Profundizando en objetos _

Introducción a la unidad

¿Qué aprenderemos en esta unidad?

En esta unidad profundizaremos en nuevos principios de la programación orientada a objetos:

- **Herencia**
- **Polimorfismo**

Recordemos que estos principios son agnósticos al lenguaje de programación y de momento hemos cubierto solo dos: Abstracción y encapsulación

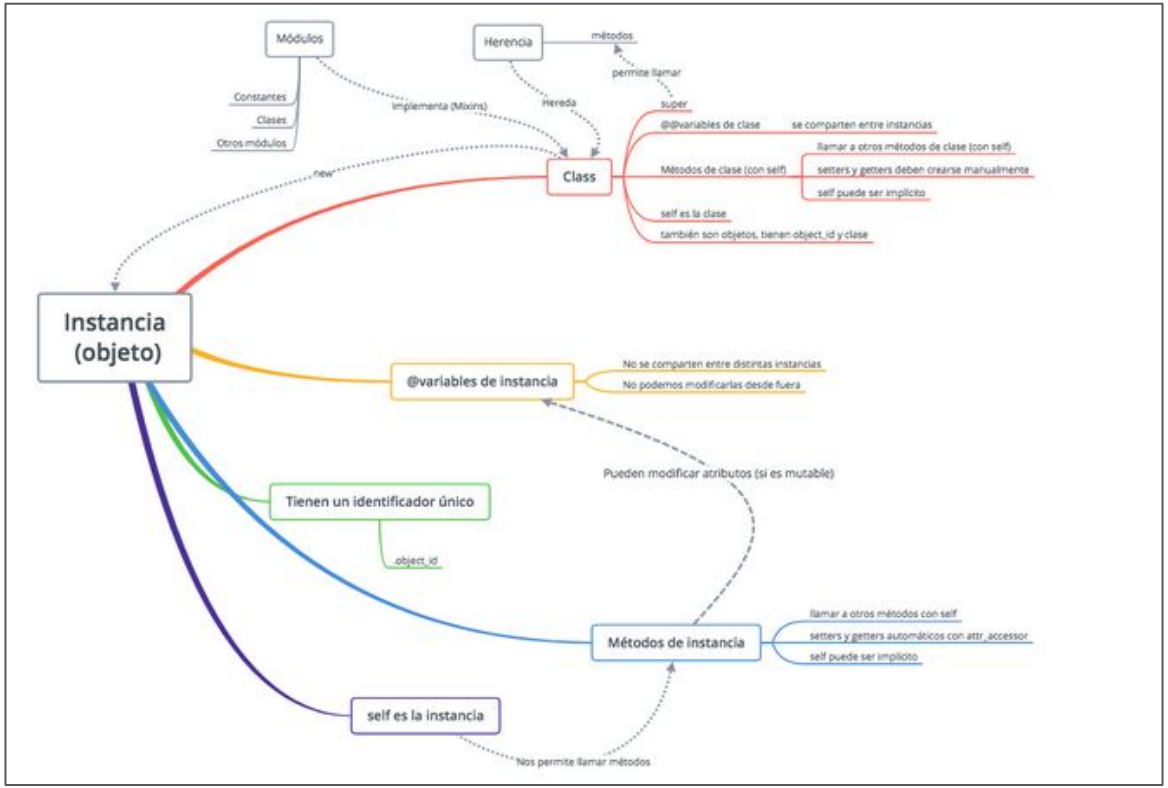
Conceptos claves que estudiaremos

En esta experiencia abordaremos diversos nuevos conceptos, los más importantes son:

- **Herencia**
- **Módulos**
- **Mixins**

Te recomendamos mantener el glosario y el siguiente mapa conceptual a mano.

Mapa conceptual:
Programación orientada a objetos



Herencia

Objetivos

- Crear clases que hereden de otras clases
- Llamar a un método heredado de la clase padre
- Sobrescribir un método heredado de la clase padre
- Agregar instrucciones al flujo de un método heredado de la clase padre utilizando la instrucción **`super`**.

Introducción a Herencia

Una clase hija hereda los **atributos** (@variables) y el **comportamiento** (métodos) de su clase padre.

