11 Ruby: Advanced Classes

To get a more complete picture of how classes fit into Ruby's object-oriented approach, let's discuss two weighty-sounding subjects: **Self** and **Inheritance**.

Self

Every object in Ruby can refer to itself. This might seem esoteric, but it's essential to how objects work. Take this `Person` class for example:

```
class Person
  attr_accessor :first, :last

  def full_name
    "#{@first} #{@last}"
  end

  def introduce_yourself
    # How do I do this without repeating the above logic here?
  end
end
```

In Ruby, instances refer to themselves using the `self` keyword:

```ruby
class Person
  attr_accessor :first, :last

  def full_name
    "#{@first} #{@last}"
  end

  def introduce_yourself
    p "Hello, my name is #{self.full_name}."
  end
end
```

> Within method bodies, `self` is implicit, and can be left out. If you were to write just `full_name` within the `introduce_yourself` method above, Ruby would have guessed that you meant `self.full_name`. This rule has exceptions which we'll explore later.

We also use `self` when defining class methods:

```ruby
class UnitedStatesPresident
  def self.citizenship
    "United States of America"
  end
end
```

This is because *classes are themselves objects*. When we type `def self.citizenship`, we are defining the `citizenship` method on the `UnitedStatesPresident` class object itself!

> Within a class method, `self` references the class object.

This may be confusing. Let's explore inheritance to understand this more deeply.

Inheritance

In object-oriented programming, inheritance refers to a specific relationship between objects where one takes functionality from another. In "classical inheritance" (so called because it's class-based, not old), one class inherits attributes and behavior (i.e. methods) from another class. The class which inherits is known as a child class, while the class it inherits from is called a parent class. Inheritance is designated with a `<` symbol. For example:

```ruby
class Elm < Tree
end
```

In the example above, we are defining a class named `Elm` that inherits from a class named `Tree`. Because we declared the inheritance in the class definition, all instances of the `Elm` class automatically have attributes and methods defined in the `Tree` class. Consider the following:

```ruby
class Tree
  def roots?
    true
  end
end

class Elm < Tree
end

e = Elm.new
e.roots?
#=> true
```

We called the `roots?` method on an instance of `Elm` even though the `roots?` method is not explicitly defined in the `Elm` class. We were able to call `roots?` because `Elm` inherits from `Tree`.

Inheritance: Using Super

Let's explore an inheritance example where a parent class has multiple child and grandchild classes. We'll start by defining an `Animal` class that will be used to store generic information for all animals.

```ruby
class Animal
  attr_accessor :name

  def initialize(name)
    @name = name
  end

  def eat(other)
    puts "#{@name} ate #{other.name}! #{self.noise}"
  end
end
```

> We did not define `noise` above, but the descendants of the `Animal` class will.

Let's write a child class and have it inherit the `name` attribute from `Animal`, but also take another attribute on initialization:

```ruby
class Human < Animal
  attr_accessor :catchphrase

  def initialize(name, catchphrase)
    super(name)
    @catchphrase = catchphrase
  end

  def noise
    @catchphrase
  end
end
```

Ruby's `super` method lets us call a method in the parent class from within the child class. In the above example, `super(name)` in `Human`'s `initialize` method calls the `initialize` method in `Animal` which executes `@name = name`. When inherited methods are more complicated, however, calling `super` can import much more nuanced behavior.

> If a child defines a method which its parent also defines, the child method will *override* the parent's. `Animal.initialize()` is, by convention, supposed to guarantee that name is set. Removing the call to `super` would break this promise, and would therefore be a bug.

Let's test our inheritance:

```
a = Human.new("Adam", "Right on!")
b = Animal.new("Chicken")
a.eat(b)
#=> "Adam ate Chicken! Right on!"
```

The `Human` class is slightly more specific than the `Animal` class, but still generic. Let's define a child class of `Human` (which is a grandchild class of `Animal`):

```
class Englishman < Human
  def initialize(name = "Mick Jagger")
    super(name, "I can't get no....")
  end
end
```

> If the above `Englishman#initialize` method still does not make sense, explain it step by step to a mentor, family member, or **inanimate object**.

Let's test our defined inheritance again:

```
a = Human.new("Adam", "Right on!")
mick = Englishman.new
mick.eat(a)
#=> "Mick Jagger ate Adam! I can't get no...."
```

The `Human` class is a child of `Animal`. Let's define a sibling, which would be another child of the `Animal` class:

```ruby
class Dog < Animal
  attr_accessor :excitement_level

  def initialize(name, excitement_level)
    super(name)
    @excitement_level = excitement_level
  end

  def noise
    "woof" * @excitement_level
  end
end
```

Test the new `Dog` class:

```ruby
d = Dog.new("Eight", 3)
d.eat(d)
#=> "Eight ate Eight! woofwoofwoof"
```


Exercises

- ~~Self~~
- ~~Good Parents~~

- [x] ~~Super~~
- [x] ~~Modules~~
- [x] ~~More Complicated Inheritance~~

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.

- **Self**
- **Good Parents**
- **Super**
- **More Complicated Inheritance**

assignment completed

?