Full Stack Notes

Object Oriented Ruby

Object-oriented programming (OOP) is a programming paradigm that uses "objects" and their interactions to design applications and computer programs. Object-oriented programming languages may include features such as encapsulation, modularity, polymorphism, and inheritance.

Objects are the nouns. Methods are the verbs. They are the stuff ~~dreams~~ programs are made of.

Table of Contents

Modelling the Real World

When you write object-oriented code, you're normally looking to model concepts from the real world. (Sometime this is easier said than done!)

In Ruby you will model these concepts by defining *classes*. A class represents a combination of state and methods that operate within (or use) this state. We can create an instance of a class in order to put it to use. We call these instances *objects*.

RESOURCES

- Classes, Objects, and Variables from the Pragmatic Guide
- Programming + Philosophy = Good Times

Class Creation

Let's create an empty class to model *students*:

```
class Student < Object
end
```

Note: Class names are capitalized.

Note 2: The student class is empty. This is not to imply that students are somehow empty, hollow, or devoid of meaning. ;)

The "< Object" indicates that our class inherits from (is a child of) the Object class. Our student class could be re-written as:

```
class Student
end
```

All classes inherit from the Object class when no other inheritance is specified.

Class Methods and Instantiation

Class methods are defined in similar fashion to regular methods:

```
class Student
  def write_code
    puts "All this hacking is making me thirsty."
  end
end

a_student = Student.new # Check it out: Instantiation
a_student.write_code
```

Output:

```
All this hacking is making me thirsty.
```

Instance Variables

Instance variables are used to save the state of an object. They are prefixed with an @ symbol. Instance variables are “private” members by default.

```ruby
class Zombie
  def greetings
    @name = "Wally Glutton"
    puts "#{@name} want Brainz!"
  end
  def name # A 'getter' method.
    @name  # Returns the value of this instance variable.
  end
end

walter = Zombie.new
walter.greetings
puts walter.name
```

Output:

```
Wally Glutton want Brainz!
Wally Glutton
```

Note that you do not need to predefine an instance variable before it is used. Unless you are careful this can cause some unexpected behaviour.

What happens if you try to “puts walter.name” before you call “walter.greetings”?

The Initialize Method

Let’s add a constructor to our class:

```
class Student
  # All constructors are named 'initialize'.
  def initialize(name, student_number)
    # Save the arguments as an instance variables.
    @name = name
    @student_number = student_number
    puts "#{@name} at your service."
  end
end

wally_glutton = Student.new("Wally Glutton",8273633)
```

Output:

```
Wally Glutton at your service.
```

Want to see something wild:

```
class Hal
  def sing
    puts "Daisy, Daisy, over the ocean blue..."
  end
end

print "hal_one sez: "
hal_one = Hal.new
hal_one.sing

class Hal
  def initialize
    puts "Hello Dave. Do you want to hear a song?"
  end
end

print "hal_two sez: "
hal_two = Hal.new
hal_two.sing
```

Output:

```
hal_one sez: Daisy, Daisy, over the ocean blue...
hal_two sez: Hello Dave. Do you want to hear a song?
Daisy, Daisy, over the ocean blue...
```

Classes can be “re-opened” and added to at any time. You can do this to any class, even pre-defined Ruby classes like Array and Hash. Do not abuse this.

Getters

We’ve already seen that we can create a ‘getter’ method which in a sense makes an instance variable a “read-only” public property.

```ruby
class Student
  def name
    @name
  end
end
```

```
bobby = Student.new("Bobby Buttons")
puts bobby.name
```

Output:

```
Bobby Buttons
```

Setters

We can also create 'setter' methods:

```
class Student
  def name=(new_name)
    @name = new_name
  end
end

jimbo = Student.new("Jimbo Jimmerson",92373673)
jimbo.name = "Jimmy Jimmerson"
puts jimbo.name
```

Output:

```
Jimmy Jimmerson
```

Get Set Shortcuts

Ruby provides *shortcut* methods that you can add to you classes to write getter and setter methods for you.

For example, to create getters and setters for the `@name` instance variable:

```
attr_accessor :name
```

Note that we specify the instance variable using a symbol. The shortcut methods can also take a comma-delimited list of symbols.

Get Set Shortcuts Example

```
class Student
  attr_accessor :name # create getter and setter methods
  attr_reader :gpa # create getter only
```

```
  attr_writer :password # create setter only
end


a_student = Student.new("Wally Glutton",337392,3.5)
puts "Student name: " + a_student.name
a_student.name = "Kilgor Trout"
puts "Student name after setter: " + a_student.name
a_student.password = "gorgonzola77"
```

Output:

```
Student name: Wally Glutton
Student name after setter: Kilgor Trout
```

The above example assumes that we are adding to the Student class we've already defined. Hence the Student constructor that takes three arguments.

Although we had no problem setting the password, attempts to access it would fail as it is read-only:

```
puts "Student Password: " + a_student.password
```

Access Control

By default all class methods are public. We can also make private and protected methods.

```
class Student
  protected
  def secret_student_handshake
      # How the heck do I code a handshake?
  end
end
```

The `secret_student_handshake` method will only be available to instances of the Student class (as well as to instances of any class that is derived from the Student class).

Private methods are created in the same way. Private methods of a specific object are only available within that specific object.

Class Scope 'Things'

So far the class variables and methods we've been creating are scoped at the instance level. What if we wanted to create certain things that were scoped across a class.

Placing a constant within a class will make it accessible to every method of a class.

We can also create class methods which can only be called via a class (not an object).

Class Constants

```ruby
class Student
  Location = "Red River College" # Class Constant

  def initialize(name, student_number, gpa)
    @name = name
    @student_number = student_number
    @gpa = gpa
  end

  def name_and_location
    @name + "is enrolled at " + Location
  end
end

wally = Student.new("Wally Glutton",3837293,3.52)
puts wally.name_and_location
```

Output

```
Wally Glutton is enrolled at Red River College
```

Class Methods

Class level methods are often called “static” methods in other languages.

```ruby
class Student
  def initialize(name, student_number, gpa)
    @name = name
    @student_number = student_number
    @gpa = gpa
  end

  # Class methods are always defined as self.method_name
  def self.generic_student
    new("Generic Student", 0, 0)
  end
end
```

☝️ Here we’ve created a class level factory method that returns as generic `Student` object. Ruby doesn’t have constructor overloading but class methods like this can serve the same purpose.

You do not need an object instance of a class to execute a class method as they are called by way of the class itself:

```
generic = Student.generic_student
puts generic.class
```

Sub-Classes

The less-than operator `<` is used to indicate class inheritance.

```
class Mammal
  def eat
    puts "Mmmmmmm Yum Yum"
  end
end

class Dog < Mammal # Dog inherits from Mammal
  def speak
    puts "woof woof"
  end
end
```

```ruby
fido = Dog.new
fido.speak # This method is specific to the Dog class.
fido.eat # This method was inherited from the Mammal class.
puts "Fido is a #{fido.class}."
puts "Fido is also a Mammal." if fido.is_a?(Mammal)
```

Output:

```
Woof Woof
Mmmmmmm Yum Yum
Fido is a Dog.
Fido is also a Mammal.
```

If a child class wants to hand-off details to its parent's constructor we use the `super` method:

```ruby
class Mammal
  def initialize(name) # Constructor with a single argument.
    @name = name
  end
end
```

```ruby
class Dog < Mammal # Dog inherits from Mammal
  def initialize(name, number_of_ears) # Constructor with two arguments.
    super(name) # Call the single argument Mammal constructor to allow the parent to set the @name variable.
    @number_of_ears = number_of_ears
  end
end
```