# 7 Ruby: Intro to Classes 1

## The Blueprint

Ruby is called an *object-oriented* (OO) programming language. In an OO language, objects communicate with each other via messages. That is, objects have defined behavior, and when we want an object to perform one of its associated actions, we send it a "message" asking it to do so. Here's an example:

```
numbers = [1,2,3]
numbers.length
#=> 3
```

In the code above, we're declaring a variable named `numbers` and setting it equal to an array consisting of 1, 2 and 3. `numbers` is an object. On the next line, we send `numbers` a message: `length`. This message is a question to which the object responds, if it knows how.

> A "method" is the set of instructions on an object called in response to a message this object receives. The method then "responds" with its return value.

Consider real-world objects. Dogs, for instance, can be considered objects which respond to `bark` and `sit`, but not `converse`. (Well, *most* dogs.)

Where are an object's "methods" stored? That's where classes come in. In our first example above, the `numbers` object came from the `Array` class. Classes are blueprints that describe objects, objects are created from classes.

Consider the following arrays:

```
numbers = [1,2,3]
names   = ["Arya", "Sansa"]
```

The `numbers` and `names` arrays are different objects, but they are created from the same blueprint, `Array`. While they have different contents, they share many of the same abilities. For example, they can both respond to a `length` message:

```
numbers = [1,2,3]
numbers.length
#=> 3

names   = ["Arya", "Sansa"]
names.length
#=> 2
```

## Class Instances

The `numbers` and `names` arrays are objects, and they are also **instances** of the `Array` class. In this section, let's focus on the concept of **instances**, rather than the more generic term of object.

Here is a simple example of a class named `Person`. Let's define a `Person` class and then create an instance of it named `arya_stark`:

```
class Person
end

arya_stark = Person.new
```

> Class names in Ruby always start with a capital letter and are **CamelCased**.

An instance is a unique manifestation of a class. For example, `arya_stark` is a type of `Person`, and `arya_stark` is therefore unique. There are other people (i.e. instances of `Person`) that could be created from the `Person` class, but there's only one `arya_stark`.

That's not to say there couldn't be another instance of `Person` with the same name, just as there are surely multiple people named Arya Stark in the world. The important thing is that each instance is unique. For example:

```
class Person
end

arya_stark = Person.new
p arya_stark
#=> #<Person:0x104375690>

arya_stark = Person.new
p arya_stark
#=> #<Person:0x104375618>
```

> The object ID (`#<Person:0x....`) is different each time we create a new `Person` using `Person.new`. That's because each instance of the `Person` class is unique.

## Instance Methods

When we have an instance of the `Person` class, we can call **instance methods** on this instance. For example, `eat` and `sleep` might be methods we could call on a `Person` instance. Let's define an instance method on our `Person` class, and call it on an individual person instance:

```
class Person
  def eat
    "nom nom nom"
  end
end

arya_stark = Person.new
arya_stark.eat
#=> "nom nom nom"

sansa_stark = Person.new
sansa_stark.eat
#=> "nom nom nom"
```

When an instance of `Person` receives the `eat` message, the `eat` *method* returns the answer. Because the `eat` method is simple, its instructions are the same regardless of the instance. That is, it instructs every instance that receives it to say, "nom nom nom."

## Attributes and Instance Variables

An instance of the `Person` class can do things like `eat` and `sleep` using instance methods, but it can also have attributes. For example, an instance of the `Person` class might have a `name`. The `name` of a person would therefore be an attribute of a `Person` instance. Consider the following example:

```ruby
class Person
  def name=(full_name)
    @full_name = full_name
  end

  def name
    @full_name
  end
end



first_lady = Person.new
first_lady.name=("Michelle Obama")
p "My name is #{first_lady.name}"
#=> "My name is Michelle Obama"

presidential_candidate = Person.new
presidential_candidate.name=("Hillary Clinton")
p "My name is #{presidential_candidate.name}"
#=> "My name is Hillary Clinton"
```

There's a lot happening above, so let's explore each step. First, we're declaring the `Person` class. The first method we define is named `name=`:

```ruby
class Person
  def name=(full_name)
    @full_name = full_name
  end
  #...
```

The `name=` method is called a **setter method**. Its sole purpose is to **set** an attribute. The attribute we're setting is the name of the person, and it's stored using an **instance variable**. We named the instance variable `@full_name` and set it to the `full_name` argument being passed to the `name=` method. In English we could say, "the name method sets an instance variable called full name."

We use `=` in the method name. This is not strictly necessary, but it is a common Ruby convention for setter methods. It makes the method more idiomatic. For example:

```ruby
name=("Michelle")

# or without parentheses

name= "Michelle"

# Ruby even allows us to add a space before =

name = "Michelle"
```

> In all three cases, the method is `name=`, and `=` is part of that name: we are not assigning a value to a local variable `name`, we are calling the method `name=`. The `=` character is *not* what makes that method a setter: it's only a convention to notify fellow developers that the method is concerned with setting an attribute.

Unlike local variables, instance variables start with `@`. By making `@full_name` an instance variable, we allow it to be used **outside** of the method where it's declared. We use the term "instance variable", because the variable is associated with and available throughout the class instance it's defined in. This is an important concept called the variable's scope. If we didn't make `@full_name` an instance variable, it would not be accessible outside of the `name=` method. Let's review the next method to solidify this concept.

```ruby
  #...
  def name
    @full_name
  end
end
```

The method above is called `name`. This is called a **getter method** because its sole purpose is to retrieve an attribute. We're returning `@full_name`. We can do this because `@full_name` is an instance variable. That is, it's declared in the `name=` method, but we can reference it in other methods within the `Person` class.

# Stack Overflow

**Stack Overflow** is a programmer's best friend. The question-and-answer site is manned by a huge number of international volunteers who can answer basic and complex technical questions with often incredible speed and accuracy. As you're learning with Bloc - and long after - Stack Overflow will be an invaluable resource. *It is well worth becoming comfortable with.* Read **Bloc's resource on using Stack Overflow**. Then:

1. Using **repl.it**, define a class with a single method by copying and pasting the following text:

   ```
   class ArgumentEchoer
    def echo_argument(argument)
      p "You just inputted a value: " + argument
    end
   end
   ```

   Then call that method:

   ```
   instance = ArgumentEchoer.new
   instance.echo_argument(5)
   ```

   You should run into an error.

2. In the Discussion tab for this checkpoint, draft a message to your mentor. Following the guidelines in the resource above, draft the question you'd ask on Stack Overflow. Your question should be clear, concise, and informative.

   - What caused the error?
   - What is your knowledge level?
   - What were you trying to accomplish?
   - What is the complete error message?
3. When your fake question looks complete, stare at the error for a second. What do you think it means? How do you think you might get around this problem?

4. Open Google in a separate tab and search for the error message you likely received (`TypeError: no implicit conversion of Fixnum into String` - but if you received something else, search for that!). After reading through an example or two, what do you think went wrong? How might you fix it?

> Most Google results for common error message searches point to Stack Overflow. Searching for error messages often makes asking your question unnecessary.

5. At the bottom of your fake-question message to your mentor, submit a hypothesis about how to fix this error. Then send the message off.

When you've completed the above, try the exercises to solidify your understanding of Class basics.

| Assignment | Discussion | Submission |
|---|---|---|

## Exercises

✓ ~~RSpec Class Errors~~

✓ ~~Basics~~

✓ ~~Getters and Setters~~

✓ ~~Attr Accessor~~

## 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. **RSpec Class Errors Solution**
2. **Basics Solution**
3. **Getters and Setters Solution**
4. **Attr Accessor Solution**

assignment completed

?