Profundizando en objetos

Lo que debes saber antes de esta unidad:

Programación Orientada a objetos

La programación orientada a objetos es un paradigma de programación que se creó con el objetivo de pensar los problemas de manera abstracta. Consiste en agrupar los elementos de un programa en torno a objetos de la vida real. Estos objetos son entidades que contienen atributos y comportamientos. Gracias a este paradigma podemos tanto organizar y reutilizar fácilmente nuestro código como abstraernos de varios de los procesos.

Conceptos claves

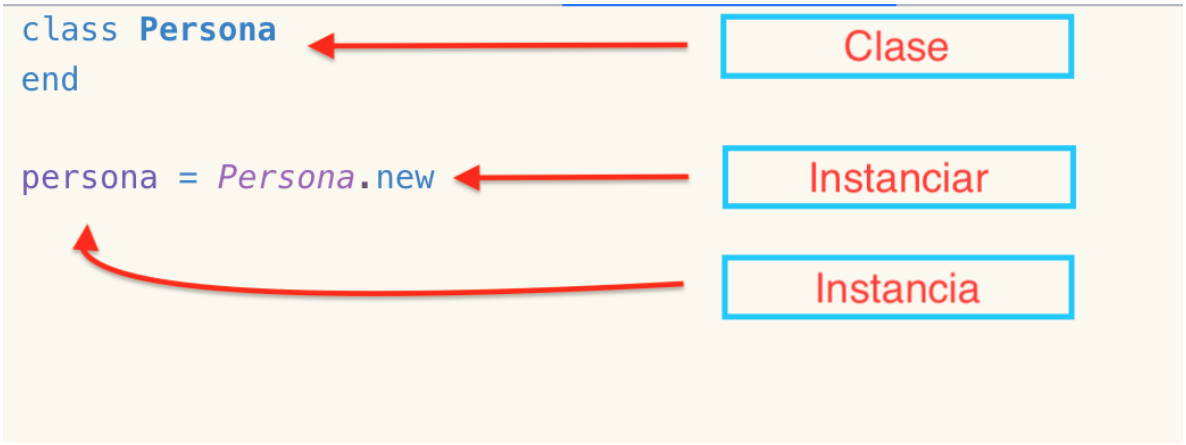
Los 3 conceptos más recurrentes de la programación orientadas a objetos son:

- Clase
- Instanciar un objeto
- Instancia (Objeto)

Lo anterior se traduce a dos acciones de nuestra parte

```ruby
# Creamos la clase
class Persona
end

# Creamos una instancia
persona = Persona.new
```

Podemos ver estos conceptos resumidos en la siguiente imagen:



Métodos de instancia

Podemos definir métodos dentro de nuestras clases. Estos métodos reflejan el comportamiento del objeto y reciben el nombre de métodos de instancia, por ejemplo, una persona (instancia) puede saludar:

```
class Persona
  def saludar
    puts "hola!!"
  end
end

persona = Persona.new
persona.saludar
# hola!!
```

En ruby también podemos llamar al método inmediatamente despues de crear la instancia.

```
Persona.new.saludar#=>hola!!
```

Guardando un estado

Para guardar los estados de un objeto (atributos) utilizaremos variables de instancia, estas se distinguen de variables locales en su sintaxis ya que comienazan con el caracter `@` que deben ser definidas y utilizadas dentro de métodos de instancia.

Por ejemplo, un vehículo puede estar encendido o apagado, como en el siguiente código:

```
class Vehiculo
    def encender()
        @encendido = true
    end
    def apagar()
        @encendido = false
    end
    def estado()
        @encendido
    end
end
```

O una persona debe poseer nombre y apellido:

```
class Persona
    def initialize(nombre, apellido)
        @nombre = nombre
        @apellido = apellido
    end
end
```

Alcance de las variables de instancia

Las variables de instancia sirven para reflejar estados de un objeto, por lo tanto, es lógico y esperable que sus cambios perduren una vez finalizada la ejecución de un método que modifica los valores de una o más variables de instancia.

```
class Test
  def foo
    @a = 5
  end
  def bar
    puts @a
  end
end

test = Test.new
test.foo
test.bar # @a el valor existe despues del end de foo porque @a es una variable
de instancia
```

El método constructor

Podemos asignar valores iniciales a una instancia utilizando un método constructor. En Ruby, un constructor se implementa a través del método llamado `initialize`:

```
class Semaforo
    def initialize(estado)
        @estado = estado
    end
end
s1 = Semaforo.new(:rojo) # => <Semaforo:0x00007f9eda8c6560 @estado=:rojo>
s2 = Semaforo.new(:verde) # => <Semaforo:0x00007f9eda9dbc20 @estado=:verde>
```

Principio de encapsulación

En Ruby todas las variables de instancia son privadas, esto quiere decir que no pueden ser accedidas desde fuera de la clase.

