Disciplina: Fundamentos básicos de Ruby
Professor Alexandre Calaça
Class 03: Classes, objects and methods

Class 03

Classes and Objects

A **class** is a **blueprint** for creating **objects** (instances). An **object** is an instance of a class, containing data and behavior (methods).

Let's say that an **object** is a class in action.

Understanding this is fundamental to Ruby, Rails, and object-oriented programming. Almost everything in Ruby is an object, including integers, strings, arrays—and even classes themselves!

🧠 Real-World Analogy:

Think of a **class** as a **blueprint for a house**, and the **object** as the actual house built from that blueprint.

- You can build many houses from one blueprint.
- Each house can have its own paint color or furniture (its **state**), but follows the same structure (its **class**).

Hands-on

In a Rails app, each model (like User, Post, Product) is a class, and each row in the database is an object (an instance of that class).

```
# class (defined in app/models/user.rb)

class User < ApplicationRecord
end
```

```ruby
# object (instance of the class)
user = User.new(name: "Alice")
```

Let's check another code snippet

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

  def bark
    "#{@name} says woof!"
  end
end

# Creating objects
dog1 = Dog.new("Buddy")
dog2 = Dog.new("Max")

puts dog1.bark  # => "Buddy says woof!"
puts dog2.bark  # => "Max says woof!"
```

Instance methods

An **instance method** is a method that you can call on an **object** (an instance of a class). These methods define the behavior of individual objects.

Instance methods allow each object to act independently, even if they're created from the same class. This makes code more **modular**, **reusable**, and **organized**.

🧠 **Real-World Analogy:**

> If a **Car class** is the blueprint, each **car object** (like a red car or blue car) can perform actions like `drive`, `brake`, or `honk`. These actions are instance methods—specific to each car.

Each car can drive at different speeds or in different directions—because the method operates on the individual object.

Hands-on

In a Rails model, methods that operate on a specific record (object) are instance methods.

```
class User < ApplicationRecord
  def full_name
    "#{first_name} #{last_name}"
  end
end

user = User.find(1)
user.full_name  # => "Alice Smith"
```

In the following code snippet, `read` is an instance method.

```
class Book
  def initialize(title)
    @title = title
  end

  def read
    "Reading #{@title}"
  end
end

book1 = Book.new("1984")
book2 = Book.new("Brave New World")

puts book1.read  # => "Reading 1984"
puts book2.read  # => "Reading Brave New World"
```

Class methods

Explanation:

A **class method** is a method that is called on the **class itself**, not on an instance of the class. It's defined using `self.` inside the class.

Class methods are useful for behavior that **doesn't depend on individual objects**, like utility functions, factory methods, or querying multiple records at once.

They help you **organize logic** that belongs to the class as a whole, not to a single instance.

🧠 Real-World Analogy:

> Imagine a **Factory class**.
> A factory can produce products without needing to be a product itself. You call the method on the factory, not on an individual item.

```
Factory.build_product  # makes a new product
```

You're not calling `.build_product` on an individual product, but on the blueprint/factory itself.

Hands-on

In Rails, class methods are often used for **scopes** or custom queries.

```
class User < ApplicationRecord
  def self.active
    where(active: true)
  end
end

User.active  # returns all active users
```

You don't need a `user` instance to call `.active`—you call it directly on the class.

In the following code snippet, we're creating a class method called `square` by using the class name before it.

```ruby
class MathUtils
  def MathUtils.square(x)
    x * x
  end
end

puts MathUtils.square(5)    # => 25
```

Another interesting alternative is the following notation, it's like you open the class `MathUtils` in order to change it, then you close it.

```ruby
class MathUtils
  class << self
    def square(x)
      x * x
    end
  end
end
```

Singleton methods

A **singleton method** is a method that is defined **only for a specific object**, not for other instances of the same class. It's a way to customize the behavior of one object without affecting others.

Singleton methods allow for **highly specific behavior**. They're powerful for metaprogramming, mocking in tests, or building DSLs. You can tweak one object without changing the class definition or affecting the other objects.

This is also how Ruby creates **class methods** under the hood: they're singleton methods on the class object!

🧠 **Real-World Analogy:**

> Think of two **smartphones** of the same model.
> You give one phone a unique ringtone that no other phone has.

Even though both are from the same class (`Phone`), one has a **unique behavior** (its own ringtone).

Hands-on

In Rails, you might see singleton methods used in:

- Initializers for setting specific behavior on a config object.
- Custom logger or third-party configuration.
- Defining one-off behavior in test doubles or factories.

In the following code snippet, `guest` is associated with one specific object.

```
user = User.new(name: "Guest")

def user.guest?
  true
end

user.guest?         # => true
User.new.guest?     # => NoMethodError (only the first object has
this method)
```

Another code snippet

```
class Animal
  def speak
    "generic animal sound"
  end
end

dog = Animal.new
cat = Animal.new

def dog.speak
  "Woof!"
end
```

```
puts dog.speak  # => "Woof!"
puts cat.speak  # => "generic animal sound"
```

- `dog` has a **singleton method** `speak`, which overrides the class version—but only for itself.
- `cat` still uses the regular `Animal#speak`.

Summary

A **class** is a blueprint. It defines how objects of that type behave.
An object is an instance of a class—it represents one specific thing built from that blueprint.

```
class User; end
user = User.new  # user is an object (instance of User)
```

Instance Methods are methods defined in a class and used by its instances (objects).
They represent actions that each individual object can perform.

```
class User
  def name
    "Alice"
  end
end

user = User.new
user.name  # => "Alice"
```

Class Methods are methods you call on the class itself, not on an object.
They are used for behaviors that relate to the class as a whole.

```
class User
  def self.count
    42
  end
```

```ruby
end

User.count  # => 42
```

Singleton Method is defined for one specific object.
It doesn't affect any other object or even other instances of the same class.

```ruby
user = User.new

def user.admin?
  true
end

user.admin?  # => true
```

Only this `user` object can call `.admin?`; other `User` objects can't.

Topic	Called on	Affects	Example
Instance Methods	object	instances	`user.name`
Class Methods	class	class only	`User.find(1)`
Singleton Methods	specific object	one object	`def user.guest?`
Classes and Objects	—	core concept	`User.new` creates an object
