Ruby
Classes & Objects

Contents

1. Ruby Review

1. What elements is a String made up of?

- Characters

2. What would you use if you wanted to check a numerical range in an If statement?

- Two comparisons on either side of an AND (&&).
- .include? on a Range (0..10)

3. How do you remove the last item in an Array?

- *.pop*

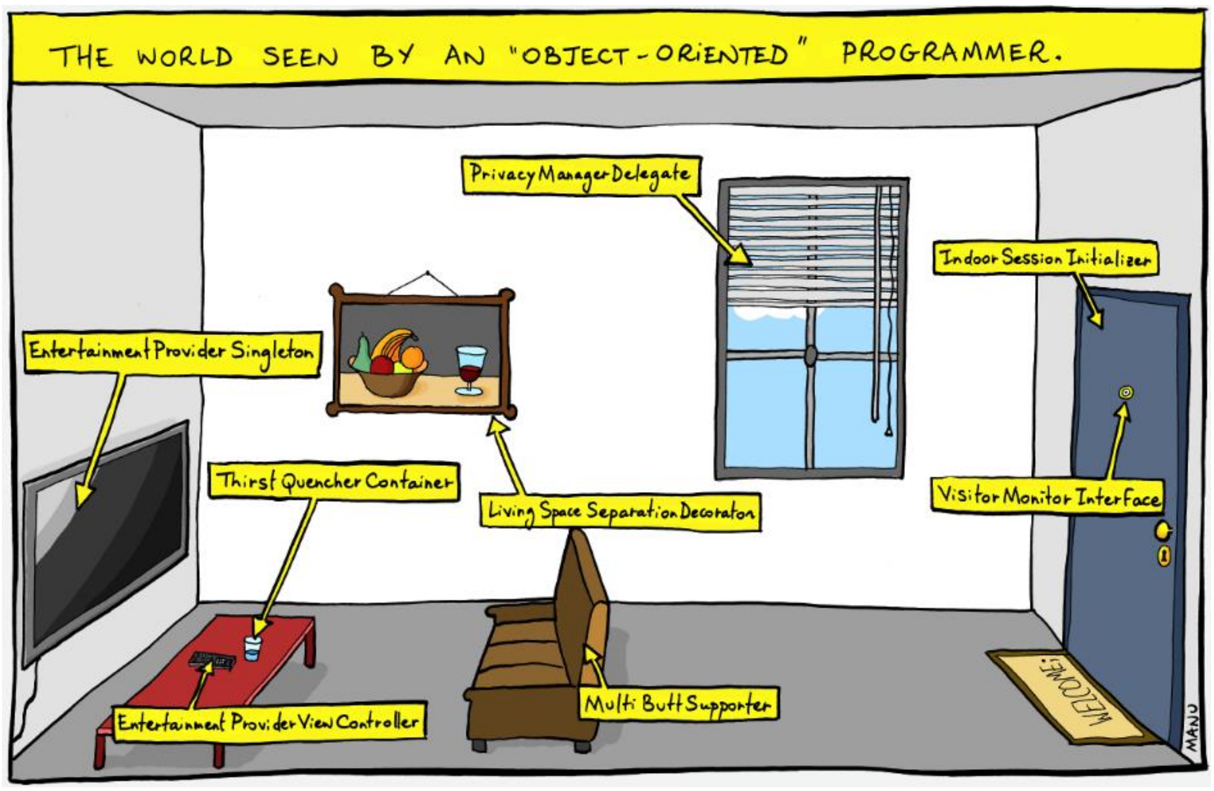
4. What two data types are best used for keys in a Hash?

- *string* and *symbol*

5. How many arguments can a method take?

- *Trick Question! No Limit!*

2. Object-Oriented World

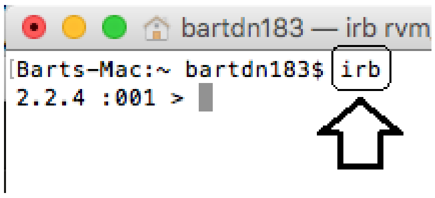

THE WORLD SEEN BY AN "OBJECT-ORIENTED" PROGRAMMER.
PrivacyManagerDelegate
IndoorSessionInitializer
EntertainmentProviderSingleton
ThirstQuencherContainer
LivingSpaceSeparationDecorator
VisitorMonitorInterface
MultiButtSupporter
EntertainmentProviderViewController
WELCOME!
MANU

3. Explore Ruby Hierarchical Organization

Introduction to Object Oriented Programming

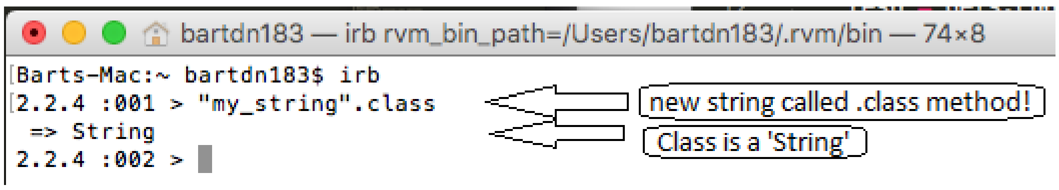
What do we mean when we say everything in ruby is an object?

Open up **Terminal** and enter **irb** (=> Interactive Ruby Shell)



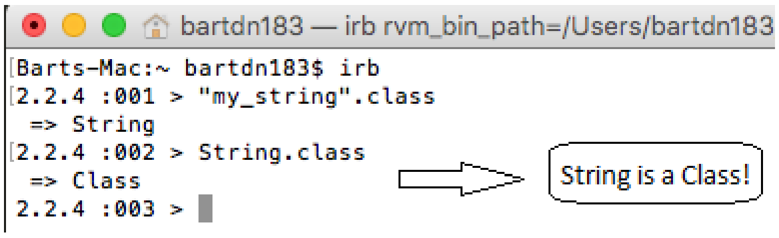
```
[Barts-Mac:~ bartdn183$ irb
 2.2.4 :001 >
```

Create a **string** and call **.class** method on it.


bartdn183 — irb rvm_bin_path=/Users/bartdn183/.rvm/bin — 74×8
new string called .class method!
Class is a 'String'