Para demostrarlo intentaremos realizarlo, pero finalmente fallaremos

```
class Mascota
  def initialize(nuevo_nombre)
    @nombre = nuevo_nombre
  end
end

m1 = Mascota.new("Shadow")
m1.nombre
# => NoMethodError: undefined method `nombre' for
    #<Mascota:0x007ff7e38b84f8 @nombre="Shadow">
```

Getters y setters

Si queremos acceder a un estado (variable de instancia) de un objeto tenemos que crear métodos, estos son tan comunes y conocidos que se les llaman **getters y setters**.

```
class Caja
  attr_accessor :ancho, :alto
end
```

Podemos agregar los getters y setters por sepadaro como en el siguiente ejemplo:

```
class Persona
  attr_reader :rut # Define solo un método getter
  attr_writer :edad # Define solo un método setter
end
```

Un **getter** es un método con el mismo nombre de una variable de instancia cuyo objetivo es retornar el valor de la variable en cuestión.

Un **setter** es un método con el mismo nombre de una variable de instancia seguido de un signo `=`, cuyo objetivo de setear un valor a la variable en cuestión.

```
class Persona
  def initialize(nombre)
    @nombre = nombre
  end

  def nombre
    @nombre
  end

  def nombre=(nuevo_nombre)
    @nombre = nuevo_nombre
  end
end
```

Podemos implementar ambos o cada uno de ellos a través de las instrucciones `attr_reader`, `attr_writter` y `attr_accessor`.

Identidad

Cada objeto tiene un identificador que es único y permite diferenciar dos objetos que pueden tener los mismos atributos. Por ejemplo, dos personas llamadas 'Trinidad' son objetos distintos con el mismo nombre.

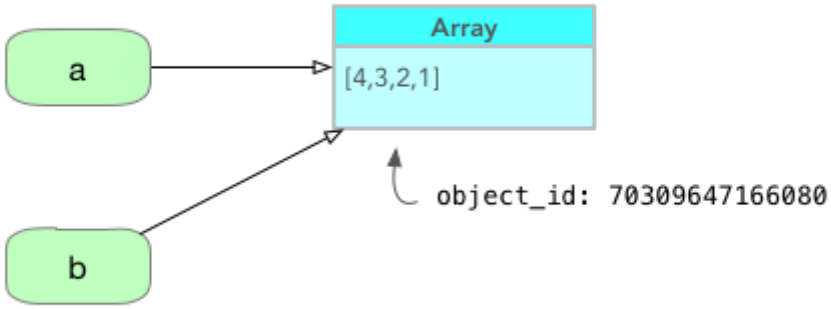
```
class Persona
  attr_accessor :nombre
  def initialize(nombre)
    @nombre = nombre
  end
end

p1 = Persona.new("Trinidad")
p2 = Persona.new("Trinidad")
puts p1 == p2
# false
```

Corroborando que son objetos distintos:

```
persona1 = Persona.new("Trinidad")
persona2 = Persona.new("Trinidad")
persona2.nombre = "Javiera"
puts persona1.nombre #=> Trinidad
puts persona2.nombre #=> Javiera
```

Mutabilidad

En Ruby la mayoría de los objetos son mutables, es decir, podemos modificar sus estados a través del llamado de métodos. Esto es un comportamiento esperado pero tenemos que tener cuidado cuando asignamos el mismo objeto a 2 variables.




