Programming for Everybody

10. Private methods, Accessors & Modules

Private and public class methods

ruby methods define the behaviour of classes objects / instances

by default, methods are **public**, meaning they can be accessed by all objects of a certain class

however, it may be useful to define some methods as **private**, to prevent users from manipulating your objects in a way which is not the one you had intended

Public methods

public methods can be called by the instances of that class or subclasses

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

    def speak
        “Meow!”
    end
end

cat = Animal.new(“Garfield")
puts cat.speak

prints out “Meow!”
```

the “cat” instance managed to access the “speak method from within the “Animal” class definition scope

Private methods

private methods are preceded by the word *private*; they cannot be called directly by an object -> the only way to have access to a private method is to call it within a public method

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

    private
    def speak
        “Meow!”
    end
end

cat = Animal.new(“Garfield")
puts cat.speak
```

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

    def access_speak
        speak
    end

    private
    def speak
        “Meow!”
    end
end

cat = Animal.new(“Garfield")
puts cat.access_speak
```

Accessing attributes

if we want to access the **instance variables** of a class we can do it in two ways

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

  def animal_name
    @name
  end
end

cat = Animal.new(“Garfield")
puts cat.animal_name

prints out “Garfield”
```

SAME AS

```
class Animal
  attr_reader :name

  def initialize(name)
    @name = name
  end
end

cat = Animal.new(“Garfield")
puts cat.name

prints out “Garfield”
```

passing our instance variables as symbols to an **attr_reader** is the more “Rubyist” way of making them available to be read

Accessing attributes (cont.)

attr_reader we use this shortcut to **read** the value of an instance classes; it's called a *getter*

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

  def animal_name
    @name
  end
end

cat = Animal.new("Garfield")
puts cat.animal_name

prints out "Garfield"
```

SAME AS

```
class Animal
  attr_reader :name

  def initialize(name)
    @name = name
  end
end

cat = Animal.new("Garfield")
puts cat.name

prints out "Garfield"
```

passing our instance variables as **symbols** to an **attr_reader** is the more "Rubyist" way of making them available to be read

Accessing attributes (cont.)

attr_writer we use this shortcut to **change** the value of an instance classes; it's called a *setter*

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

    def name=(value)
        @name = value
    end
end

cat = Animal.new("Garfield")

puts cat.name = "Kitty"

prints out "Kitty"
```

SAME AS

```
class Animal
    attr_writer :name
    def initialize(name)
        @name = name
    end
end

cat = Animal.new("Garfield")

puts cat.name = "Kitty"

prints out "Kitty"
```

Accessing attributes (cont.)

attr_accessor is a shortcut that allows us to both **read** and **change** the value of an instance classes at once; it's thus a *getter* and a *setter* simultaneously

```
class Animal
    attr_accessor :name
    def initialize(name)
        @name = name
    end
end

cat = Animal.new("Garfield")
puts cat.name
prints out "Garfield"

puts cat.name = "Kitty"
prints out "Kitty"
```

Modules

modules store methods that can then be used by different classes, allowing us to keep our code DRY

like classes, modules also hold methods but they can’t be instantiated -> we can’t create objects from a module

modules are useful if we have methods that we want to reuse in different classes while keeping them in a central place to avoid repeating them everywhere

Modules (cont.)

ruby has some built in modules (ex: date) we can use by typing the **require** keyword + their name

but we can also build our own modules -> the syntax is similar to that of a class, only modules don't include variables since variables, by definition, change and a module is supposed to be immutable

```
module Cream
  def cream?
    true
  end
end
```

A module can be ***mixed*** into different classes two ways:

1) at *instance level* (through the **include** keyword)
2) at *class level* (through the **extend** keyword)

Modules (cont.)

mixing a module at instance level

```
module Cream
    def has_cream?
        true
    end
end

class Cookie
    include Cream
end

cookie = Cookie.new
p cookie.has_cream?
(prints out true)

class Cake
    include Cream
end

cake = Cake.new
p cake.has_cream?
(prints out true)
```

mixing a module at class level

```
module ID
    def item_category(category)
      "Congrats! You've just created a new \"#{category}\" category!"
    end
end

class Cocktail
    extend ID
end

puts Cocktail.item_category("Cocktail")


class Cake
    extend ID
end

puts Cake.item_category("Cake")
```

Thank you! :)

le wagon