Object-Oriented Programming

What you've been doing thus far programming-wise is called procedural programming. It proceeds line-by-line. The problem with it is that it is difficult to write complicated programs using this method.

Object-oriented programming assumes that everything in the world is an object. You model the object for the computer. For example, a person is an object. What are the attributes or properties that define a person? Name, age, height, and so on. An object also has actions, things that the object can do, such as: run, jump, code, cry, and repeat. So objects have states and actions.

To create an object in Ruby, you first have to create a class and give it a name.

```
class Person

end
```

All classes begin with a capital letter. They are variables, but constant variables. Classes end with `end`.

You can think of a class like a **factory** that produces **instances** of the class, called **objects**.

```
class Person



end


p = Person.new
```

The p variable is an **object**. You can create as many of them as you like. They are all objects.

```
p1 = Person.new

p2 = Person.new

p3 = Person.new

p4 = Person.new
```

Enter pry. Remember how with functions, you define them once, but use them over and over? Classes are similar. Notice that when you create a class, pry knows you aren't done. That's because it needs to end with **end**.

```
~/Documents/wdi3/ruby/lectures ✓ master $ pry

[1] pry(main)> class Person

[1] pry(main)* end
```

```
=> nil

[2] pry(main)> Person

=> Person
```

Now create a few people. What are the numbers that show up? Those are the addresses in memory.

```
[3] pry(main)> p1 = Person.new

=> #<Person:0x007fc622a17800>

[4] pry(main)> p2 = Person.new

=> #<Person:0x007fc622a61ab8>

[5] pry(main)> p3 = Person.new

=> #<Person:0x007fc622ab2030>
```

Let’s do some more to make this class a bit more interesting. Inside classes, you can create methods, aka functions. We’ll create one for speak. It won’t have any inputs.

```
[6] pry(main)> class Person

[6] pry(main)*   def speak

[6] pry(main)*     puts "good morning"

[6] pry(main)*   end

[6] pry(main)* end

=> nil
```

Now if we create a person object, it can speak.

```
[7] pry(main)> p1 = Person.new

=> #<Person:0x007fc622a6a910>

[8] pry(main)> p1.speak

good morning

=> nil
```

The speak function belongs to the Person class. Whenever you create a function in a class, the objects made from the class can use the function.

What if we want to tell the person what to say? We'll create a new function with some input.

```
[9] pry(main)> class Person

[9] pry(main)*   def talk(words_to_say)

[9] pry(main)*     puts "i like saying #{words_to_say}"

[9] pry(main)*   end

[9] pry(main)* end

=> nil

[10] pry(main)> Person.new

=> #<Person:0x007fc622b50d20>

[11] pry(main)> p1.speak

good morning

=> nil

[12] pry(main)> p1.talk("this is cool")

i like saying this is cool

=> nil
```

Why can the person still speak? Pry/Ruby remembers all the other functions you've added to a class.

Let's try it out in our text editor, since what we've done so far is interesting, but not that useful.

```
~/Documents/wdi3/ruby/lectures ✓ master $ touch 2013-01-31.rb

~/Documents/wdi3/ruby/lectures ✗ master $ subl 2013-01-31.rb
```

A class or an object can also remember things. Set some functions to set the age of a person, then get it later.

```
class Person
  def set_age(age)
  end

  def get_age
  end
end
```

How can you get the program to remember something. Use a variable, putting an @ symbol in front of it. This makes the variable an instance variable, which lives inside an object. As long as the object lives, the instance variable also lives. Add pry and put a `binding.pry` at the end so we can play around with this in the terminal.

```ruby
require 'pry'

class Person
  def set_age(age)
    @age = age
  end

  def get_age
    @age
  end
end

binding.pry
```

```
~/Documents/wdi3/ruby/lectures ✗ master $ ruby 2013-01-31.rb


From: 2013-01-31.rb @ line 13 :


      8:   def get_age

      9:     @age

     10:   end

     11: end

     12:

 => 13: binding.pry


[1] pry(main)> p1 = Person.new

=> #<Person:0x007f8d95b20d98>

[2] pry(main)> p2 = Person.new

=> #<Person:0x007f8d95a52038>

[3] pry(main)> p1.set_age(21)

=> 21

[4] pry(main)> p2.set_age(23)
```

```
=> 23

[5] pry(main)> p1.get_age

=> 21

[6] pry(main)> p2.get_age

=> 23
```

The instance variable **@age** is the key to the age being remembered. Normally when a function runs, all the variables disappear when the function ends. The next time the function is run, it starts over with variables, sort of like "Groundhog Day". The instance variable saves the information—no Groundhog Day!
What would be a better way to set and get the age? Wouldn't it be easier to just set the age and get it by calling **age**? Keep in mind that you cannot have functions within the same program with the same name.