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

# En la variable b guardamos a
puts a.object_id == b.object_id # true

a[0] = 8
print b # [8, 2, 3, 4]
```

Al modificar el objeto contenido en la variable `a` estamos modificando el contenido de `b`. El objeto que contienen es el mismo.

Requisitos

Para poder entender los conceptos detrás de los objetos necesitamos:

- Manejar Métodos
  - Paso de parámetros
  - Retorno (y entender que es implícito)
  - Variables locales / globales
  - Alcance de variables

Introducción

¿Qué aprenderemos en esta experiencia?

En esta unidad profundizaremos en nuevos principios de la programación orientada a objetos: **Herencia** y **polimorfismo**.

Estudiar los principios fundamentales de la programación orientada a objetos no es tarea fácil, sin embargo, aprender cuándo y cómo utilizar estos principios facilitará la organización y reutilización de nuestro código además de entregar las competencias necesarias para desarrollar aplicaciones tanto en **Ruby on Rails** como otros framework que utilicen este paradigma.

La programación orientada a objetos tiene 4 principios fundamentales:

- Abstracción
- Encapsulación
- **Herencia**
- **Polimorfismo**

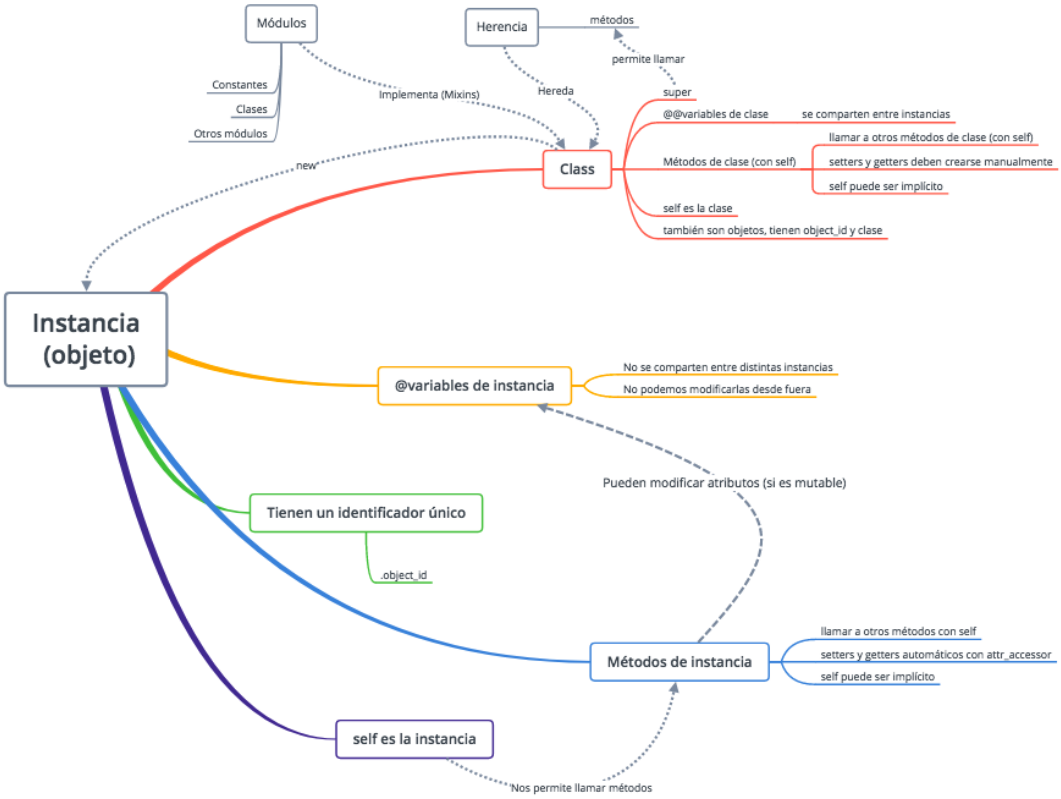
Estos principios son independientes del lenguaje de programación.

De momento solo hemos cubierto los primeros dos: Abstracción y Encapsulación.

Conceptos claves que estudiaremos:

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

- Herencia
- Módulos
- Mixins



Herencia

Objetivos:

- Crear clases que hereden de otras clases.
- Llamar 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

¿Recuerdas que mencionamos que la programación orientada a objetos nos permite abordar los problemas desde una perspectiva más abstracta generando objetos semejantes a la vida real?

Herencia no es la excepción; En biología aprendimos que los seres vivos heredan rasgos genéticos de sus padres.

¡En programación orientada a objetos también aplica!

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
```