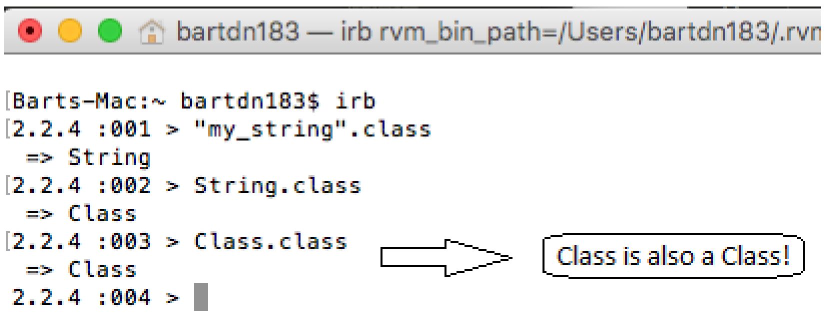
```
[Barts-Mac:~ bartdn183$ irb
[2.2.4 :001 > "my_string".class
  => String
 2.2.4 :002 >
```

Call the **.class** method on **String** itself.



```
[Barts-Mac:~ bartdn183$ irb
[2.2.4 :001 > "my_string".class
  => String
[2.2.4 :002 > String.class
  => Class
 2.2.4 :003 >
```

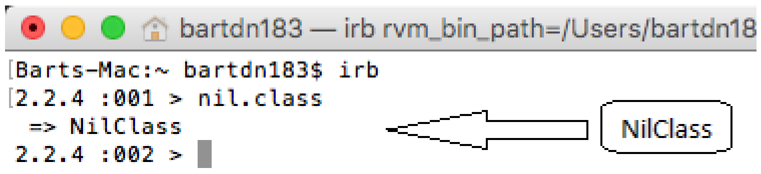
Call the **.class method** on **Class** itself.



```
[Barts-Mac:~ bartdn183$ irb
[2.2.4 :001 > "my_string".class
  => String
[2.2.4 :002 > String.class
  => Class
[2.2.4 :003 > Class.class
  => Class
 2.2.4 :004 >
```

⇨ **Ruby organizes information into different classes**!

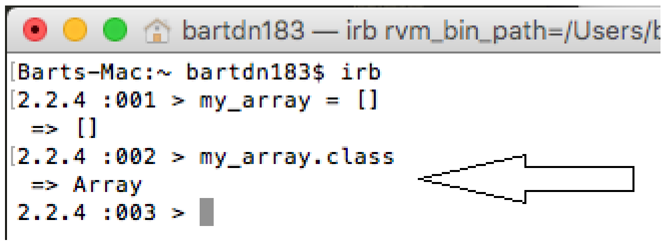
Call the **.class** method on nil:



```
Barts-Mac:~ bartdn183$ irb
2.2.4 :001 > nil.class
 => NilClass
2.2.4 :002 >
```

⇨ **Nilclass** is a class of the object **nil**.

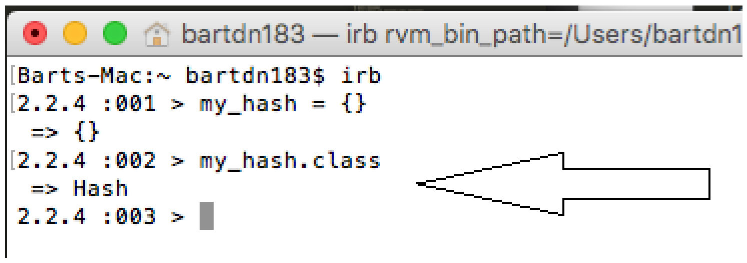
Set a variable on empty **Array**. Then call the **.class** method on the variable:



```
Barts-Mac:~ bartdn183$ irb
2.2.4 :001 > my_array = []
 => []
2.2.4 :002 > my_array.class
 => Array
2.2.4 :003 >
```

⇨ Arrays (Array class) are ordered, integer-indexed data collections.

Set a variable on empty Hash. Then call the .class method on the variable:

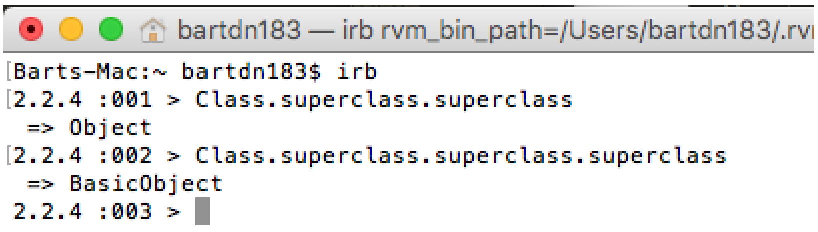


```
Barts-Mac:~ bartdn183$ irb
2.2.4 :001 > my_hash = {}
 => {}
2.2.4 :002 > my_hash.class
 => Hash
2.2.4 :003 >
```

⇨ All of these classes are a member of a Class, including **Class** itself.

How does that fit the idea that everything is an object?

To go further up back up the organizational hierarchy, call the **.superclass** method twice on Class!



```
Barts-Mac:~ bartdn183$ irb
2.2.4 :001 > Class.superclass.superclass
 => Object
2.2.4 :002 > Class.superclass.superclass.superclass
 => BasicObject
2.2.4 :003 > 
```

⇨ The highest level of the organization is **BasicObject** and that everything organized in the language falls under this umbrella.

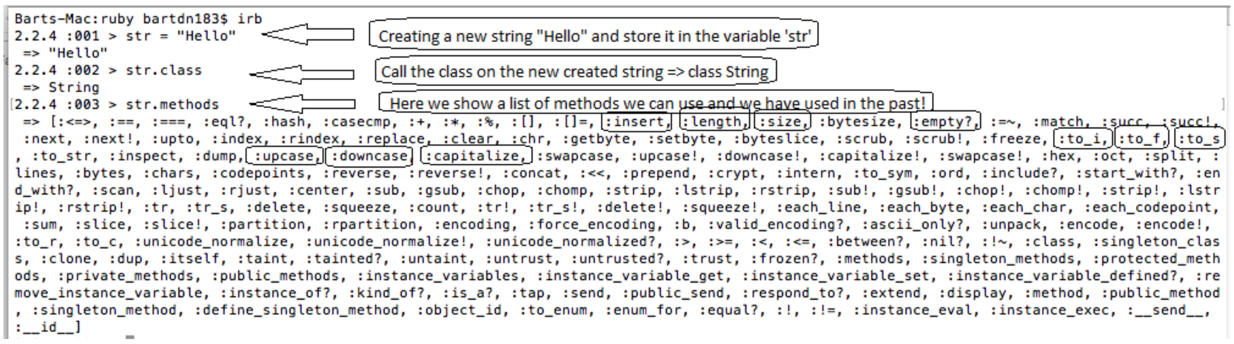
The idea of **everything is an object** may seem very abstract. It helps to think about the qualities of objects in the real world.

Example:

Pen is an object. Being a pen it has certain attributes. It is plastic, it is lightweight and it contains ink. It can also be used for a number of different functions, including writing (of course), as a bookmark, or to hold hair back in a bun. The pen is identified by certain qualities, and used for a variety of functions.