Esta implementación es tan simple como agregar el operador **<** en la definición de la clase.

```
class Hija < Padre
end
```

Especialización

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

class Hija < Madre
 def initialize(nombre)
   @nombre = nombre + " jr"
 end
end

madre = Madre.new('Ada')
hija = Hija.new(madre.nombre)
hija.nombre
```

Ejercicio

Crear la clase **Moto** que herede de vehículo y posea solo 2 ruedas.

```ruby
class Vehiculo
 def initialize
     @ruedas = 4
 end

 def arrancar
     puts 'rRRRRRrRRRRRrrR'
 end

 def detenerse
     puts 'El motor se ha detenido...'
 end
end
```

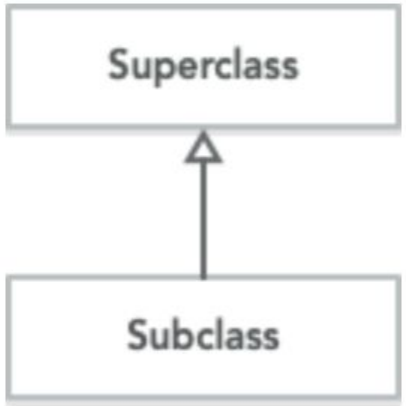
Ejercicio: Solución

```
class Vehiculo
 def initialize
     @ruedas = 4
 end

 def arrancar
     puts 'rRRRRRrRRRRRrrR'
 end

 def detenerse
     puts 'El motor se ha detenido...'
 end
end

class Moto < Vehiculo
 def initialize
   @ruedas = 2
 end
end
```

Herencia y UML

A veces uno suele referirse a la clase padre como **superclase** y a la clase hija como **subclase**.

En **UML**, la herencia se representa con una **flecha con punta triangular vacía** que apunta hacia la superclase.



Herencia y UML: ejemplo práctico

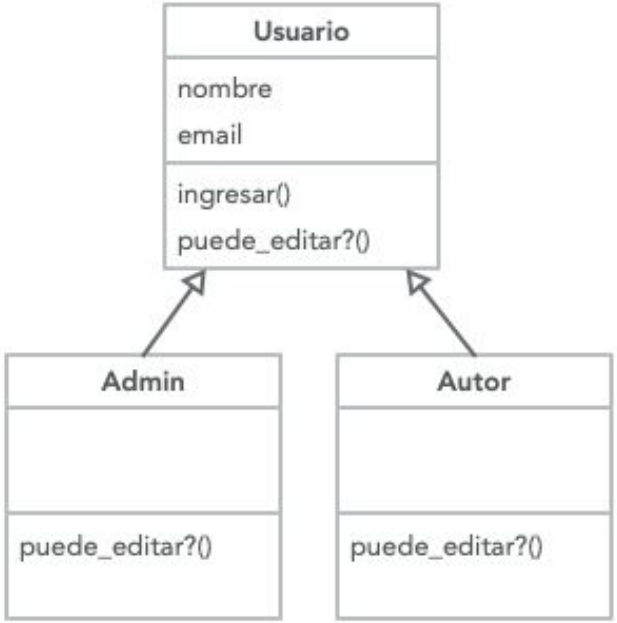
Necesitamos documentar una aplicación web donde construiremos un Blog con diferentes tipos de usuario.

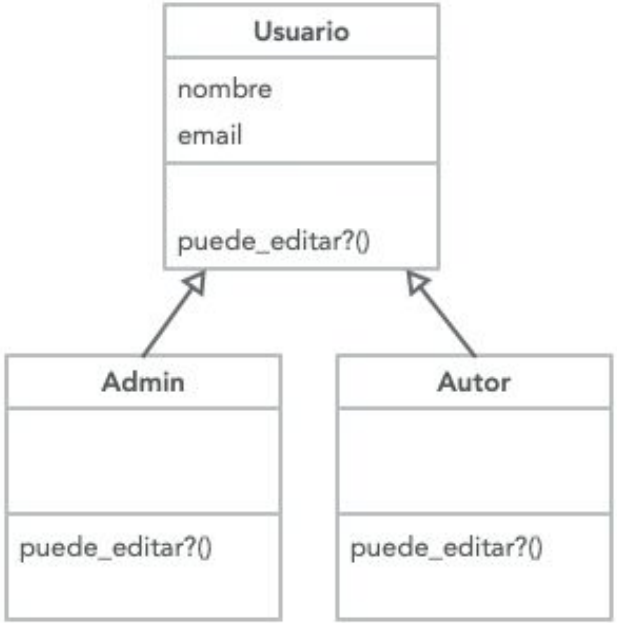
Necesitamos 3 tipos de usuario:

- Usuarios
- Autores
- Administradores.

Los usuarios no pueden editar documentos; los autores y administradores, sí. El resto de los atributos será común para todos.

Herencia y UML: ejemplo práctico