Pero, para que esto sea válido, primero la clase `Padre` debe estar definida; entonces:

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

class Hija < Padre
end

fernanda = Hija.new("Fernanda")
fernanda.nombre
```

De este ejemplo podemos destacar que la clase `Hija` **hereda de** la clase `Padre`. Por ende, `Hija` posee todos los métodos y atributos de la clase `Padre`.

Por consiguiente, la clase `Hija` posee también un constructor, un getter y un setter. Entender este comportamiento asegura el éxito en la comprensión de este principio. Saber aplicarlo nos facilitará la organización y reutilización de nuestro código.

Especialización

La herencia nos permite especializar la clase hija. Dicho de otra forma, podemos asignar comportamiento a la clase hija que la clase padre no posee.

Una clase hija puede tener sus propios métodos y atributos e incluso sobrescribir comportamiento heredado.

Para sobrescribir un método heredado, simplemente debemos **definir un método con el mismo nombre del método heredado**. Por ejemplo, podemos sobrescribir el constructor de la clase padre:

```
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
```

Este tipo de comportamiento será **muy importante** en nuestro camino hacia el desarrollo web en Ruby on Rails. A menudo trabajaremos con clases, herencia y la sobrescritura de métodos.

Particularmente, cuando trabajemos con gemas o componentes desarrollados por terceros, no realizaremos modificaciones directas a la herramienta importada, más bien, crearemos una clase que hereda de esta y aplicaremos ahí nuestras modificaciones sobrescribiendo los métodos heredados.

Ejercicio:

- Crear la clase moto que herede de vehículo, pero que posea solo 2 ruedas.

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

    def arrancar
        puts 'rRRRRRrRRRRRrrR'
    end

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

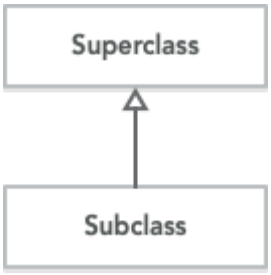
```ruby
## Solución:

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
```

Superclase y subclase

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

Herencia y UML

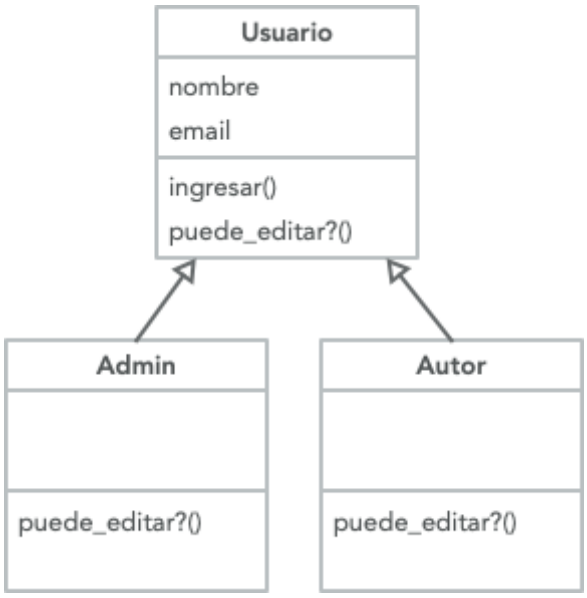
En UML, la herencia se representa con una flecha con punta triangular vacía que apunta hacia la clase padre.



Herencia y UML: Un ejemplo práctico

Imaginemos que necesitamos documentar una aplicación web donde construiremos un Blog con diferentes tipos de usuario. Necesitamos 3 tipos de usuario: usuarios, autores y administradores.

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




```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 está pasando en el ejemplo?

La clase `Autor` hereda de `Usuario`, por lo tanto, recibe el constructor de la clase padre. Sin embargo, el método `puede_editar` es sobrescrito, por lo tanto, imprime `true`.

¿Qué sucede en el siguiente ejemplo?

```
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.

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

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

¿Puedes observar como, a pesar de que estamos sobrescribiendo el método, 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`.

Super

`super` es una instrucción que nos permite llamar a un método de una superclase que se llama exactamente igual. Esto nos permite operar sobre ese método sin tener que sobrescribir el método completo.

