientos y nosotros podemos agregarles cuantos queramos.

Clase :

Se les llama métodos de clase

Es un método que se encuentra disponible para la clase

Objeto:

Se les llamas métodos de instancia

Es un método específico para cada instancia de la clase.”

Cuando definimos métodos de clase los debemos ocupar desde la clase, cuando definimos métodos para las instancias los tenemos que ocupar desde las instancias.

Métodos de clase

```ruby
class Alumno
  def initialize()
      @notas = []
      nombre = "Humberto"
  end

  def self.cantidad_de_alumnos
     10
  end
end

Alumno.cantidad_de_alumnos
```

Los métodos de clase empiezan con self.

Los métodos de clase se aplican sobre la clase

Recordemos un ejemplo de clase anterior

```ruby
class Movie
    attr_accessor :name, :date, :studio, :category, :votes
    def initialize(name, date, studio, category, votes)
        @name = name
        @date = date
        @studio = studio
        @category = category
        @votes = votes
    end
end
```

```ruby
movies = []
file = File.open("movies.txt")
data = file.readlines()
file.close

data.each_slice(5) do |movie_data|
    movies << Movie.new(*movie_data)
end

movies.group_by(&:studio).each do |studio, group|
    puts studio
    puts group.count
end
```

Con un método de clase podríamos reordenar nuestro código

```ruby
class Movie
    attr_accessor :name, :date, :studio, :category, :votes
    def initialize(name, date, studio, category, votes)
        @name = name
        @date = date
        @studio = studio
        @category = category
        @votes = votes
    end

    def self.load_data(file)
        movies = []
        file = File.open(file)
        data = file.readlines()
        file.close

        data.each_slice(5) do |movie_data|
            movies << Movie.new(*movie_data)
        end

        movies
    end
end
```

```ruby
movies = Movie.load_data("movies.txt")
puts movies
```

De esta forma nuestro código es más fácil de reutilizar y mantener

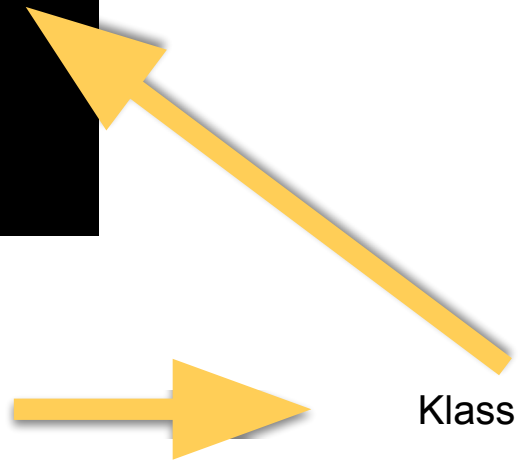
Otra opción sin métodos de clase pudo haber sido crear otra clase, que contenga un arreglo de movies

Determinar cómo separar el código a lo largo de diferentes objetos es un arte complejo de dominar, pero hay principios interesantes y fáciles de aprender cómo SOLID.

self

Dentro de la clase, self es la clase

```
class Klass
  def self.foo
    self
  end
end
```

