Programming for Everybody

9. Classes & Instances

le wagon

Classes and instances

Ruby has some built in classes you already know: String, Integer, Array, Hash, etc.

A **Ruby** ***Class*** is like a “cake mold”, from which several ***instances*** can be originated -> because they come from the same “mold”, all of these ***instances*** share similar methods and their respective **attributes**

Each instance of a ***Class*** is a **Ruby** ***object***

`“John”` ← **this object is an instance of the String class**

`[1,2,3,4]` ← **this object is an instance of the Array class**

`12` ← **this object is an instance of the Integer class**

Building our own classes

We can also create new Classes from scratch

Class syntax: **class** keyword + **class name** + **end** keyword

Within this, we include the **.initialize** method, which "boots up" each object created by the Class, and which includes its **instance variables** (these set the new objects' specificities)

```
class Car
  def initialize(make, model)
    @make = make
    @model = model
  end
end

Car.new("Honda", "Civic");
```

this Class will allows us to create as many Car instances as we want

each Car object will have its own make and model

we can create an instance of a Class just by calling .new on the Class name, and defining values for the instance variables

Instance methods

We often define other methods for our ***Classes*** so that their instances can do interesting stuff

While **instance variables** define an object's attributes, **methods** define its ***behaviour***

```
class Person
  def initialize(name)
    @name = name
  end

  def greeting
    puts "Hi!"
  end
end
```

```
solene = Person.new("Solene")

solene.greeting

# prints out: Hi!
```

Scope

An important aspect of Ruby ***Classes*** is their *scope* -> the context in which they're available

global variables are available everywhere and can be declared in two ways:
- defined outside of any method or class
- preceded by an $ if we want them to become global from inside a method or class (ex: `$foo`)

local variables are only available inside certain methods

Scope (cont.)

class variables belong to a certain *Class*, are preceded by two `@`s (ex: `@@files`) and there's only one copy of a *Class* variable which is then shared by all instances of that *Class*

instance variables are only available to particular instances of a *Class,* and are preceded by an `@`

Global variables can be changed from anywhere in the program and it's better to create variables with limited scope that can only be changed from a few places (ex: **instance variables** which belong to a particular object)

Scope (cont.)

The same goes for **methods**

global methods are available everywhere

class methods are only available to members of a certain Class

instance methods are only available to particular instances

Inheritance syntax

inheritance is the process by which one Class takes on the attributes and methods of another

the derived Class (or *subclass*) is the new Class we're creating

the base Class (or *parent* or *superclass*) is the Class from which the derived Class inherits

inheritance syntax:

```
class DerivedClass < BaseClass
  # some stuff
end
```

we read "<" as "inherits from"

Overriding inheritance

Sometimes we may want one Class that inherits from another to **override** certain methods of their parent

```
class Creature
  def initialize(name)
    @name = name
  end

  def skin_color
    puts “Green"
  end
end
```

```
class Dragon < Creature
  def skin_color
    puts “Purple"
  end
end
```

Class Dragon has inherited its parent’s Class instance variables but has overridden its .skin_color method

```
bob = Dragon.new("bob")
bob.skin_color

# prints out: Purple
```

Inheritance with super

We can directly access the attributes or methods of a parent Class with Ruby's built-in **super** keyword

```
class DerivedClass < ParentClass
  def some_method
    super(optional_args)
      # Some stuff
    end
  end
end
```

when we call super from inside a method, we're telling Ruby to look in the parent Class of the current Class and find a method with the same name as the one from which super is called

if it finds it, Ruby will use the parent class' version of the method

Thank you.