A string for example consists some number of characters and is contained between single or double quotes. We can manipulate strings by calling various methods on them, such as ".length", ".reverse", ".capitalize". The reason why those methods work on strings is because they're part of the "class String" and therefore inherit all the qualities of that class.

Let's look at the list of the existing **String Methods** we can use on strings:



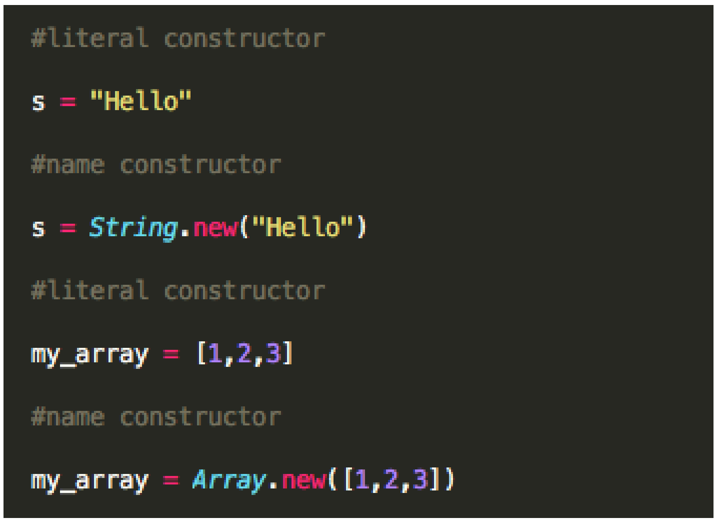
```
Barts-Mac:ruby bartdn183$ irb
2.2.4 :001 > str = "Hello"
 => "Hello"
2.2.4 :002 > str.class
 => String
2.2.4 :003 > str.methods
 => [:<=>, :==, :===, :eql?, :hash, :casecmp, :+, :*, :%, :[], :[]=, :insert, :length, :size, :bytesize, :empty?, :=~, :match, :succ, :succ!,
 :next, :next!, :upto, :index, :rindex, :replace, :clear, :chr, :getbyte, :setbyte, :byteslice, :scrub, :scrub!, :freeze, :to_i, :to_f, :to_s
, :to_str, :inspect, :dump, :upcase, :downcase, :capitalize, :swapcase, :upcase!, :downcase!, :capitalize!, :swapcase!, :hex, :oct, :split, :
lines, :bytes, :chars, :codepoints, :reverse, :reverse!, :concat, :<<, :prepend, :crypt, :intern, :to_sym, :ord, :include?, :start_with?, :en
d_with?, :scan, :ljust, :rjust, :center, :sub, :gsub, :chop, :chomp, :strip, :lstrip, :rstrip, :sub!, :gsub!, :chop!, :chomp!, :strip!, :lstr
ip!, :rstrip!, :tr, :tr_s, :delete, :squeeze, :count, :tr!, :tr_s!, :delete!, :squeeze!, :each_line, :each_byte, :each_char, :each_codepoint,
 :sum, :slice, :slice!, :partition, :rpartition, :encoding, :force_encoding, :b, :valid_encoding?, :ascii_only?, :unpack, :encode, :encode!,
:to_r, :to_c, :unicode_normalize, :unicode_normalize!, :unicode_normalized?, :>, :>=, :<, :<=, :between?, :nil?, :!~, :class, :singleton_clas
s, :clone, :dup, :itself, :taint, :tainted?, :untaint, :untrust, :untrusted?, :trust, :frozen?, :methods, :singleton_methods, :protected_meth
ods, :private_methods, :public_methods, :instance_variables, :instance_variable_get, :instance_variable_set, :instance_variable_defined?, :re
move_instance_variable, :instance_of?, :kind_of?, :is_a?, :tap, :send, :public_send, :respond_to?, :extend, :display, :method, :public_method
, :singleton_method, :define_singleton_method, :object_id, :to_enum, :enum_for, :equal?, :!, :!=, :instance_eval, :instance_exec, :__send__,
:__id__]
```

Creating a new string "Hello" and store it in the variable 'str'

Call the class on the new created string => class String

Here we show a list of methods we can use and we have used in the past!

To make a new string (or array, or hash) we can use a **literal constructor**, or we can use a **name constructor**.

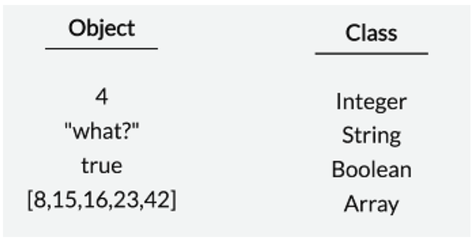


```ruby
#literal constructor

s = "Hello"

#name constructor

s = String.new("Hello")

#literal constructor

my_array = [1,2,3]

#name constructor

my_array = Array.new([1,2,3])
```

⇨ When we call a method, like .new, on the class itself, it's called a **class method**. Calling **.new** on a class produces an object within that class, also an instance of the class String. In our above example, the string "Hello" is an object, and also an instance of the class String. A method called on an instance, such as .length, is called an **instance method**. **"Hello".length** would be an example of an **instance method**.

Object vs Class

⇨ An object is a piece of data.
⇨ A class is what type of data that is.