```
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
```

Podemos agregar lógica tanto antes como después de `super`. La diferencia radica en dónde necesitamos agregar el comportamiento: ¿Antes o después de la ejecución del método padre?

Analicemos un ejemplo sencillo. Imaginemos que tenemos una clase creada por otro desarrollador. Queremos crear una clase hija y utilizar el mismo método pero agregar lógica antes y después de su ejecución:

```
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

Una clase puede heredar de otra clase que herede:

```
class Abuelo
end

class Padre < Abuelo
end

class Hijo < Padre
end

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

Con el método `.ancestors` podemos observar la cadena completa de herencia. Todas las clases que definamos heredan de la clase `Object`.

Herencia simple vs Herencia múltiple

En Ruby, **una clase solo puede heredar de una clase**. A esto se le llama herencia simple.

```
class A < B
end

class A < C
end

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

Algunos otros lenguajes tienen herencia múltiple, o sea pueden heredar de múltiples clases simultaneamente, las ventajas y desventajas de esto son discutibles, la principal ventaja de la herencia simple es que evita conflictos de nombre.

Para entender más sobre las complejidades de la herencia múltiple se recomienda estudiar el problema del diamante https://es.wikipedia.org/wiki/Problema_del_diamante

Más adelante estudiaremos **módulos y mixins** que son mecanismos que nos permiten incorporar comportamientos de otros objetos sin heredar de ellos.

Resumen

En este capítulo hemos visto temas como **herencia**, el uso de la instrucción *super* y la cadena de ancestros. Aprendimos que: herencia es un mecanismo que nos permite organizar y reutilizar mejor nuestro código, además de modificar instrucciones de clases creadas por terceros sin intervenirlas directamente, sino mas bien las propias que heredamos de ellas. Tambien dimos una mirada a los tipos de herencia y cuales son las ventajas y posibles desventajas de esta. Tambien conocimos la forma en la que usando la instrucción **super** podemos reutilizar métodos de la clase padre y asi mantener nuestro código mas limpio y facil de leer. Y por último conocimos el método `ancestors` el cual nos permite ver la cadena de herencia que tiene una clase.

Variables de clase

Objetivos

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

Introducción a variables de clase

Hasta el momento hemos trabajado exclusivamente con variables locales y variables de instancia, ahora introduciremos un nuevo tipo de variable llamada **variable de clase.**

Las clases pueden tener variables propias, esto permite tener estados compartidos a lo largo de múltiples objetos.

La primera variable de clase

Imaginemos que queremos tener información de cuántas instancias de una clase se han generado.

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 es no.

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

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

   Las variable de clase se definen dentro de la clase y comienzan con `@@`.

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

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

Este ejemplo incorpora una **variable de clase** llamada `@@count`.

Las variables de clase se guardan directamente en la clase y no en la instancia.

El constructor de la clase `Lego` aumenta el valor de la variable de clase en `1`. Esto permite que cada vez que instanciamos un lego llevamos la cuenta del número de piezas creadas.

Esta implementación la podemos llamar un **contador de instancias** y es un ejemplo práctico del uso de variables de clase.

¿Cómo accedemos a nuestra variable de clase?

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

```
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
```

¡Funciona! Sin embargo estamos preguntando a la instancia por la cantidad de instancias creadas. El problema ya no es el almacenamiento de la variable, es el mecanismo de obtención de esa información. Entonces nos volvemos a preguntar

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

Sí.

¿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.

Principio de encapsulación

No podemos acceder a la variable `@@count` de forma directa por el mismo motivo que no podemos acceder a los atributos de un objeto sin un getter y un setter. **Tanto las variables de instancia como las de clase requieren de métodos para poder acceder a ellas**.

Recordatorio: La protección de las variables, a través de métodos, es importante para conservar la integridad de un programa. Es parte importante del paradigma de orientación a objetos y se le denomina principio de encapsulación.

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

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

Resumen

En este capítulo dimos los primeros pasos para conocer las variables de clase y cómo poder accederlas, pero aún nos falta mucho para poder entender su verdadero potencial. En los capítulos siguientes podremos ver más de su uso y otras formas de utilizarlas.

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` y contamos la cantidad de instancias creadas. Para ello, utilizamos una variable de clase `@@count` que llevaba la cuenta.