```ruby
class Usuario
 def initialize(nombre, email)
   @nombre = nombre
   @email = email
 end
 def puede_editar?
   false
 end
end

class Admin < Usuario
 def puede_editar?
   true
 end
end

class Autor < Usuario
 def puede_editar?
   true
 end
end

# Usa el constructor de la superclase
a1 = Autor.new('Erick', 'erick@desafiolatam.com')
a1.puede_editar? # => true
```

¿Qué sucede en el siguiente ejemplo?

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

 def despertar
     puts 'Es hora de despertar'
 end
end

class Hija < Madre
 def despertar
     puts 'Es hora de despertar para ir al colegio!'
 end
end
```

Analicemos los métodos `despertar()` de cada clase

```ruby
# Madre
def despertar
 puts 'Es hora de despertar'
end

# Hija
def despertar
 puts 'Es hora de despertar para ir al colegio!'
end
```

¿Puedes observar que lo que necesitamos es **agregar comportamiento** al método heredado?

Dicho de otra forma, necesitamos **especializar la clase hija sin tener que sobreescribir todo el método**.

Para ello existe una instrucción, se denomina **`super`**.

```ruby
# Madre
def despertar
 puts 'Es hora de despertar'
end


# Hija
def despertar
 puts 'Es hora de despertar para ir al
colegio!'
end
```

SUPER

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

 def despertar
     'Es hora de despertar'
 end
end

class Hija < Madre
 def despertar
     super + ' para ir al colegio!'
 end
end
```

SUPER

```ruby
class X
 def metodo_complejo
   puts "durante"
 end
end

class Y < X
 def metodo_complejo
   puts "antes"
   super
   puts "después"
 end
end

Y.new.metodo_complejo
```

Cadena de ancestros

```ruby
class Abuelo
end

class Padre < Abuelo
end

class Hijo < Padre
end

Hijo.ancestors
# [Hijo, Padre, Abuelo, Object, Kernel, BasicObject]
```

Una clase puede heredar de otra clase que herede.

Con el método **`ancestors`** podemos conocer la cadena completa de herencia.

Herencia simple vs Herencia múltiple

```ruby
class A < B
end

class A < C
end

# TypeError (superclass mismatch for class A)
```

En Ruby, **una clase puede heredar sólo de una clase**. Esto se denomina herencia simple.

Resumen

Herencia es un mecanismo que nos permite organizar y reutilizar mejor nuestro código, además de modificar instrucciones de clases creadas por terceros sin intervenir directamente estas clases sino más bien las propias que heredamos de ellas.

1. Podemos heredar comportamiento y utilizarlo
2. Podemos sobrescribir comportamiento heredado
3. Podemos agregar instrucciones al comportamiento heredado a través de la instrucción **`super`**

Variables de clase

Objetivos

- Definir variables de clase
- Distinguir una instancia de una de clase
- Utilizar variables de clase desde un método de instancia

Introducción

Hasta el momento hemos trabajado exclusivamente con variables locales y variables de instancia. En este capítulo introduciremos un nuevo tipo de variable llamado **variable de clase**.

Las clases pueden tener variables propias.

Esto nos permite tener estados compartidos a lo largo de múltiples objetos.

Nuestra primera variable de clase

Analicemos el siguiente ejemplo: Necesitamos saber cuántas piezas de Lego se han creado.

1. **¿Tiene sentido que una pieza de Lego almacene la información correspondiente a cuántas piezas existen?**

   Para efecto de esta unidad, la respuesta de no.

2. **¿Tiene sentido que la "máquina fabricadora de Legos" (la clase) almacene la cuenta de piezas creadas?**

   Cuando entendemos este acercamiento a la solución es cuando nos damos cuenta que necesitamos una variable de clase.

Las variables de clase se definen dentro de la clase y comienzan con @@