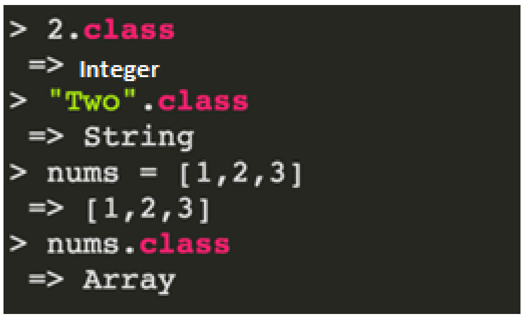


Object	Class
4	Integer
"what?"	String
true	Boolean
[8,15,16,23,42]	Array

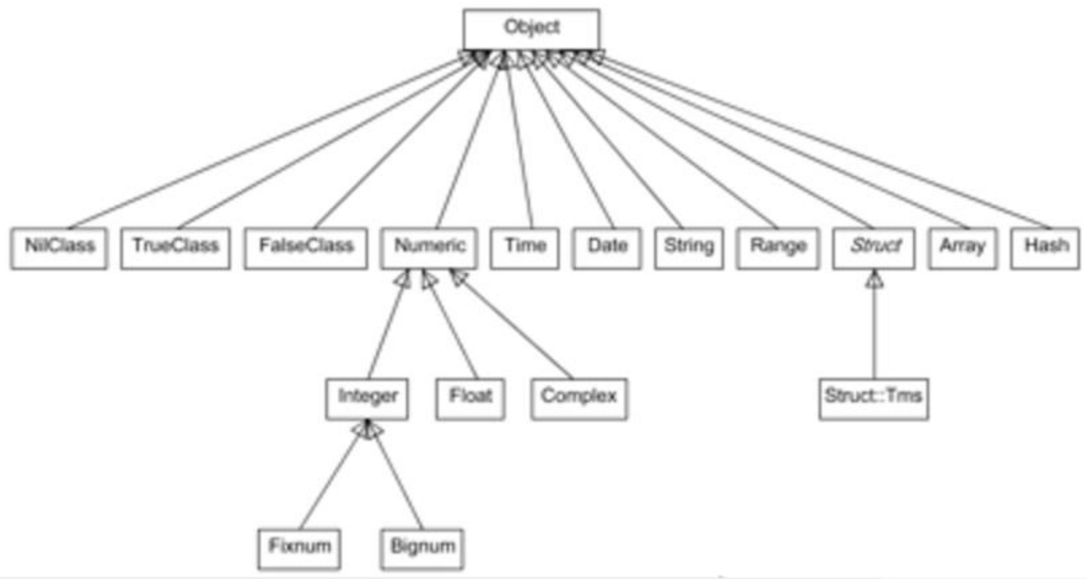
Data Types == Classes

So in Ruby, the different data types are considered classes (whereas any instance of the data type is an object).

There's even a way to tell what class and object is:



```
> 2.class
 => Integer
> "Two".class
 => String
> nums = [1,2,3]
 => [1,2,3]
> nums.class
 => Array
```

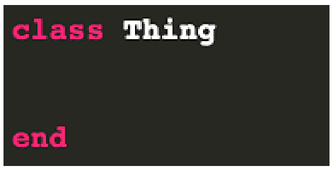


4. Define & Create Class

4.1. Class

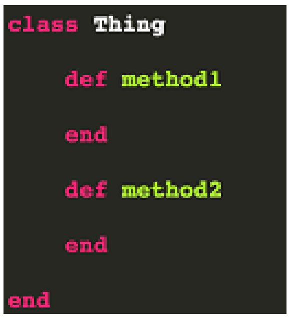
⇨ Give your program some Class

We define a Class using… well, **class** and there is end to tell Ruby when the definition is over!



```ruby
class Thing

end
```

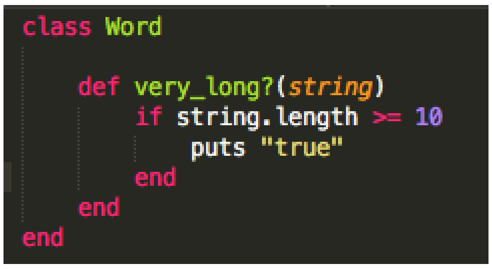
Notice that **Thing** – what the class is to be called – is capitalized and then we fill in with our methods.



```ruby
class Thing
    def method1
    end
    def method2
    end
end
```

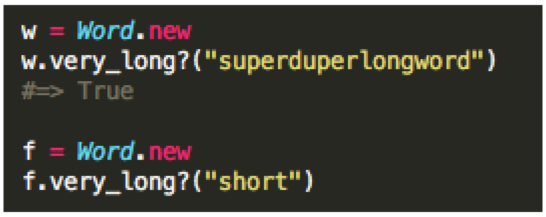
4.2. Define & Create Class

We have already used many of Ruby's built-in classes, like String, Array and Hash. Now let's make our own class. Remember that every class is an object, that every object has certain attributes and every object has certain functions it can perform. Let's make a class called **Word** (=> class **name** is always Capitalized!), and create a function called **very_long?**



```ruby
class Word
    def very_long?(string)
        if string.length >= 10
            puts "true"
        end
    end
end
```

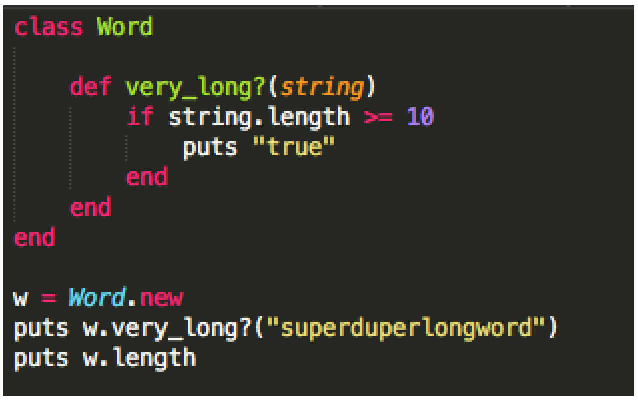
Let's create a new **Word** and call the method **very_long?** we just created on.



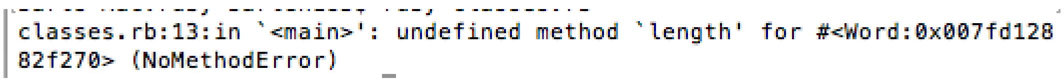
```
w = Word.new
w.very_long?("superduperlongword")
#=> True

f = Word.new
f.very_long?("short")
```

⇨ So far our Word class doesn't do very much except tell us whether or not a word is more or less than 10 characters.

Let's try a few things. Call the **.length** method on our word **w**. Then call the **.class** method on our word **w**. Then call the **.class** method.



```
class Word

    def very_long?(string)
        if string.length >= 10
            puts "true"
        end
    end
end

w = Word.new
puts w.very_long?("superduperlongword")
puts w.length
```

