In this lab we will be exploring object-oriented programming in ruby. Ruby supports object-oriented notions like classes, inheritance, and polymorphism.

A class is a piece of code that serves as a template for the creation of an object in exactly the same way that Plato's forms were abstract philosophical templates for real world objects, so a class can be seen as a way in programming to describe objects with common properties.

So lets make a class that describes a person. Our person class will have a name and an age create person.rb and copy the following snippet into it.

class Person

def initialize(aName, aAge)

@name = aName

@age = aAge

end

def name=( aName )

@name = aName

end

def name

return @name

end

def age=( age )

@age = aAge

end

def age

return @name

end

end

So, there is quite a lot going on in this snippet of code, Person is the name of this class, initialize is the constructor for this class, this is a special method associated with a class and is used for making new instances of a class.

An instance of a class is an object that belongs to a class for example if we wanted to encode a details about a person called Josh who is 30 this would require making an instance of the person class.

We can see this constructor takes in two variables aName and aAge these being a name for the person and age of the person.

@name and @age are instance variables these are variables that begin with @ and can be accessed anywhere within the class. To allow the aName and aAge be accessed anywhere with the class these are assigned to @age and @name.

def name= and def age= are set methods which can be used to update the @name and @age of a person object.

def name and def age are get methods which can be used to get the name and age from a person object.

So lets make a new instance of this person class for Josh who is 30. Update your person.rb with the following snippet:

p1 = Person.new("Josh", 30)

Lets get Josh's age now update your person.rb with the following snippet:

puts(p1.name)

puts(p1.age)

Run ruby person.rb to run the file and you should see joshs name and age be printed console.

Lets use the name setter to update Josh's name to Joshua. Update your person.rb with the following snippet:

p1.name="Joshua"

puts p1.name

Run ruby person.rb to run the file and you should see joshs updated name printed to the console.

There is quite a lot of code in defining setters and getters so far. A very nice and common shortcut for making setters and getters in ruby is using attr\_accessor using this will allow use to define setters and getters in much fewer lines of code.

Update your person.rb to look like the following

class Person

attr\_accessor :name, :age

def initialize(aName, aAge)

@name = aName

@age = aAge

end

end

p1 = Person.new("Josh", 30)

puts(p1.name)

puts(p1.age)

p1.name="Joshua"

puts p1.name

Run ruby person.rb to make sure there are no errors in your code. Isn’t that amazing with one line of code we have setters and getters for our name and age!

So far we have seen that we can have variables that can be accessed anywhere in a class but what if we have a variable that is common across all class?

We can use @@. Let's make a variable to count how many people there are in our program and a method called numPersons that will print out the number of people in our program.

A method is a function that belongs to a class.

Update your person.rb to look like the following.

class Person

@@counter = 0

attr\_accessor :name, :age

def initialize(aName, aAge)

@name = aName

@age = aAge

@@counter +=1

end

def numPersons

puts(@@counter)

end

end

p1 = Person.new("Josh", 30)

puts(p1.name)

puts(p1.age)

p1.name="Joshua"

puts p1.name

p2 = Person.new("Keith", 21)

p1.numPersons

p2.numPersons

Run ruby person.rb we can see we added another person called Keith now, and when we call the p1.numPersons and p2.numPersons we can see we will get 2 this is because the @@counter variable is shared between classes.

Method names should begin with a lowercase letter. If you begin a method name with an uppercase letter, Ruby might think that it is a constant and hence can parse the call incorrectly.

**Note:**

Local variables (and pseudo variables such as self and nil) begin with a lowercase letter or an underscore and can be accessed in the function or method where they are defined and is not visible anywhere else in the program.

Instance / Object variables begin with @these are variables that can only be used within the same instance of an object.

Class variables (within a class) begin with @@ (two at signs) is a variable that is shared between a given class. If the value of the class variable changes, it will have changed for all instances of the object.

Global variables begin with $ (a dollar sign) these are variables that can be accessed from anywhere within an program.

Constants begin with capital letters and value should never change.

Right now if we attempted to puts(p1) to see the contents details about p1 we would see something like "#<Person:0x00007f95f7193888>" in our terminal. This is because by default Ruby will print the memory address of a class instance. If we wanted to change this we can use define a to\_s method to choose what is printed when puts(p1) so so lets print the persons name and age out.

Update your person.rb to look like:

class Person

@@counter = 0

attr\_accessor :name, :age

def initialize(aName, aAge)

@name = aName

@age = aAge

@@counter +=1

end

def to\_s

"The persons name is #{@name} and they are #{@age}\n"

end

def numPersons

puts(@@counter)

end

end

p1 = Person.new("Josh", 30)

puts(p1.name)

puts(p1.age)

p1.name="Joshua"

puts p1.name

p2 = Person.new("Keith", 21)

p1.numPersons

p2.numPersons

puts(p1)

We can now see when we use puts(p1) we get the following message: The persons name is Joshua and they are 30 years old.

Try adding "puts(p2)" on a new line and run your person.rb file to see what the results are.

This is called overriding all classes in ruby come with a to\_s method which by default prints out the memory address of an object here we are overriding our to\_s method to print details about our Person.

Inheritance is a mechanism that enables one class to inherit behaviour and attributes of another class. So what if we wanted to make a student class that inherits from the Person class, the student class will have a student number, we will also want to override the to\_s method aswell to tell us the students student number.

Update your person.rb to contain the following:

class Student < Person

attr\_accessor :studentNumber

def initialize(aName, aAge, aStudentNumber)

super(aName, aAge)

@studentNumber = aStudentNumber

end

def to\_s

"The students student number is #{@studentNumber}, their name is #{@name} and they are #{@age}\n"

end

end

s1 = Student.new("Michael", 25, "X11223344")

puts(s1)

Ruby allows you to view all the methods /instance variables associated with an object using the “methods” method. To view the available methods of our Student class update your person.rb file to contain.

puts s1.methods

puts s1.instance\_variables

**Task**

Create an Animal class that has a breath method which will print "inhale and exhale"

Create a Cat class that will inherit from the animal class and will have a method called speak when this method is run speak method will print "Meow"

Create a Dog class that will inherit from the animal class and will have a method called speak when this method is run speak method will print "Woof"

Create instances of the dog class and the cat class and call the speak method on both.

When complete upload all files you created for this lab here!

https://www.geeksforgeeks.org/method-visibility-in-ruby/