```
class Lego
 @@count = 0
end
```

Ejemplo variable de clase

```ruby
class Lego
 @@count = 0 # Declaración variable de clase
 def initialize(size)
   @size = 1
   @@count += 1 # Aumenta en uno.
 end
end

piece1 = Lego.new(1)
piece2 = Lego.new(2)
piece3 = Lego.new(3)
```

¿Cómo accedemos a nuestra variable de instancia?

Para comprobar que nuestra variable de clase está almacenando la información que necesitamos podemos crear, **respetando el principio de encapsulación**, un método que retorne su valor.

Accediendo a la variable de clase

```ruby
class Lego
 @@count = 0
 def initialize(size)
   @size = 1
   @@count += 1
 end
 def total_pieces
   @@count # Retorna valor de variable
 end
end

piece1 = Lego.new(1)
piece2 = Lego.new(1)
piece3 = Lego.new(1)

piece1.total_pieces
```

Entonces nos preguntamos...

1. **¿Tiene más sentido preguntar directamente a la clase la cantidad de instancias creadas?**

   Sí.

2. **¿Cómo accedemos a ella si no es a través de un método de instancia?**

   Para responder esta pregunta debemos retomar un importante principio: el de encapsulación

Recordatorio: Principio de encapsulación

No podemos acceder a la variable **`@@count`** de forma directa por la misma razón que no podemos acceder a los atributos de un objeto sin métodos **getter** y **setter**.

La protección de variables es importante para conservar la integridad de un programa. Es parte importante del paradigma de orientación a objetos y se denomina principio de encapsulación

Entonces

¿Podemos crear un método que nos entregue el valor de la variable de clase sin haber instanciado una pieza de Lego?

La respuesta es sí.

A través de la implementación de un nuevo tipo de método: **método de clase**

Métodos de clase

Objetivos

- Definir métodos de clase
- Diferenciar un método de instancia de un método de clase
- Llamar métodos de instancia y de clase
- Crear métodos getter y setter para variables de clase

Contexto

En el capítulo anterior construimos la clase **Lego** e implementamos un contador de instancias a través de una variable de instancia llamada **@@count**

```ruby
class Lego
 @@count = 0
 def initialize(size)
   @size = 1
   @@count += 1
 end
 def total_pieces
   @@count # Retorna valor de variable
 end
end

piece1 = Lego.new(1)
piece2 = Lego.new(1)
piece3 = Lego.new(1)

piece1.total_pieces
```

Métodos de clase

```ruby
class Lego
 @@count = 0
 def initialize(size)
   @size = 1
   @@count += 1
 end
 def total_pieces
   @@count # Retorna valor de variable
 end
end

piece1 = Lego.new(1)
piece2 = Lego.new(1)
piece3 = Lego.new(1)

piece1.total_pieces
```

```ruby
class Lego
 @@count = 0
 def initialize(size = 1)
   @size = size
   @@count += 1
 end
 def self.total_pieces # Método de clase
   @@count
 end
end

piece1 = Lego.new
piece2 = Lego.new
piece3 = Lego.new
```

```ruby
puts Lego.total_pieces # Se llama desde la clase
20.times{ Lego.new }
puts Lego.total_pieces
```

Anatomía de un método de clase

En el ejemplo utilizamos la palabra reservada **`self`**.

```ruby
class MiClase
 def self.metodo_de_clase
   # Lógica del método de clase
 end
end
```

Por ahora simplemente diremos que **`self`** nos permite definir un método de clase. Profundizaremos en el concepto de **`self`** en el próximo capítulo.

Los métodos de clase pueden comenzar con **self** o con el **nombre de la clase**.

Es mucho más frecuente que comiencen con **self**.

```ruby
class Lego
 @@count = 0
  def initialize(size = 1)
    @size = size
    @@count += 1
  end
  def Lego.total_pieces
    @@count
  end
end

10.times.each { Lego.new }

puts Lego.total_pieces
```

¡Importante!

Esto ya lo hemos mencionado pero es tan importante que lo volveremos a repetir:

Los métodos de clase se llaman directamente desde la clase y no desde la instancia.