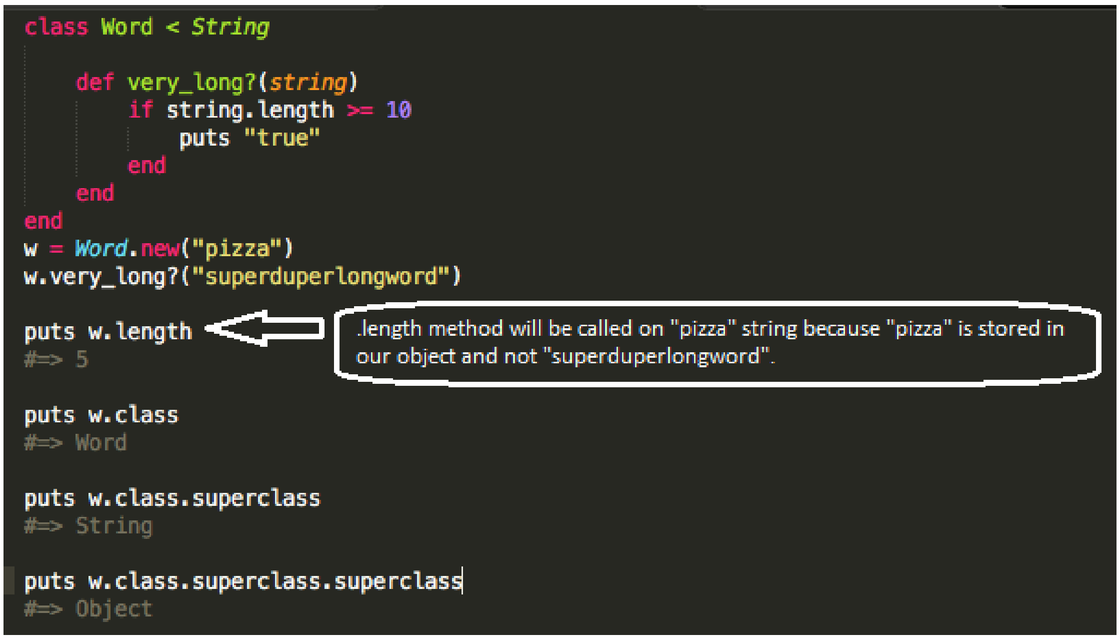


```
classes.rb:13:in `<main>': undefined method `length' for #<Word:0x007fd128
82f270> (NoMethodError)
```

Calling **.length** on our Word gives us an error message. **Word** is a Class, and does not have the same functions as the class **String** unless we set it up to inherit those functions.

4.3. Explore Class Inheritance

Let's setup the inheritance on the created **class Word** from the **class String**.



```
class Word < String

    def very_long?(string)
        if string.length >= 10
            puts "true"
        end
    end
end
w = Word.new("pizza")
w.very_long?("superduperlongword")

puts w.length
#=> 5

puts w.class
#=> Word

puts w.class.superclass
#=> String

puts w.class.superclass.superclass
#=> Object
```

.length method will be called on "pizza" string because "pizza" is stored in our object and not "superduperlongword".

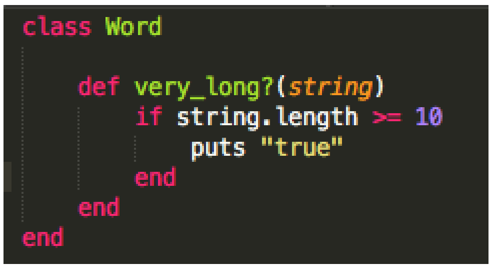
⇨ The inherited **.length** String method works now on our **class Word** but returns a 0. This is because our object **w** is empty. When we add a string to our object ("Pizza") it will return the length of the object!

Now that we have set up our class Word to inherit from the class String, we can call all the String methods on **w** and they work.

⇨ We can create our own classes but if we need the functionality of a **String** class and need to be able to call the existing string methods than we have to inherit them from the "**String Class**"

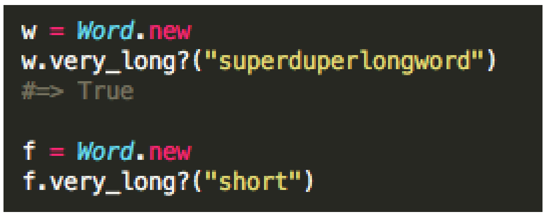
(=> `class Word < String`)

We have already used many of Ruby's built-in classes, like String, Array and Hash. Now let's make our own class. Remember that every class is an object, that every object has certain attributes and every object has certain functions it can perform. Let's make a class called **Word** (=> class **name** is always Capitalized!), and create a function called **very_long?**



```
class Word

    def very_long?(string)
        if string.length >= 10
            puts "true"
        end
    end
end
```

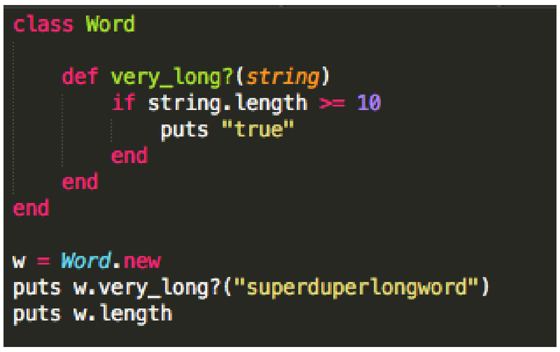
Let's create a new **Word** and call the method **very_long?** we just created on.



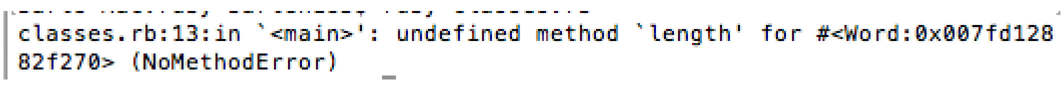
```
w = Word.new
w.very_long?("superduperlongword")
#=> True

f = Word.new
f.very_long?("short")
```

⇨ So far our Word class doesn't do very much except tell us whether or not a word is more or less than 10 characters.

Let's try a few things. Call the **.length** method on our word **w**. Then call the **.class** method on our word **w**. Then call the **.class** method.



```
class Word

    def very_long?(string)
        if string.length >= 10
            puts "true"
        end
    end
end

w = Word.new
puts w.very_long?("superduperlongword")
puts w.length
```

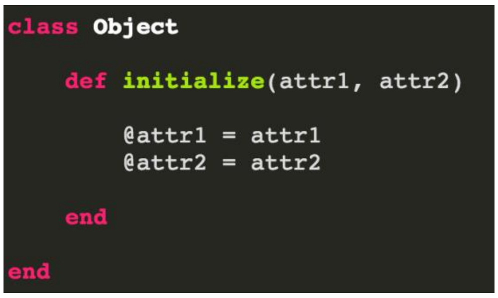


```
classes.rb:13:in `<main>': undefined method `length' for #<Word:0x007fd128
82f270> (NoMethodError)
```

Calling **.length** on our Word gives us an error message. **Word** is a Class, and does not have the same functions as the class **String** unless we set it up to inherit those functions.

4.4. Instance & Initialization

There is one method we need that is a must: **initialize**.
This method will allow for the creation of a new Object.