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

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

piece1.total_pieces
```

Este código tiene un problema ¿Cómo podemos saber cuantas piezas de lego tenemos sin tener que crear una pieza?

La respuesta es con métodos de clase.

Introducción a métodos de clase

Los métodos de clase se aplican directamente sobre la clase sin necesidad de utilizar instancias.

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

puts Lego.total_pieces # Se aplica sobre la clase
20.times{ Lego.new(2) }
puts Lego.total_pieces #=>23
```

Anatomía de un método de clase

En el ejemplo utilizamos la palabra reservada `self`. Por ahora simplemente diremos que `self` nos permite definir un método de clase.

Los métodos de clase pueden comenzar con `self` o con el nombre de la clase, sin embargo, es mucho mas frecuente y es mucho mas mantenible 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 #=> 10
```

Esto ya lo hemos mencionado pero es tan importante que lo repetiremos:`

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

De hecho, 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. En nuestro código total_pieces es un método de clase.
Lego.new.class.total_pieces # Forma innecesariamente rebuscada que también funciona.
```

Getters y setters en métodos de clase

En Ruby no existe una instrucción automática para agregar setters y getters en una variable de clase. Pero los podemos agregar facilmente:

```
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 instacia.
- Hacer uso de self para llamar métodos de clase.

¿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.

De tal manera, utilizando `self`, podemos hacer llamadas al objeto, desde dentro, tal y como si lo estuvieramos llamando desde fuera.

1. Self en un método de instancia:

```
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
```

En el ejemplo:

- `Persona` tiene un método de instancia `cantar`.
- Podemos utilizar `self` dentro de este método para hacer referencia al objeto mismo.
- El objeto posee un getter para el atributo `@nombre` a través de la instrucción `attr_accessor :nombre`.
- Podemos acceder al getter, dentro del método de instancia, utilizando `self.nombre` gracias a la sintáxis que ya conocemos `objeto.metodo`.

En el método `ir_a_karaoke` podemos, nuevamente, utilizar la instrucción `self` para llamar al método `cantar` haciendo referencia al objeto mismo.

En resumen:

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

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

```
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` sin un método asociado?

Analicemos el siguiente ejemplo:

```
class Fantasma
  def reflejar
    self
  end
end

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

En el ejemplo podemos corroborar que `self`, dentro del método `reflejar`, está retornando la instancia misma.

2. Self en un método de clase

¿Recuerdas que podemos utilizar `self` para definir métodos de clase?

También podemos utilizar `self` dentro de un método de clase para hacer referencia a la clase misma.

Observemos un ejemplo similar al anterior:

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

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

Similiar al comportamiento que ya conocemos, la instrucción `self` en un método de clase hace referencia a la clase misma.

En resumen:

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 el principio de polimorfismo
- Utilizar polimorfismo para evitar el uso innecesario de sentencias `if`

Introducción: ¿Qué es Polimorfismo?

En programación, los objetos de comunican a través de mensajes; en Ruby, este comportamiento se lleva a cabo a través del llamado a métodos. Es decir, la comunicación entre objetos es a través de métodos.

Poli-morfismo: Cualidad de tener muchas formas.

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

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

Primero, porque es uno de los principios fundamentales de la programación orientada a objetos; segundo, porque una de las formas en que podemos lograr polimorfismo es **a través de la herencia**.

Analicemos el siguiente ejemplo:

```
class Animal
end

class Perro < Animal
end

class Gato < Animal
end

class Vaca < Animal
end
```

¿Nada nuevo verdad?

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

A continuación, crearemos un método de instancia en nuestra superclase que permita a las subclases "hablar".

```
class Animal
  def hablar
  end
end

class Perro < Animal
end

class Gato < Animal
end

class Vaca < Animal
end
```

Pero... Perro, gato y vaca "hablan" de forma distinta ¿ O no?

La primera solución que viene a nuestras mentes es la de agregar sentencias `if` en la superclase y que la respuesta sea distinta según el tipo de subclase desde donde estoy llamando al método.

```
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

class Perro < Animal
end

class Gato < Animal
end

class Vaca < Animal
end

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

Es aquí donde debemos detenernos un segundo y analizar el código que estamos escribiendo: La cantidad de sentencias `if` genera sospecha.

A medida que vayamos *escalando* nuestra implementación, por ejemplo agregando más subclases que heredan de `Animal`, la enésima cantidad de sentencias `if` se volverá ilegible transformandose en un claro indicador de código mal escrito.