```
Lego.total_pieces # ¡Bien!
Lego.new.total_pieces  # Mal...
Lego.new.class.total_pieces # Forma rebuscada que también funciona.
```

Getter y Setter

```ruby
class Estudiante
  @@cantidad_de_estudiantes = 0
  def self.cantidad_de_estudiantes
     @@cantidad_de_estudiantes
  end

  def self.cantidad_de_estudiantes=(valor)
     @@cantidad_de_estudiantes = valor
  end
end

Estudiante.cantidad_de_estudiantes = 10
puts Estudiante.cantidad_de_estudiantes
```

self

Objetivos

- Conocer el rol de **`self`** dentro de una clase.
- Hacer uso de **`self`** para llamar métodos de instancia.
- Hacer uso de **`self`** para llamar métodos de clase.

Las reglas del uso de `self` son muy importantes para trabajar con objetos en Ruby.

¿Qué es `self`?

`self` es una palabra reservada que nos da acceso al objeto actual.

Dicho de otra forma, utilizando **`self`** podemos acceder al “dueño” del código donde lo estamos utilizando.

self

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

 def cantar
   "Soy #{self.nombre} y estoy cantando!"
 end

end

persona_amable = Persona.new('John Doe')
puts persona_amable.cantar
```

self

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

 def cantar
   "Soy #{self.nombre} y estoy cantando!"
 end

 def ir_a_karaoke
   self.cantar
 end
end

persona_amable = Persona.new('John Doe')
puts persona_amable.cantar
puts persona_amable.ir_a_karaoke
```

Es **muy común** llamar a un método del objeto dentro del objeto mismo.

```ruby
class Persona
 def initialize
   @edad = 0
 end
 def envejecer
   @edad += 1
 end
 def envejecer_rapido
   10.times { self.envejecer }
 end
end
```

Y si utilizamos la instrucción **self** por sí sola en un método de instancia:

¿Qué obtendremos como retorno?

En el ejemplo podemos corroborar que **self**, dentro de un método de instancia, está retornando la instancia misma.

```ruby
class Fantasma
 def reflejar
   self
 end
end

fantasma = Fantasma.new
fantasma.reflejar == fantasma
# true
```

self en un método de clase

```ruby
class Fantasma
 def self.reflejar
   self
 end
end

fantasma = Fantasma.new
Fantasma.reflejar == fantasma # false
Fantasma.reflejar == Fatasma # true
```

Resumen: self

El uso de `self` dentro de un método de instancia hace referencia a la instancia misma.

El uso de `self` dentro de un método de clase hace referencia a la clase misma.

Polimorfismo

Objetivos

- Conocer principio de polimorfismo en programación orientada a objetos.
- Utilizar polimorfismo para evitar uso innecesario de sentencias `if`.

¿Qué es polimorfismo?

En programación, los objetos se comunican a través de **mensajes**.

En **Ruby**, este comportamiento se lleva a cabo a través del **llamado de métodos**.

Poli-morfismo: Cualidad de tener muchas formas.

Este principio consiste en que dos objetos, de distinta clase, pueden responder al mismo método de forma distinta.

¿Cómo se relaciona este principio a lo aprendido en esta unidad?

- Es uno de los principios fundamentales de la programación orientada a objetos.
- Una de las formas en que podemos lograr polimorfismo es a través de la herencia.

Analicemos el siguiente ejemplo:

Nada nuevo ¿Verdad?

Sabemos que las clases **`Perro`**, **`Gato`** y **`Vaca`** heredan de **`Animal`**.

```ruby
class Animal
end

class Perro < Animal
end

class Gato < Animal
end

class Vaca < Animal
end
```

Pero… Perro, gato y vaca “hablan” de forma distinta ¿O no?