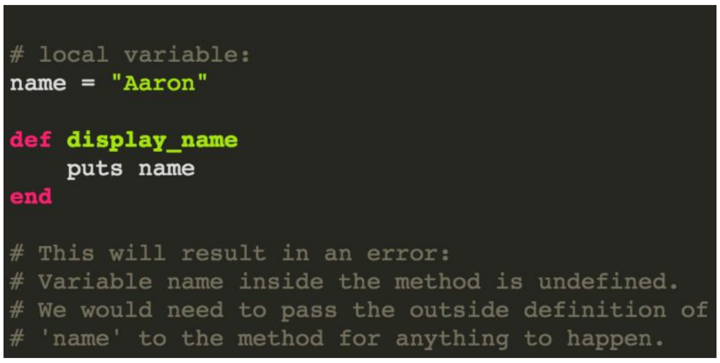


```
class Object

    def initialize(attr1, attr2)

        @attr1 = attr1
        @attr2 = attr2

    end

end
```

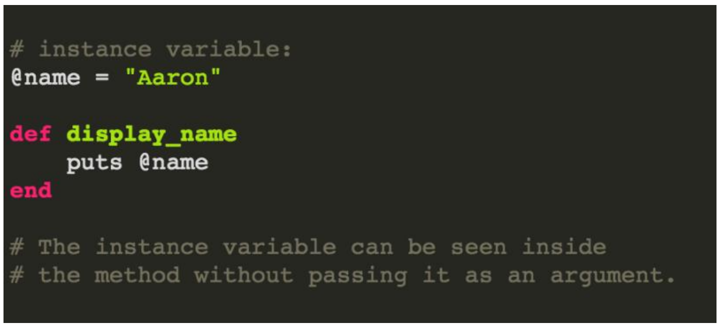
We pass the method **attributes** as arguments, and set these attributes to corresponding **instance variables** (variables preceded by an @-symbol). **Instance variables** can be used throughout the class definition.

4.5. Instance vs Local Variable

What is the difference?



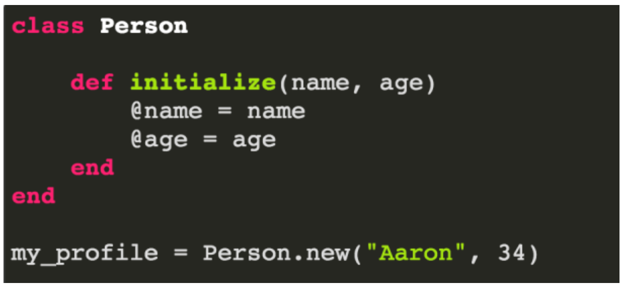
```
# local variable:
name = "Aaron"

def display_name
    puts name
end

# This will result in an error:
# Variable name inside the method is undefined.
# We would need to pass the outside definition of
# 'name' to the method for anything to happen.
```



```
# instance variable:
@name = "Aaron"

def display_name
    puts @name
end

# The instance variable can be seen inside
# the method without passing it as an argument.
```

5. Person Class

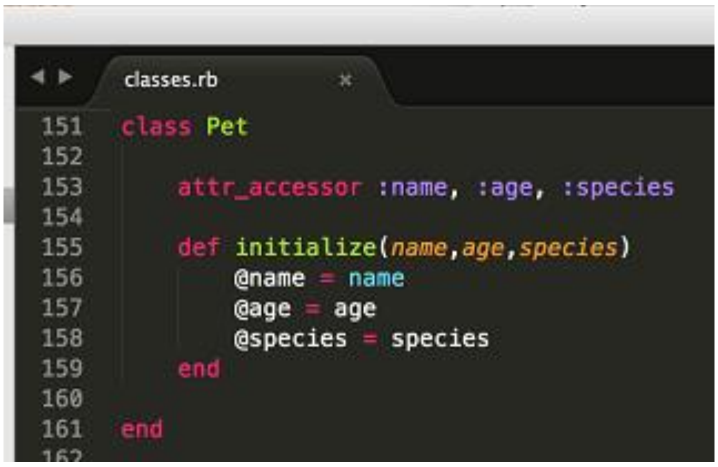
Let's spend some time making classes:



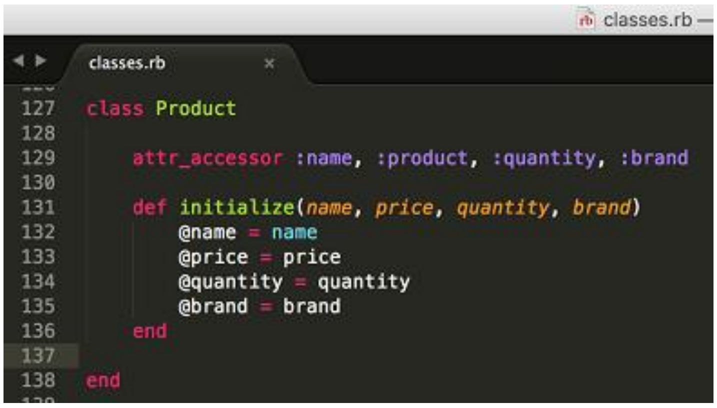
```ruby
class Person

    def initialize(name, age)
        @name = name
        @age = age
    end
end

my_profile = Person.new("Aaron", 34)
```

Exercise :

Create a **Pet** class.
Create a **Product** class.



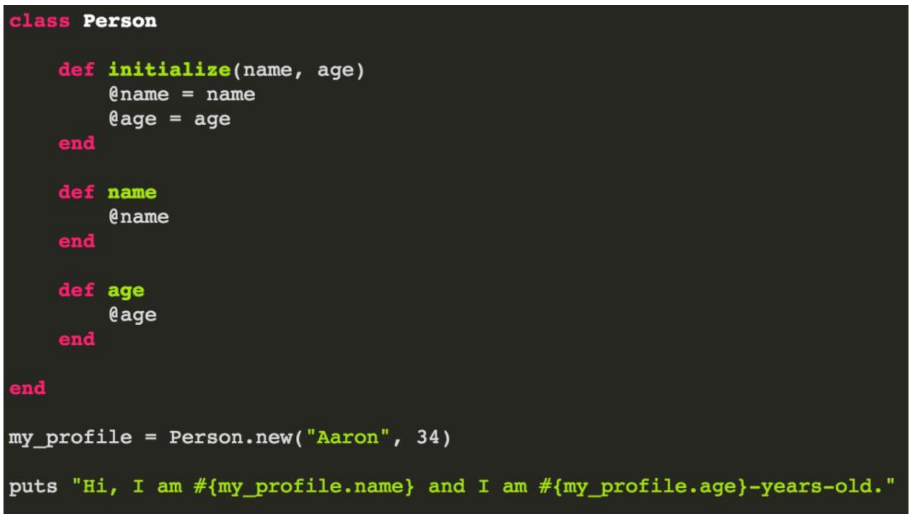
```ruby
class Pet

    attr_accessor :name, :age, :species

    def initialize(name,age,species)
        @name = name
        @age = age
        @species = species
    end

end
```



```ruby
class Product

    attr_accessor :name, :product, :quantity, :brand

    def initialize(name, price, quantity, brand)
        @name = name
        @price = price
        @quantity = quantity
        @brand = brand
    end

end
```