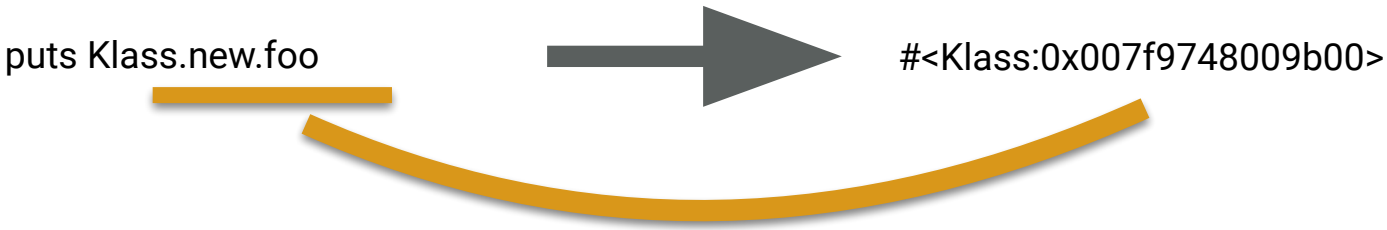


2.3.1 :010 > puts Klass.foo

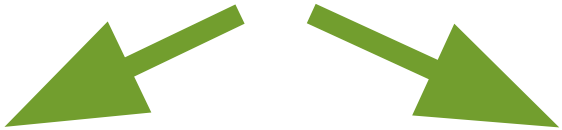
self

Dentro de un método de instancia, self es la instancia

```ruby
class Klass
  def foo
    self
  end
end
```



self



Dentro de la clase sirven para crear un método de clase

Dentro de la instancia sirven para evitar confundir una variable local con un método

self

Evitando confundir una variable local con un método

```ruby
class Circle
  attr_accessor :radius
  def initialize
    @radius = 1
  end

  def bigger
  end

  def to_s
    "círculo de radio #{@radius}"
  end
end

c = Circle.new
print c
```

Resumen

Caso	Consecuencia
radius + 1	Incrementa la variable local si está definida, llama al método getter si la variable no existe
radius = 7	Define una variable local llamada radius
radius = radius +1	Aumenta el valor de la variable local radio en 1, si la variable local no está definida obtendremos un error
@radius + 1	Utiliza la variable de instancia
@radius = @radius + 5	Utiliza la variable de instancia
self.radius + 1	Llama al método getter de radius
self.radius = 7	Utiliza el método setter de radius

Los casos peligrosos

radius = 7	Define una variable local llamada radius
radius = radius +1	Aumenta el valor de la variable local radio en 1, si la variable local no está definida obtendremos un error

Variables de clase

Así como las instancias tienen sus métodos y variables

Las clases también tienen sus propios métodos (métodos de clase) y variables

Variables de clases

En ruby se crean con @@.

```ruby
class T
  @@foo = 5
  def self.bar
    @@foo
  end
end

T.bar # => 5
```

Un ejemplo útil de variables de clases

Contador de instancias

```
class T
  @@instances = 0
    def initialize()
    @@instances +=1
  end

  def self.get_number_of_instances
    @@instances
  end

end

10.times do |i|
    T.new
end
```

T

¿Es correcto o no?

```ruby
class Alumno
  def initialize()
      @notas = []
      nombre = "Humberto"
  end

  def self.cantidad_de_alumnos
     10
  end
end
```

- Alumno.new.cantidad_de_alumnos
- Alumno.new.class.cantidad_de_alumnos
- a = Alumno.new; a.cantidad_de_alumnos
- Alumno.class.cantidad_de_alumnos

Herencia

La herencia es un mecanismo que le permite a una clase adoptar los atributos y comportamientos de otra clase.

Clase Padre

```ruby
class Person
  attr_accessor :name, :age
  def initialize(name)
    @name = name
    @age = 0
  end
  def get_older
    @age += 1
  end
end
```

Clase Hija

```ruby
class Company < Person


end
```

```ruby
c = Company.new("DesafioLatam")
c.get_older
c.get_older
puts c.age
```

¿Por qué es importante la herencia?

Nos permite reutilizar código y lo tenemos que manejar por que muchas clases de Rails la ocupan.

