

OBJETOS AVANZADOS

MUTABILIDAD

CONCEPTOS CLAVES DE ESTA CLASE

- Métodos de clase
- Variables de clase
- self
- herencia
- super

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

```
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.”

!! ES OBVIO !!

- 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

```
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

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

SELF



SELF

Dentro de la clase, self es la clase

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

```
2.3.1 :010 > puts Klass.foo
```



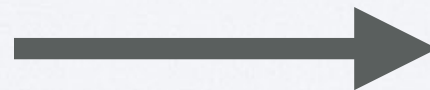
Klass

SELF

Dentro de un método de instancia, self es la instancia

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

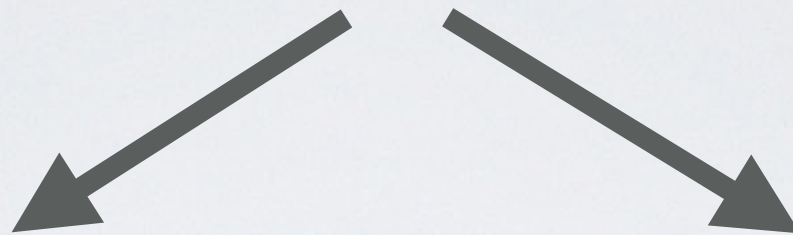
```
puts Klass.new.foo
```



```
#<Klass:0x007f9748009b00>
```



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

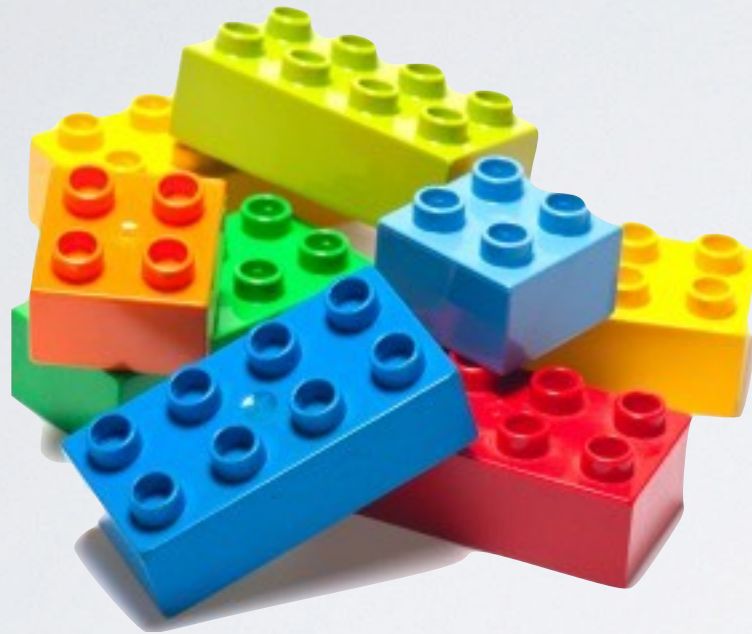
```
class Circle
  attr_accessor :radius
  def initialize
    @radius = 1
  end

  def bigger
  end

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

c = Circle.new
print c
```


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 @@.

A yellow arrow points from the text "con @@." to the line "@@foo = 5" in the code block.

```
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.get_number_of_instances
# => 10
```

NO PROFUNDIZAREMOS MÁS EN VARIABLES
DE CLASE PORQUE NO LAS OCUPAREMOS
MUCHO MÁS ALLÁ DE LO VISTO.

ES CORRECTO O NO?

```
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

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

Clase Hija

```
class Company < Person
  attr_accessor :name, :age
end
```

```
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

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

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

```
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

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

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

```
Child.new.foo
```

antes
hola
despues