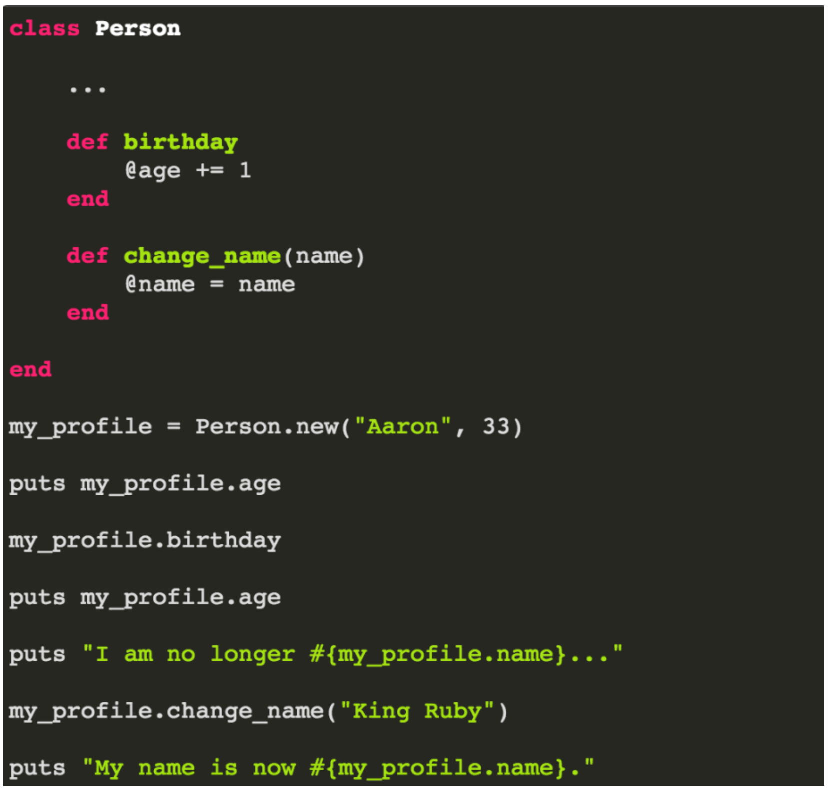
5.1. Accessing Object Data

We need a way to access the data in our saved *Object*, and there is actually two ways to go about it. The hard way is creating a method to call each instance variable.



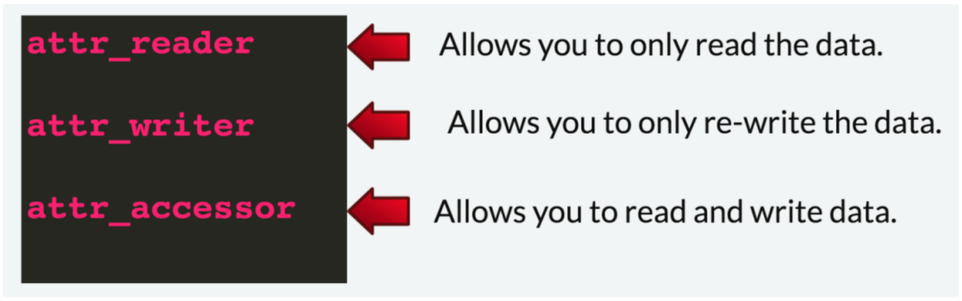
```ruby
class Person

    def initialize(name, age)
        @name = name
        @age = age
    end

    def name
        @name
    end

    def age
        @age
    end

end

my_profile = Person.new("Aaron", 34)

puts "Hi, I am #{my_profile.name} and I am #{my_profile.age}-years-old."
```

Then creating methods that let you modify the data:



```ruby
class Person

    ...

    def birthday
        @age += 1
    end

    def change_name(name)
        @name = name
    end

end

my_profile = Person.new("Aaron", 33)

puts my_profile.age

my_profile.birthday

puts my_profile.age

puts "I am no longer #{my_profile.name}..."

my_profile.change_name("King Ruby")

puts "My name is now #{my_profile.name}."
```

The easy (or, at least, DRY-er) way can be approached with three different keywords:


```
attr_reader     Allows you to only read the data.
attr_writer     Allows you to only re-write the data.
attr_accessor   Allows you to read and write data.
```

Following those keywords would be one or more of your attributes, written as symbols (same variable name, but with a colon in front).

Reader – Writer – Accessor

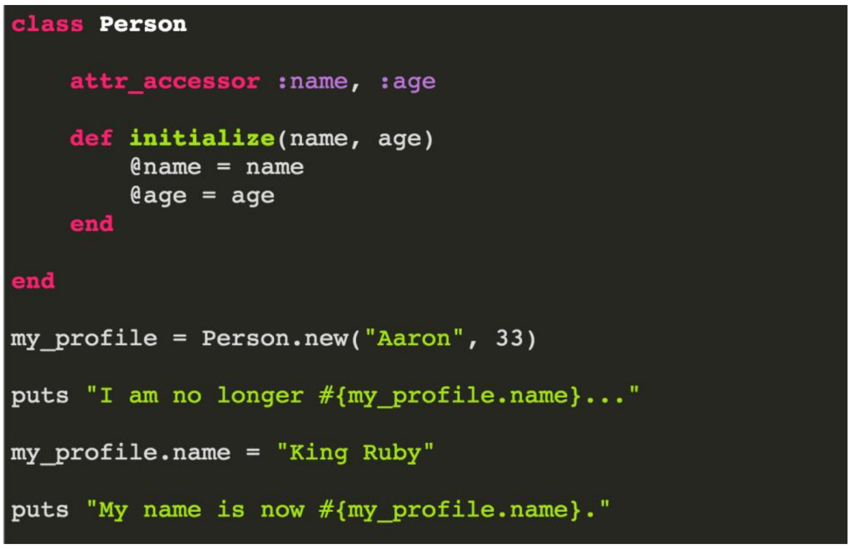
⇨ **attr_reader** would be good for attributes that you do not want changed (like ID#s).

⇨ **attr_writer** would be for some situation where you want to override data, but not read it back.

⇨ **attr_accessor** would be the combination of the other two. The power to read and the power to override.