Functions in Ruby can have special characters at the end, such as ?, which returns true or false, or !, which means the function is potentially dangerous, and =, which differentiates functions with similar names. The first function name is "age equals".

```ruby
class Person
  def age=(age)
    @age = age
  end

  def age
    @age
  end
end
```

Note that although the **age=** and **age** functions we've defined look similar, they are completely different functions without any special relation to each other.

```
~/Documents/wdi3/ruby/lectures ✗ master $ ruby 2013-01-31.rb


From: 2013-01-31.rb @ line 13 :


     8:   def age

     9:     @age

    10:   end

    11: end
    12:

 => 13: binding.pry


[1] pry(main)> p1 = Person.new

=> #<Person:0x007faf6f0d8880>

[2] pry(main)> p1.age = 20

=> 20

[3] pry(main)> p1.age

=> 20
```

Now when you set the variable age, Ruby knows to call the function with the = appended on it. You could also do this, which is the same thing.

```
[4] pry(main)> p1.age=(20)

=> 20
```

Now add some functionality that will let you both get and set the age, name, and gender, then create some objects with information.

```ruby
class Person
  def age=(age)
    @age = age
  end

  def age
    @age
  end

  def gender=(gender)
    @gender = gender
  end

  def gender
    @gender
  end

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

end
```

```
[1] pry(main)> p1 = Person.new

=> #<Person:0x007fc1dd91cb88>

[2] pry(main)> p1.age = 3

=> 3

[3] pry(main)> p1.gender = 'm'

=> "m"

[4] pry(main)> p1.name="sam"

=> "sam"

[5] pry(main)> p2 = Person.new

=> #<Person:0x007fc1dd96d6a0>

[6] pry(main)> p2.age = 5

=> 5

[7] pry(main)> p2.name = "sally"

=> "sally"

[8] pry(main)> p2.gender = "f"

=> "f"

[9] pry(main)> p1

=> #<Person:0x007fc1dd91cb88 @age=3, @gender="m", @name="sam">

[10] pry(main)> p2

=> #<Person:0x007fc1dd96d6a0 @age=5, @gender="f", @name="sally">
```

Notice that these person objects are sort of like hashes, but this is a way better way to organize information.

This is cool, but how can we set information for an object in bulk? When you create a new object using **`.new`**, you can, when you initialize it, set variables to values. Create a method called initialize. Make sure you spell it correctly. Pass age, gender and name as inputs for this function.

```
class Person
  def initialize(age, gender, name)
     @age = age
     @gender = gender
     @name = name
  end
...
end
```

This sets the values initially, and you can use the other functions to change the values later.

```
[1] pry(main)> p1 = Person.new

ArgumentError: wrong number of arguments (0 for 3)

from 2013-01-31.rb:4:in `initialize'
```

Inputs are called parameters when you are writing the function and arguments when you are passing them to the function. `.new` is a function, which calls the initialize function. You cannot create the object without passing in the arguments.

```
[2] pry(main)> p1 = Person.new(21, 'f', 'sally')

=> #<Person:0x007fbb02e951c8 @age=21, @gender="f", @name="sally">

[3] pry(main)> p2 = Person.new(23, 'm', 'sam')

=> #<Person:0x007fbb02eac990 @age=23, @gender="m", @name="sam">

[4] pry(main)> p1.age

=> 21

[5] pry(main)> p2.gender

=> "m"
```

If you removed the remaining functions, you wouldn't be able to update the variables once they were initialized. You may want to do this; for example, some sites don't let you change your username once it's been set.

