CodeCoreYVR / **bootcamp_summary_notes** Private

Branch: **master** **bootcamp_summary_notes** / week_02 / **ruby_classes_and_objects.md** Find file Copy path

tkbeili fix up week1 and week2 files 34504a1 on 8 Jan

1 contributor

279 lines (242 sloc) 7.13 KB

Ruby Classes & Objects Summary Notes

Ruby is a truly an object oriented programming language as everything in Ruby is an object. Object and Classes are core parts of most of Ruby programs.

Creating a class

To create a class in Ruby use the `class` keyword as in:

```
class Cookie

end
```

Class names must start with a capital letter as they are treated as `Constants`. The convention is to use camel case for naming classes. Camel case means omitting spaces in the name and capitalizating the first letter of every work. For every if you wanted to use British Columbia as a class name you would use `BritishColumbia`.

Creating Objects

To create an object you call the `new` method on the class as in:

```
c = Cookie.new
```

Methods

To make best use of classes, you will need to create methods for the classes. Let's define our first method:

```
class Cookie
  def eat
    "Yummy!"
  end
end
```

The method `eat` above will be a public instance method. This means you can call it from outside the class on any instance (object) of the class:

```
c = Cookie.new
c.eat # returns: Yummy!
```

Public vs. Private methods

The method we create above `eat` is a public method which means you can call from outside the class. You can create a `private` method that can only be called from methods within the same class.

```
class Cookie
  def bake_n_eat
    bake
    eat
  end

  def eat
    "Nom. Nom. Nom!"
  end

  private

  def bake
    "baking the cookie"
  end
end

c = Cookie.new
c.bake_n_eat
c.eat
c.bake # will give error
```

Class Methods

Sometime you may wish to create methods that don't require objects to be created first. You can do so using the `self.` as in:

```
class Cookie
  def self.info
    "I'm the Cookie class!"
  end
end

Cookie.info
# returns: I'm the Cookie class!


c = Cookie.new
c.info # gives error
```

You can create class methods as public or private.

The Initialize Method

You can define a special method for Ruby classed called `initialize` . This method will be called when you create an object:

```
class Cookie
  def initialize
    puts "in the initialize method"
  end
end

c = Cookie.new # prints:in the initialize method
```

Instance Varibles

If you'd like to define variable in a Ruby method that can be accessed and changed from other methods later on then you can use instance variables which are defined by prefixing the variable name with `@`. Instance variables persist throughout the life of the object. Note that by default accessing an undefined instance variable gives you `nil` instead of raising an exception. Here is an example of using instance variables:

```
class Cookie
  def initialize(sugar, flour)
    @sugar = sugar
    @flour = flour
  end

  def details
    "sugar: #{@sugar} / flour: #{@flour}"
  end
end

c = Cookie.new(10,15)
c.details # prints: sugar: 10 / flour: 15
```

Class Variables

You can define a variable that is shared between different instances of the same class by creating a class varibles which is defined by prefxing the variable name with `@@`:

```
class Cookie
  def initialize
    # class variable can be accessed and changed
    # by all objects of Cookie class and subclasses
    @@color = "Brown"
  end
end
```

In the example above an instance of the `Cookie` class and subclasses can access and modify the `@@color` variable. Note that it's advisable that you avoid using class variable as much as possible as they share state between all object which may cause unpreditable consquences as your code grows.

Attributes

In many cases you may need to access an instance variable so you can create what is called a `getter` method:

```
class Cookie
  def sugar_amount
    @sugar_amount
  end
end
```

You may also want to alter an instance variable so you can create a `setter` method:

```
class Cookie
  def sugar_amount=(new_amount)
    @sugar_amount = new_amount
  end
end
```

Alternatively you can use `attr_reader` which defines a getter method, `attr_writer` which defines a setter method or `attr_accessor` which defines both:

```
class Cookie
  attr_reader :sugar_amount
```

```
    # def sugar_amount
    #   @sugar_amount
    # end

    attr_writer :sugar_amount
    # def sugar_amount=(new_amount)
    #   @sugar_amount = new_amount
    # end

    attr_accessor :flour_amount
    # attr_reader :flour_amount
    # attr_writer :flour_amount
  end

  c = Cookie.new
  c.sugar_amount = 10
  puts c.sugar_amount
```

Interitance

Inheritance is a powerful way to structure and reuse code in object oriented programming. If you have class that shares a lot of functionalty of another class or extends the other class with more methods and attributes then inheritance maybe a good solution. You can make the class inherit from another using teh `<` operator:

```
class Cookie
  attr_accessor :sugar_amount
  attr_accessor :flour_amount

  def eat
    "Yummy!"
  end
end

class Oreo < Cookie
  attr_accessor :filling_type
end
```

The `Oreo` class is called the child class or subclass, the `Cookie` class is called the super class or parent class. The Oreo class inherits all the methods and attributes from the Cookie class and can add its own unique methods and attributes:

```
c = Oreo.new
c.sugar_amount = 10
c.filling_type = "Chocolate"
c.eat
```

Note that you can only inherit from a single class in Ruby.

Method Override

You can override a method (redefine it) in the `Oreo` class. You can use the `super` keyword to call the one from the parent class:

```
class Cookie
  def eat
    "Yummy!"
  end
end

class Oreo < Cookie
  def eat
    puts "in Oreo class"
    super
  end
```

```
end

c = Oreo.new
c.eat
```

Modules

Modules has two primany users: as a namespace and as a mixin.

Modules as Namespaces

If you have two classes that have the same name but different purpose, you can avoid redefining the class by using a module as in:

```
module Computer
  class Apple
  end
end

module Fruit
  class Apple
  end
end

mac  = Computer::Apple.new
gala = Fruit::Apple.new
```

This way we defined two `Apple` classes without name collision. You can also write the code above as:

```
class Computer::Apple

end

class Fruit::Apple

end

mac  = Computer::Apple.new
gala = Fruit::Apple.new
```

Modules as Mixins

If you have methods or variables that are shared with difference classes you can use modules to inject those methods into other class as in:

```
module HelperMethods
  def name_display
    name.squeeze(" ").capitalize
  end
end

class User
  attr_accessor :name
  include HelperMethods
end

class Car
  attr_accessor :name
  include HelperMethods
end

u      = User.new
u.name = "tam"
```

```ruby
puts u.name_display
c      = Car.new
c.name = "toyota"
puts c.name_display
```

This is useful if the classes don't necessarily share a lot of functionality to use inheritance.

Include vs. Extend

You can inject the methods from a module into a class using either `include` or `extend`. The `include` adds the methods from the module into the class as instance methods while `extend` add the methods as class methods:

```ruby
module HelperMethods
  def greeting(name)
    "Hello #{name}"
  end
end

class Class1
  include HelperMethods
end

class Class2
  extend HelperMethods
end

c = Class1.new
c.greeting("John")

Class2.greeting("John")
```