Una clase puede reescribir un método de su clase padre

```ruby
class Person
  attr_accessor :name, :age
  def initialize(name)
    @name = name
    @age = 0
  end
  def get_older
    @age += 1
  end
end
```

```ruby
class Company < Person
  def get_older
    @age += 2
  end
end
```

```ruby
c = Company.new("DesafioLatam")
c.get_older
c.get_older
puts c.age
```

Super

Super nos permite llamar a un método de la clase padre que se llame exactamente igual

```ruby
class Parent
  def foo
    puts 'hola'
  end
end
```

```ruby
class Child < Parent
  def foo
    puts 'antes'
    super
    puts 'después'
  end
end
```

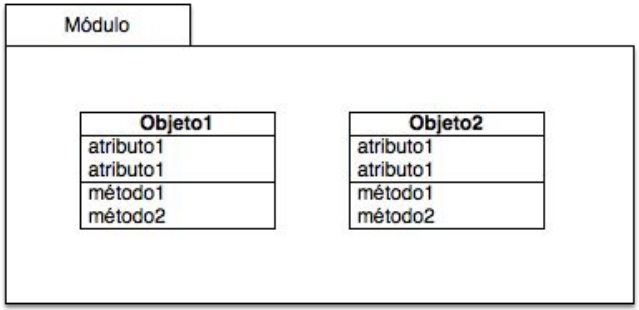
```ruby
Child.new.foo
```

antes
hola
después

Ruby
Módulos y mixins

Introducción a módulos

Los módulos en ruby tienen diversas funciones, una importante es agrupar objetos, también pueden agrupar otros módulos



Hacer un módulo no es muy distinto a hacer un objeto

En este caso, self es importante porque nos permite llamar el método desde fuera del módulo.

```ruby
module Foo
  def self.bar
    puts "Desafío !!!"
    puts "LATAM !!!"
  end
end


Foo.bar
```

Los módulos pueden tener constantes que son fácilmente accesibles

```
module Foo
  D = 20
end

puts Foo::D
```

Dentro de los módulos podemos poner objetos

```
module Foo
  class Bar
    def initialize()
      puts "hola"
    end
  end
end
```

```
Foo::Bar new
```

De esta forma podemos generar namespaces

Módulos vs clases

```ruby
class Formula
  @@pi = 3.1415

  def self.pi
    @@pi
  end

  def self.diameter(r)
    2*r
  end

  def self.perimeter(distance)
    diameter(distance) * pi
  end
end
```

Formula.pi

```ruby
module Formula
  PI = 3.1415

  def self.diameter(r)
    2*r
  end

  def self.perimeter(distance)
    diameter(distance) * PI
  end
end
```

Formula::PI

Módulos

- Sirven para agrupar el código bajo un namespace
- Son útiles para guardar constantes
- Sirven para implementar mixins

Clases dentro de módulos

```ruby
module Mammal
  class Dog
    def speak(sound)
      p "#{sound}"
    end
  end

  class Cat
    def say_name(name)
      p "#{name}"
    end
  end
end
```

Llamamos a las clases dentro de los módulos anexando el nombre de la clase con el nombre del módulo por medio de : :

Instanciando clases específicas dentro de módulos

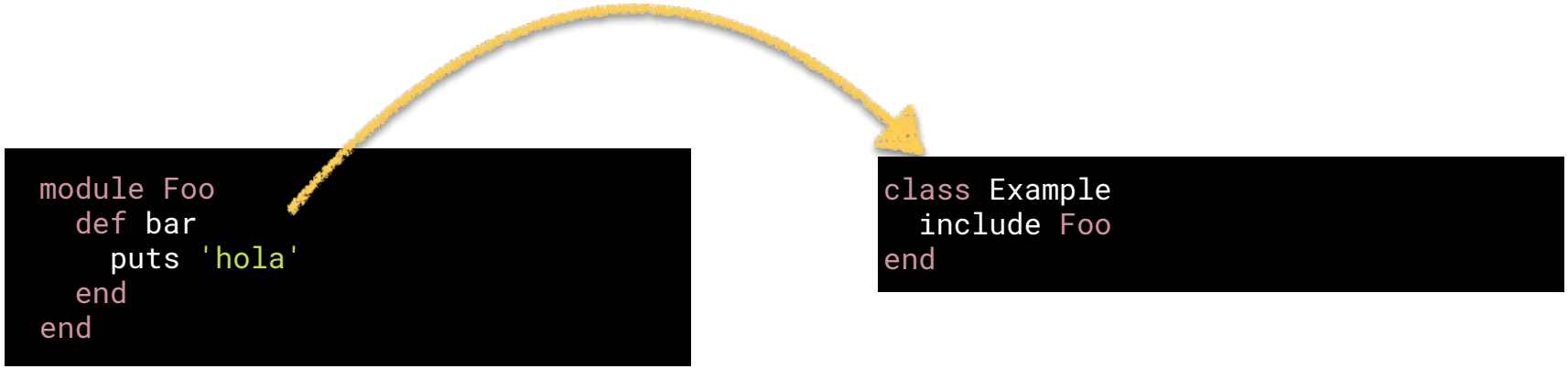
```ruby
buddy = Mammal::Dog.new
kitty = Mammal::Cat.new

buddy.speak('Arf!')             # => "Arf!"
kitty.say_name('kitty')         # => "kitty"
```