Now imagine that you want to add more properties. Each one requires 6 lines of code as we've written it. There's a faster way to do this that will do what we just did behind the scenes. So we'll refactor our code to make it better.

```
class Person
  attr_accessor :age, :name, :gender
  def initialize(age, gender, name)
     @age = age
     @gender = gender
```

```ruby
    @name = name
  end
end
```

What happens when you say `attr_accessor :age`? `attr_accessor` is a special function (note the color change; in your editor it will be red) that reproduces the 6 lines of code needed for each element of information (it's all still there, you just didn't have to write it; the computer wrote it for you). `:age` is a symbol. Let's try it out in the terminal to see if it works.

```
[1] pry(main)> p1 = Person.new(21, 'f', 'sally')

=> #<Person:0x007ffee2616dc0 @age=21, @gender="f", @name="sally">

[2] pry(main)> p1.age

=> 21

[3] pry(main)> p1.name

=> "sally"
```

In pry, you can `cd` into an object. The prompt is now Person and you are inside that person object. Use `ls` to see the variables available. Notice that it has all the variables we originally created, but are now created by the `attr_accessor` function (`@age=`, `@age`, etc.). You can also see the instance variables.

```
[4] pry(main)> cd p1

[5] pry(#<Person>):1> ls

Person#methods: age  age=  gender  gender=  name  name=

self.methods: __pry__

instance variables: @age  @gender  @name

locals: _  __  _dir_  _ex_  _file_  _in_  _out_  _pry_
```

You can also `cd` into a number and `ls` all the functions that a number can do!

```
[7] pry(main)> cd 3

[8] pry(3):1> ls

Comparable#methods: between?

Numeric#methods:

  +@      coerce     i          phase                quo     rectangular
```

```
to_c

  abs2   conj       imag       polar               real   remainder

  angle  conjugate  imaginary  pretty_print        real?
singleton_method_added

  arg    eql?       nonzero?   pretty_print_cycle  rect   step

Integer#methods:

  ceil  denominator  floor  gcdlcm    lcm   numerator  pred          round
to_i    to_r       upto

  chr   downto       gcd    integer?  next  ord        rationalize  times
to_int  truncate

Fixnum#methods:

  %   *   +  -@  <   <=   ==   >   >>  ^       abs  divmod  fdiv
modulo  size  to_f  zero?  ~

  &  **  -  /   <<  <=>  ===  >=  []  __pry__  div  even?  magnitude  odd?
succ  to_s  |

locals: _  __  _dir_  _ex_  _file_  _in_  _out_  _pry_
```

Notice that when we create a new object and ask it about itself, the output is not very friendly.

```
[1] pry(main)> p1 = Person.new(32, 'f', 'sally')

=> #<Person:0x007fbe33443128 @age=32, @gender="f", @name="sally">

[2] pry(main)> p1

=> #<Person:0x007fbe33443128 @age=32, @gender="f", @name="sally">
```

How might you want it to appear instead? What if you typed in p1 and the output was something like, “sally is a 32 year-old female”. Imagine that p1 is the avatar for the object and we can change that avatar. The **`.to_s`** function takes any kind of object and turns it into a string. We can create our own version of **`.to_s`**.

```
def to_s
  "this is cool"
end
```

```
[1] pry(main)> p1 = Person.new(32, 'f', 'sally')

=> this is cool
```

```
[2] pry(main)> p1

=> this is cool

[3] pry(main)> p1.to_s

=> "this is cool"
```

You can see here that calling p1 implicitly calls the `.to_s` function. The pry console does this automatically when you ask it for a variable. Let's update our `to_s` function to describe our object.

```ruby
def to_s
  "#{name} is a #{age} year old #{gender}"
end
```

```
[1] pry(main)> p1 = Person.new(32, 'female', 'sally')

=> sally is a 32 year old female

[2] pry(main)> p2 = Person.new(21, 'male', 'sam')

=> sam is a 21 year old male

[3] pry(main)> p1

=> sally is a 32 year old female

[4] pry(main)> p2

=> sam is a 21 year old male
```

This shows you that there are two person objects in memory and presents the output in a friendlier way.

The functions that we've talked about thus far are called instance methods. These use instance variables, including attr_accessor, which is using instance variables in the form of symbols.