¿Podemos refactorizar este código?

Este es justamente el problema que viene a resolver el polimorfismo.

Principio de polimorfismo al rescate

"El principio de polimorfismo consiste en que dos objetos, de distina clase, pueden responder al mismo método de forma distinta."

```
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
```

Las intancias de `Perro` y `Vaca` responden al mismo método `hablar` de forma distinta. Hemos utilizado el principio de polimorfismo para que, a través de herencia, nuestro código sea ordenado y escalable.

En este ejemplo además del polimorfismo, podemos ver una aplicación del concepto de especialización mediante herencia. La clase mas generica **Animal** no especifica como debe "hablar" un animal, pero las clases hijas si lo hacen. De esta forma cada clase hija (Perro, Gato, Vaca) es encargada de definir su forma de actuar especializandose en su comportamiento y acciones.

Módulos y mixins

Objetivos:

- Conocer módulo como herramienta para la reutilización de código.
- Crear un módulo.
- Diferenciar un módulo de una clase.
- Incluir un módulo a una clase a través de mixin
- Diferenciar mixin de herencia

Introducción

Los módulos en Ruby se caracterizan por ser elementos que nos permiten **agrupar**. Imagina el módulo como un set de herramientas que, dependiendo del escenario, podemos disponibilizar y/o utilizar directamente.

Los módulos tienen 4 funciones principales:

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

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

Creando 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
```

1. Agrupando constantes

La primera función que estudiaremos de un módulo es la de agrupar constantes.

Imagina que nuestra aplicación relacionada con el mundo de las matemáticas requiere que, en diferentes escenarios y a través de diferentes entidades, pueda acceder a constantes matemáticas como **Pi** o **Euler**.

> No, no te asustes. No profundizaremos en constantes matemáticas.

Podemos definir un módulo llamado `MyMath` donde vamos a definir las constantes que utilizaremos en nuestra aplicación.

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

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

```
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:

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

2. Agrupando métodos

La segunda función que estudiaremos de un módulo es la de agrupar métodos. Para que los métodos pertenezcan al módulo utilizaremos la instrucción `self` en su definición. Luego podemos llamar a estós métodos utilizando `Modulo.método`:

```
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
```

El módulo anterior, **MyMath**, a primera vista es muy similar a una clase, pero existen diferencias.

- La principal es que no podemos crear instancias de un módulo a diferencia de lo que sucede con una clase.
- En las clases existe herencia, en los módulos no.
- El principal uso de las clases es crear objetos, en cambio los módulos nos ayudan a agregar comportamiento a las clases por medio de mixins.

3. Agrupando clases

La tercera función que estudiaremos de un módulo es la de agrupar clases.

```
module TiposDeUsuario
  class Usuario
  end

  class Conductor < Usuario
  end

  class Pasajero < Usuario
  end
end

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

¿Por qué necesitamos agrupar clases?

Para evitar **colisiones de nombres** ya que a veces necesitamos que dos clases posean el mismo nombre.

¿Puede una clase ser definida dos veces?

La respuesta es sí y sus contenidos serán mezclados. Dicho de otra forma, la última definición será agregada a la primera. Esto se conoce como extender una clase.

```
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

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

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

En programación, esta implementación se conoce como **monkey patching** y hay que poner cuidado en el motivo por el cuál lo estamos haciendo ya que puede considerarse una mala práctica.

4. Evitar colisiones de nombre

A través de módulos podemos solucionar las colisiones de nombre implementando un **namespace** (espacio de nombre). Los espacios de nombre nos permiten definir dos clases con el mismo nombre sin que sus definiciones se mezclen.

En Rails nos encontraremos con algo como esto:

```ruby
module ActiveRecord
  class Base
  end
end

module ActionView
  class Base
  end
end
```

Existe más de una clase llamada `Base`, sin embargo, podemos acceder a `ActiveRecord::Base` y `ActionView::Base` como clases que, a pesar que tienen el mismo nombre, son distintas. Ambas están definidas bajo un **namespace**.