Mixins

Los módulos se pueden incluir o extender por otras clases

Mixin

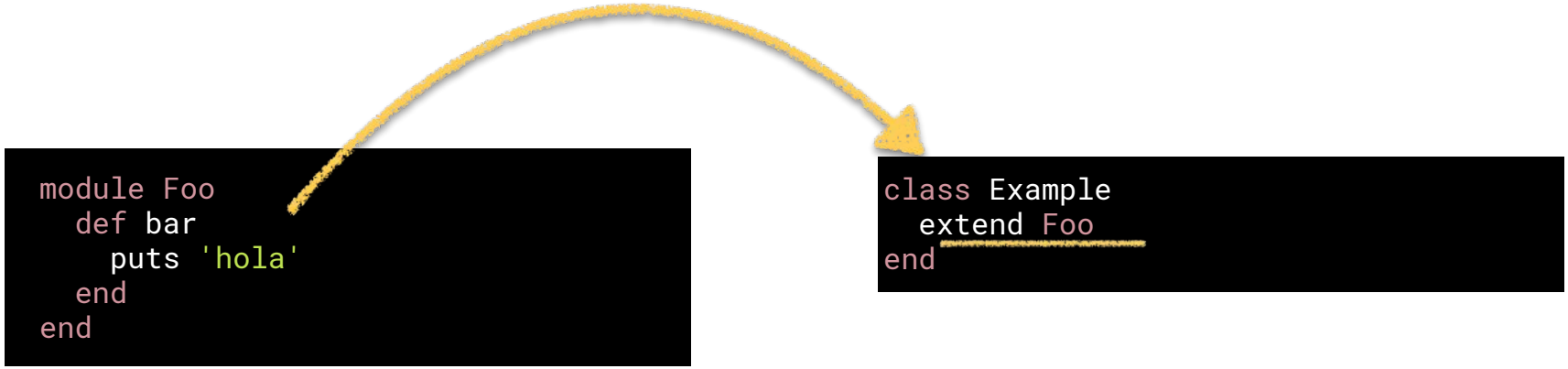


```ruby
module Foo
  def bar
    puts 'hola'
  end
end
```

```ruby
class Example
  include Foo
end
```

include permite integrar los métodos de un módulo como métodos de instancia

Mixin



extend permite integrar los métodos de un módulo como métodos de clase

Módulos y herencia

```ruby
module Nadador
  def nadar
    puts "Puedo nadar!"
  end
end

class Animal; end

class Pez < Animal
  include Nadador  #mixing en módulo Nadador
end

class Mamifero < Animal
end

class Gato < Mamifero
end

class Perro < Mamifero
  include Nadador  #mixing en módulo Nadador
end
```

- Sólo puedes heredar de una clase, pero puedes hacer mixing de muchos módulos.
- No se puede instanciar módulos, los módulos son para hacer namespacing y para agrupar métodos en común.

```ruby
cocky = Perro.new
nemo = Pez.new
garfield = Gato.new

cocky.nadar          # => I'm swimming!
nemo.nadar           # => I'm swimming!
galfield.nadar       # => NoMethodError: undefined method
```