There are also class functions. They include the class name in the function name.

```ruby
def Person.speak
 puts "I am a Person class!"
end
```

They are called on the class.

```
[1] pry(main)> Person.speak

I am a Person class!
```

```
=> nil
```

How is this useful? What if you want to do something, but you don't need an object to accomplish it? If the function is going to do the same thing no matter which Person (or other class) called it, it's better to just use a class function.

```ruby
def Person.squared(x)
  puts "The square of #{x} is #{x**2}"
end
```

```
[2] pry(main)> Person.squared(3)

The square of 3 is 9

=> nil
```

Inheritance

Classes can inherit from other classes. What do they inherit? Functions.

```ruby
class Animal
  def speak
    puts "I am an animal"
  end
end
```

To call this class, you would do something like a1 = Animal.new. This animal object can speak.

```
[1] pry(main)> a1 = Animal.new

=> #<Animal:0x007fabc156be58>

[2] pry(main)> a1.speak

I am an animal

=> nil
```

What if we wanted a Person to be able to speak? That's where inheritance comes in.

```ruby
class Person < Animal
```

```
[1] pry(main)> p1 = Person.new(32, 'female', 'sally')
```

```
=> sally is a 32 year old female

[2] pry(main)> p1.speak

i am an animal!

=> nil
```

The person object can now speak too. Let’s cd into it. Notice that the Animal methods are available here, along with the Person methods.

```
[3] pry(main)> cd p1

[4] pry(#<Person>):1> ls

Animal#methods: speak

Person#methods: age  age=  gender  gender=  name  name=  to_s

self.methods: __pry__

instance variables: @age  @gender  @name

locals: _  __  _dir_  _ex_  _file_  _in_  _out_  _pry_
```

The p1 object knows which class it comes from:

```
[6] pry(main)> p1.class

=> Person
```

and it accesses the class to ask if there are methods available when a function is called. If it is, the function is run. But, speak is not in the Person class. So how does the Person class know that it can use the Animal class methods too? The Person class looks for its ancestors and finds the speak function there. Note that the Animal class may inherit from other classes. How do you know what the ancestors are?

```
[7] pry(main)> Person.ancestors

=> [Person, Animal, Object, PP::ObjectMixin, Kernel, BasicObject]
```

The Person class inherits from the Animal class, which inherits implicitly from the Object class, which is like God. This is called a call chain and it only works up the chain. If the method called is not in this chain somewhere, an error message results.

```
[8] pry(main)> p1.x

NoMethodError: undefined method `x' for sally is a 32 year old
female:Person
```

```
from (pry):6:in `<main>'
```

A class can only inherit from one class: single inheritance. There is a trick though that allows you to 'inherit' from other type things, using modules.

Modules are like methods, but they cannot be instantiated. They are containers that hold functions. Modules are included in the class that will use this functionality.

```
module A
end

module B
end
```

Now add some methods to the modules.

```
module A
  def stuff1
    puts "this is stuff1"
  end
  def stuff2
    puts "this is stuff2"
  end
end

module B
  def temp1
    puts "temp1"
  end
  def temp2
    puts "temp2"
  end
end
```

Now add these modules to the Person class.

```
class Person < Animal
  include A
  include B
  attr_accessor :age, :name, :gender
...
end
```

```
[1] pry(main)> p2 = Person.new(21, 'male', 'sam')

=> sam is a 21 year old male
```

```
[2] pry(main)> cd p2

[3] pry(#<Person>):1> ls

Animal#methods: speak

A#methods: stuff1  stuff2

B#methods: temp1  temp2

Person#methods: age  age=  gender  gender=  name  name=  to_s

self.methods: __pry__

instance variables: @age  @gender  @name

locals: _  __  _dir_  _ex_  _file_  _in_  _out_  _pry_
```

Now you see that there are more methods available for the Person class.

Now check out the call chain using **`.ancestors`**. Notice that the modules B and A are in the chain.

```
[4] pry(#<Person>):1> Person.ancestors

=> [Person, B, A, Animal, Object, PP::ObjectMixin, Kernel, BasicObject]
```

Modularizing your code makes it easy for you to swap in and out functionality. Think of the modules like Legos that you can put together into something cool, and it remains flexible for you to make changes.

Are there conventions for creating these modules? Each class or module would be its own `.rb` file (`person.rb, animal.rb, A.rb, B.rb`). There would be a `main.rb` file as well. The `person.rb` file requires the A, B, and Animal files. The `main.rb` file requires the `person.rb` file. If you included the modules in the Animal class instead, the Person class would be able to access the functionality in these modules. Modules can include other modules, though this may make your code less flexible.

Inheritance is not used that much in the real world, nor are modules.