Bonus: Anidación

Un módulo también puede contener otros módulos:

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

Money::Currency::Dolar
```

Resumen

Conocemos las principales funciones de un módulo. Sabemos que un módulo puede almacenar constantes, clases, métodos e incluso otros módulos. Para declarar métodos pertenecientes al módulo debemos hacerlo con la instrucción `self` (como si fuera un método de clase). Para acceder a los elementos contenidos en un módulo debemos utilizar la notación `::`.

Ahora, introduciremos un nuevo concepto que nos ayudará a entender el verdadero poder de los módulos: la opción de disponibilizar su contenido a una o más clases a través de **mixins**.

Mixins

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

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

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

class Gato
end

class Perro
  include Nadador
end

bobby = Perro.new
puts bobby.nadar # Puedo nadar!
```

Luego de utilizar la instrucción `include` en la clase `Perro`, los métodos definidos en el módulo `Nadador` quedan disponibles como **métodos de instancia** para la clase.

Para incluir todos los métodos de un módulo como métodos de clase utilizaremos `extend`:

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

class Gato
end

class Perro
  extend Nadador
end

Perro.nadar # Puedo nadar!
```

¿Recuerdas que mencionamos con anterioridad que Ruby no posee multiherencia?

La implementación de **mixins** es la solución a la multiherencia.

Una clase puede heredar sólo de una clase, pero podemos implementar, a través de mixins, tantos módulos como sea necesario.

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

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

class Mamifero
end

class Gato < Mamifero
  include Carnivoro
end

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

Resumen

- Un módulo lo definimos como un set de herramientas que podemos utilizar directamente o disponibilizar para su utilización en una clase o varias clases.
- La técnica que nos permite integrar módulos en una clase se denomina **mixin**.
- Podemos incluir, a través de mixins, tantos módulos 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 forma didactica los principios de herencia y polimorfismo
- Crear un video juego utilizando 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 tanto en Linux, Windows y OSX, 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
```

En las primeras líneas de código ya podemos observar lo aprendido. Nuestra clase `Tutorial` se define heredando de la clase `Window` del módulo `Gosu`. Luego, el constructor de la clase utiliza la instrucción `super` para llamar al constructor de la clase padre con los argumentos correspondientes a la resolución de la ventana.

El método `update` debe contener la lógica del juego y `draw` el código para redibujar la escena. Ambos métodos están sobrescribiendo los definidos en la clase `Gosu::Window`.

Finalmente se crea una nueva instancia de `Tutorial` y el método de instancia `show` para levantar la ventana del juego.

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
```

La variable de instancia `@background_image` corresponde a una nueva instancia de la clase `Image` contenida en el módulo `Gosu`.

> Para que esto funcione debe existir la carpeta `media` y el archivo space.png.

La imagen de fondo es dibujada en las coordenadas `0, 0, 0`.

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
```

El constructor de `Player` también genera una instancia de la clase imagen `Gosu::Image`.

> Para que esto funcione debe existir la carpeta `media` y el archivo starfighter.bmp.

Esta clase contiene los métodos necesarios para mover al jugador. Podemos identificar que el método `accelerate` utiliza los métodos del `offset_x` y `offset_y` del módulo `Gosu` para manejar la aceleración.

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
```

El método `update` maneja la lógica del movimiento. Las constantes `Gosu::KB_RIGHT`, `Gosu::KB_RIGHT` y `Gosu::KB_UP` Almacenan los códigos que identifican las teclas de nuestro teclado. A través del método `Gosu.button_down?` podemos condicionar la acción en función a la tecla presionada.

El método `button_down` pertenece a la clase `Gosu::Window`. La constante `Gosu::KB_ESCAPE` almacena el código correspondiente a la tecla Escape de nuestro teclado. Si el botón presionado es la tecla Escape, entonces cierra la aplicación.

> La implementación por defecto de este método permite utilizar comandos como alt+Enter(Linux, Windows) o cmd+F(MacOS). Para no sobrescribir este comportamiento utilizamos la instrucción `super` cuando la tecla presionada no es Escape.

Ahora podemos ejecutar nuestro juego que contiene un fondo de pantalla y una nave manipulable a través de las flechas de movimiento del teclado.

Para levantar la ventana de nuestro juego simplemente ejecutamos `ruby nombre_de_archivo.rb`. Nuestra terminal quedará secuestrada hasta que, dentro del juego presionemos la tecla Escape.

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? :)