```ruby
class Animal
 def hablar
 end
end

class Perro < Animal
end

class Gato < Animal
end

class Vaca < Animal
end
```

Ejemplo

```ruby
class Animal
 def hablar
   if self.class == Perro
     puts 'Guau'
   elsif self.class == Gato
     puts 'Miau'
   elsif self.class == Vaca
     puts 'Muuu'
   end
 end
end
```

```ruby
class Perro < Animal
end

class Gato < Animal
end

class Vaca < Animal
end

Perro.new.hablar # Guau
Vaca.new.hablar # Muuu
```

Principio de polimorfismo al rescate :)

Las instancias de **Perro**, **Vaca** y **Gato** responden al mismo método hablar de forma distinta.

Hemos utilizado los principios de polimorfismo y herencia para que nuestro código sea ordenado y escalable.

```ruby
class Animal
end

class Perro < Animal
 def hablar
   puts 'Guau'
 end
end

class Gato < Animal
 def hablar
   puts 'Miau'
 end
end

class Vaca < Animal
 def hablar
   puts 'Muuu'
 end
end

Perro.new.hablar # Guau
Vaca.new.hablar # Muuu
```

Módulos y mixins

Introducción

Los módulos en Ruby se caracterizan por ser elementos que nos permiten **agrupar**.

Imagina un **módulo** como un **set de herramientas** que, dependiente del escenario, podemos **disponibilizar** y/o **utilizar directamente**.

Introducción

Los módulos tienen 4 funciones principales:

1. Agrupar constantes.
2. Agrupar métodos.
3. Agrupar clases.
4. Evitar colisiones de nombre.

Cabe destacar que estas funcionalidades no son excluyentes entre ellas. Un mismo módulo puede agrupar constantes, métodos y/o clases.

Nuestro primer módulo

Crear un módulo no es muy distinto a crear una clase, la diferencia radica en que utilizaremos la palabra reservada **`module`**

```
module MiModulo
end
```

A continuación estudiaremos cada una de las funcionalidades descritas

1. **Agrupando Constantes**

```ruby
module MyMath
 PI = 3.14
 E = 2.718
end
```

Podemos acceder a las constantes contenidas en este módulo con la sintaxis **`Modulo::CONSTANTE`**

```ruby
module MyMath
 PI = 3.14
 E = 2.718
end

puts MyMath::PI # 3.14
puts MyMath::E # 2.718
```

De hecho, el módulo **Math** existe dentro de Ruby y lo podemos utilizar de la misma forma.

```ruby
puts Math::PI # 3.141592653589793
```

2. Agrupando Métodos

```ruby
module MyMath
 PI = 3.14
 E = 2.718
 def self.sumar(x, y)
   x + y
 end

 def self.restar(x, y)
   x - y
 end

 def self.multiplicar(x, y)
   x * y
 end
end

MyMath.sumar(4,3) # 7
MyMath.restar(4,3) # 1
MyMath.multiplicar(4,3) # 12
```

3. Agrupando Clases

```ruby
module TiposDeUsuario

 class Usuario
 end

 class Conductor < Usuario
 end

 class Pasajero < Usuario
 end

end

TiposDeUsuario::Conductor.new
#<TiposDeUsuarios::Conductor:0x00007feb9e03d040>
```

¿Puede una clase ser definida dos veces?

Sí y sus contenidos serán mezclados. La último definición será agregada a la primera.

Esto se conoce como **extender una clase**.

```ruby
class Usuario
 def saludar
   'Hola!'
 end
end

class Usuario
 def presentar
   'Soy un usuario!'
 end
end

usuario = Usuario.new
usuario.saludar # Hola!
usuario.presentar # Soy un usuario!
```

Podemos incluso extender clases que ya existen en Ruby.

En programación, esta implementación se conoce como **monkey patching**.

Debemos poner cuidado en el motivo por el cual lo estamos haciendo ya que puede ser considerado una mala práctica.

```ruby
class String
 def palindrome?
   self.reverse == self
 end
end

'reconocer'.palindrome? #true
'esto no es palindrome'.palindrome? #false
```

3. **Evitar colisiones de nombre**

```ruby
module ActiveRecord
 class Base
 end
end

module ActionView
 class Base
 end
end
```

Bonus: Anidación

Un módulo también puede contener otros módulos.

