←Prev Submission Next→

8 Ruby: Intro to Classes 2

Initializers

Getter and setter methods can be used for accessing the attributes of a class, but you'll often want to create a new instance of a class with default attribute values. Getter and setter methods can only access attributes **after** an instance of a class has been created. Ruby has a standard method named `initialize` that is executed when you create a `new` class instance. Let's declare a new class named `President` and then define an `initialize` method:

```
class President
  def initialize
    p "Hello, I'm an instance of the President class!"
  end
end

President.new

#=> "Hello, I'm an instance of the President class!"
```

Let's add an `attr_accessor` declaration, so that we can access an attribute when we `initialize`:

> Remember, `attr_accessor` dynamically creates getter and setter methods for the given symbol (e.g. :age).

```
class President
  attr_accessor :age

  def initialize
    @age = 55
  end
end

pres = President.new
p pres.age

#=> 55
```

In the example above we set an **instance variable** called `@age` to `55` in our `initialize` method. Because we declared an `attr_accessor` for an attribute named `age` (which automatically defines both `age=()` and `age` methods), we can call the `age` getter method on the `pres` instance of the `President` class.

This is an awfully rigid way of setting attributes, so let's make the attribute values more flexible. That is, let's allow the `age` value to be set when a `new` instance of `President` is created:

```
class President
  attr_accessor :age

  def initialize(years)
    @age = years
  end
end

pres = President.new(49)
p pres.age

#=> 49
```

Now we are essentially creating an interface for our `President` class. We have allowed the `age` attribute to be set by a theoretical user, rather than "hard-coding" the value as we did in the prior example. We defined the `initialize` method to take an argument (e.g. `years`), and when we pass a value to the `new` method (e.g. `49`), the `initialize` method recognizes it, and assigns it to the `@age` instance variable.

Let's try one more example, only this time we'll access two attributes:

```
class President
  attr_accessor :age, :party

  def initialize(years, party)
    @age, @party = years, party
  end
end

pres = President.new(49, "Independent")
p "The President is #{pres.age} years old and is a member of the #{pres.party} party

#=> "The President is 49 years old and is a member of the Independent party."
```

> We used some fancy string interpolation to display our attributes. You'll use string interpolation **a lot** as a Rails developer, so get used to it!

Class Methods

The methods we have written in classes up to this point are **instance methods**. That is, they are called on a unique instance of a class. There's another type of method that can be written in a class. **Class methods** pertain to the class itself, rather than a unique instance of the class. Let's modify our `President` class to demonstrate this. We'll make our class slightly more specific and rename it `UnitedStatesPresident`:

```
class UnitedStatesPresident
  attr_accessor :age, :party

  def initialize(years, party)
    @age, @party = years, party
  end

  def self.citizenship
    "United States of America"
  end
end

p UnitedStatesPresident.citizenship
#=> "United States of America"
```

> When defining class methods, you must put `self.` in front of the method so that Ruby recognizes it as a class method.

To fully grasp the difference between class and instance methods, let's talk through the `citizenship` example in plain English. Our class, which is named `UnitedStatesPresident` could possibly share some attributes with other President-based classes. For example, we could also have `GermanyPresident` and `FrancePresident` classes as well. Some attributes would be unique to specific instances of these classes, like `age`, `party` and `name`.

But the `citizenship` of a country's President applies to the class itself. *All* instances of `UnitedStatesPresident` have citizenship in the United States of America. In other words, you could say that `citizenship` applies to any and all instances of one these aforementioned classes. For example:

```
class UnitedStatesPresident
  attr_accessor :age, :party

  def initialize(years, party)
    @age, @party = years, party
  end

  def self.citizenship
    "United States of America"
  end
end

class GermanyPresident
  attr_accessor :age, :party

  def initialize(years, party)
    @age, @party = years, party
  end

  def self.citizenship
    "Germany"
  end
end

class FrancePresident
  attr_accessor :age, :party

  def initialize(years, party)
    @age, @party = years, party
  end

  def self.citizenship
    "France"
  end
end
```

You could also say that whichever particular `UnitedStatesPresident` we're talking about, how old they are, or what party they're affiliated with, they'll always have the same `citizenship`. That's the law, after all!

More About Classes and Objects

We've only covered the basics of Classes in this checkpoint. Read our **About Objects resource** for broader context and history.

Indentation

Before trying the exercises, read this erratically indented version of our Presidents code file:

```
class UnitedStatesPresident
  attr_accessor :age, :party

  def initialize(years, party)
@age, @party = years, party
end

  def self.citizenship
    "United States of America"
  end
  end

class GermanyPresident
  attr_accessor :age, :party

      def initialize(years, party)
        @age, @party = years, party
end

    def self.citizenship
      "Germany"
    end

  class FrancePresident
  attr_accessor :age, :party

  def initialize(years, party)
      @age, @party = years, party
      def self.citizenship
      "France"
  end
  end
```

Following the code is much harder.

> We have one or more `end`s too few or too many in this version — can you find the problem(s)?

As you try the following exercises, indent consistently. As your logic becomes more complicated, indentation will become more important.

Exercises

✓ ~~Initialize~~

✓ ~~Class Methods~~

✓ ~~Calculator Class~~

Solution Walk-Throughs

Do not watch these videos until after you've attempted your own solutions. If you struggle to complete these exercises, submit your best effort to your mentor *before watching a solution walk-through*. Submit your answers even if some tests don't pass, as it's important for your mentor to understand your thought process. Discuss difficult exercises with your mentor.

There are many possible solutions to the exercises. A walk-through is meant to provide insight into the thought process of *one possible solution* per exercise. If your solution is different yet passes all the tests, do not change your solution to match what is seen in the video. Instead, discuss your thought process with your mentor.

1. **Initialize Solution**
2. **Class Methods Solution**
3. **Calculator Solution**

assignment completed