```ruby
module Money
 module Currency
   class Dolar
   end
 end
end

Money::Currency::Dolar
```

Mixins

Los **mixins** nos permiten incorporar los métodos definidos en un módulo como **métodos de instancia o de clase** de un objeto.

Para incluir los métodos de un módulo como **métodos de instancia** utilizaremos la instrucción **`include`**.

Para incluir los métodos de un módulo como **métodos de clase** utilizaremos la instrucción **`extend`**.

Mixins

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

```ruby
class Gato
end

class Perro
 extend Nadador
end

Perro.nadar # Puedo nadar!
```

La implementación de mixins es la solución a la multiherencia

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

module Carnivoro
 def comer
   puts 'Puedo comer carne!'
 end
end
```

```ruby
class Mamifero
end

class Gato < Mamifero
 include Carnivoro
end

class Perro < Mamifero
 include Nadador
 include Carnivoro
end
```

Resumen

- Un **módulo** le definimos como un **set de herramientas** que podemos utilizar de manera directa o disponibilizar su contenido para su utilización en una o varias clases.
- Destacamos 4 funciones principales de los módulos.
    - Agrupar constantes
    - Agrupar métodos
    - Agrupar clases
    - Evitar colisiones de nombre
- La técnica que nos permite incluir módulo en una clase se denomina **mixin**. Podemos incluir, a través de mixins, tantos módulo como queramos.
- Si utilizamos la instrucción **include**, los métodos del módulo serán incluidos como métodos de instancia.
- Si utilizamos la instrucción **extend**, los métodos del módulo serán incluidos como métodos de clase.

Gosu

Objetivos

- Estudiar de manera práctica los conceptos aprendidos de herencia, polimorfismo, módulos y mixins.
- Crear un videojuego utilizando el framework Gosu

Introducción

- Gosu es un framework para desarrollar juegos de videos en Ruby y C++.

- Si bien desarrollar juegos está fuera del alcande del curso, este framework es bien sencillo y nos permitirá experimentar de forma práctica con los conceptos que hemos ido aprendiendo sobre objetos.

Instalación

Gosu funciona en Linux, Windows y macOS, la instrucciones de instalación las podemos encontrar en la página oficial.

Este capítulo está basado en el el siguiente tutorial.

1. **La clase Tutorial**

```
require 'gosu'

class Tutorial < Gosu::Window
 def initialize
   super 640, 480
   self.caption = "Tutorial Game"
 end
  def update
   # ...
 end
  def draw
   # ...
 end
end

Tutorial.new.show
```

2. Utilizando imágenes

```
require 'gosu'

class Tutorial < Gosu::Window
 def initialize
   super 640, 480
   self.caption = "Tutorial Game"

   @background_image =
Gosu::Image.new("media/space.png", :tileable =>
true)
 end
  def update
 end
  def draw
   @background_image.draw(0, 0, 0)
 end
end

Tutorial.new.show
```

3. Player y sus movimientos

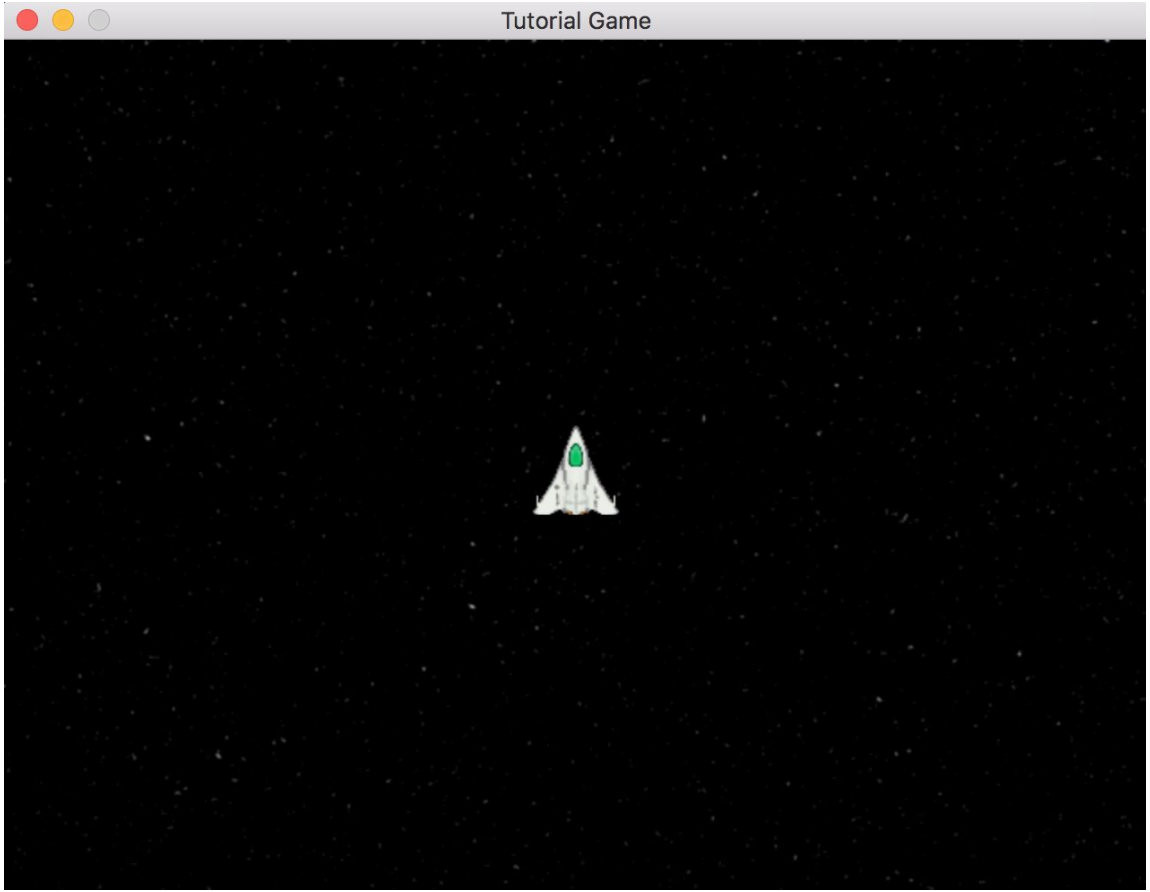
```
class Player
 def initialize
   @image = Gosu::Image.new("media/starfighter.bmp")
   @x = @y = @vel_x = @vel_y = @angle = 0.0
   @score = 0
 end

 def warp(x, y)
   @x, @y = x, y
 end
  def turn_left
   @angle -= 4.5
 end
  def turn_right
   @angle += 4.5
 end
  def accelerate
   @vel_x += Gosu.offset_x(@angle, 0.5)
   @vel_y += Gosu.offset_y(@angle, 0.5)
 end
  def move
   @x += @vel_x
   @y += @vel_y
   @x %= 640
   @y %= 480

   @vel_x *= 0.95
   @vel_y *= 0.95
 end

 def draw
   @image.draw_rot(@x, @y, 1, @angle)
 end
end
```

4. Clase Tutorial 2.0: Utilizando la clase Player

```
class Tutorial < Gosu::Window
 def initialize
   super 640, 480
   self.caption = "Tutorial Game"

   @background_image = Gosu::Image.new("media/space.png", :tileable => true)

   @player = Player.new
   @player.warp(320, 240)
 end

 def update
   if Gosu.button_down? Gosu::KB_LEFT or Gosu::button_down? Gosu::GP_LEFT
     @player.turn_left
   end
   if Gosu.button_down? Gosu::KB_RIGHT or Gosu::button_down? Gosu::GP_RIGHT
     @player.turn_right
   end
   if Gosu.button_down? Gosu::KB_UP or Gosu::button_down? Gosu::GP_BUTTON_0
     @player.accelerate
   end
   @player.move
 end

 def draw
   @player.draw
   @background_image.draw(0, 0, 0)
 end

 def button_down(id)
   if id == Gosu::KB_ESCAPE
     close
   else
     super
   end
 end
end

Tutorial.new.show
```

¡Ya podemos ejecutar nuestro juego!

Nuestro terminal quedará secuestrado hasta que, dentro del juego, presionemos la tecla *Escape*.


Tutorial Game

Hemos aplicado lo aprendido en esta unidad a través del framework **Gosu** y su tutorial.

¿Te animas a continuar el tutorial de Gosu para crear elementos con los que nuestra nave espacial pueda interactuar?

{desafío} latam_

Academia de talentos digitales

www.desafiolatam.com