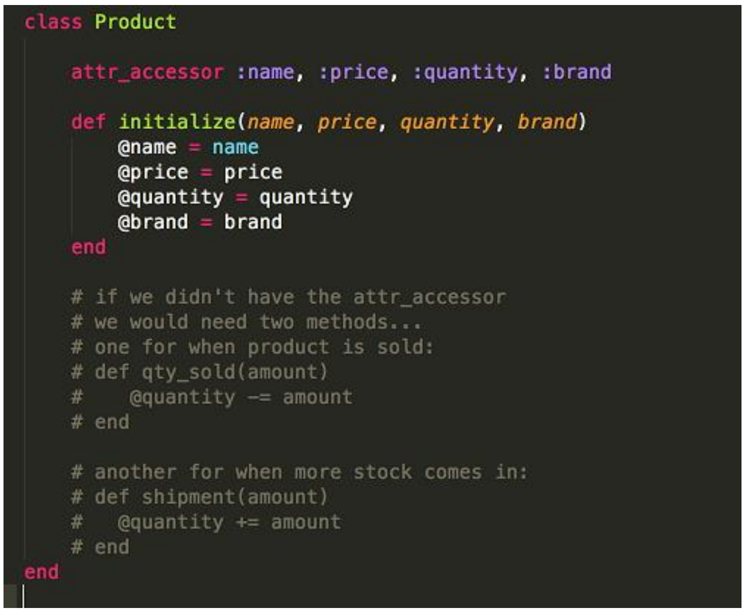
5.2. Accessor

So if we user **attr_accessor** with the person object:

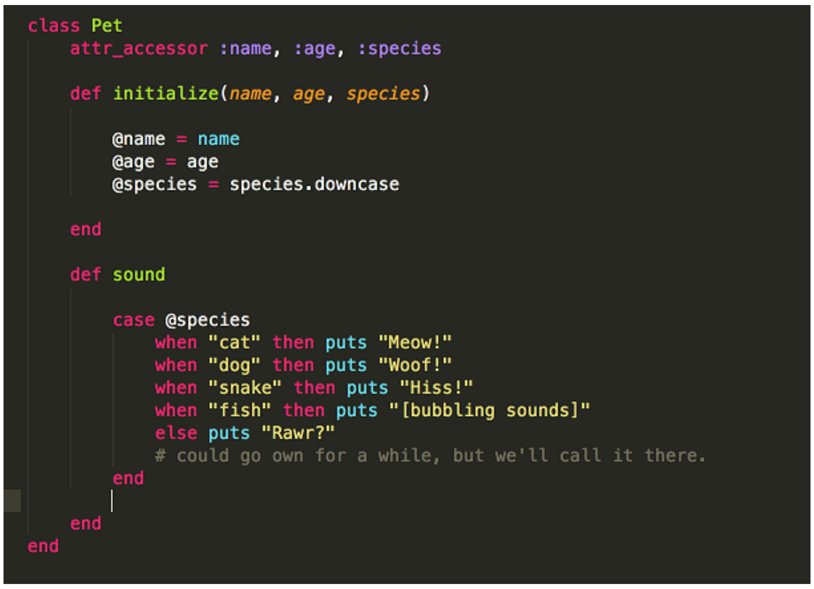

```
class Person

    attr_accessor :name, :age

    def initialize(name, age)
        @name = name
        @age = age
    end

end

my_profile = Person.new("Aaron", 33)

puts "I am no longer #{my_profile.name}..."

my_profile.name = "King Ruby"

puts "My name is now #{my_profile.name}."
```

Exercise:

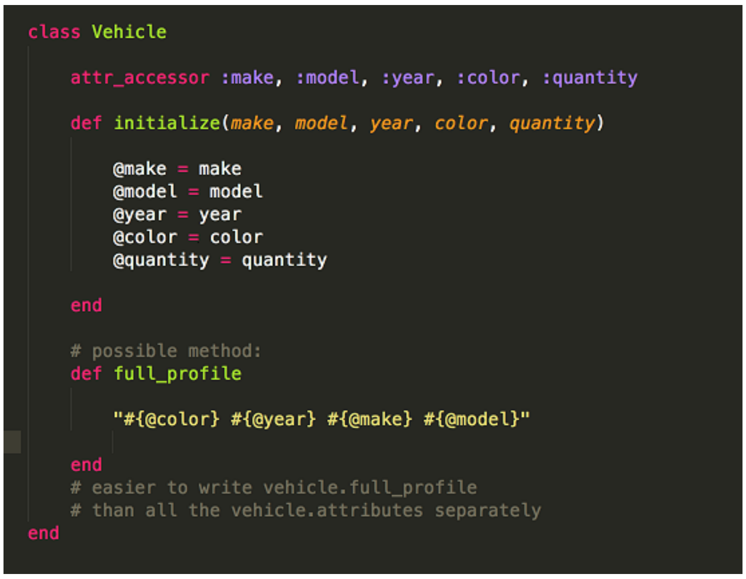
⇨ Create methods within the Product class to control quantity.
⇨ Create a method for the Pet class to return the animal’s sound.
⇨ Create a brand new class: Vehicle. What should the attributes be? What methods should we create?



```ruby
class Product

    attr_accessor :name, :price, :quantity, :brand

    def initialize(name, price, quantity, brand)
        @name = name
        @price = price
        @quantity = quantity
        @brand = brand
    end

    # if we didn't have the attr_accessor
    # we would need two methods...
    # one for when product is sold:
    # def qty_sold(amount)
    #    @quantity -= amount
    # end

    # another for when more stock comes in:
    # def shipment(amount)
    #   @quantity += amount
    # end
end
```



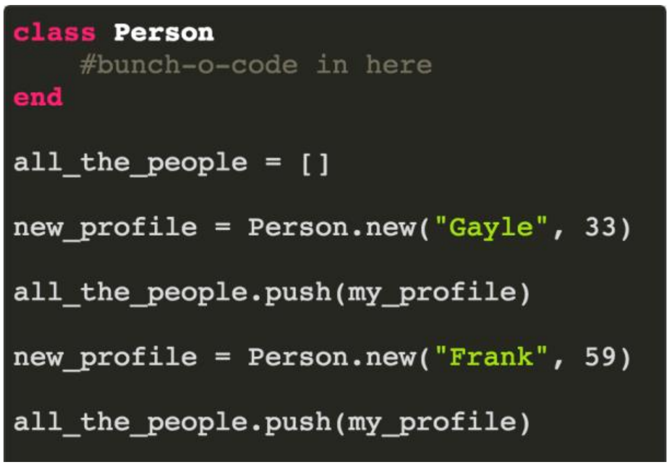
```ruby
class Pet
    attr_accessor :name, :age, :species

    def initialize(name, age, species)

        @name = name
        @age = age
        @species = species.downcase

    end

    def sound

        case @species
            when "cat" then puts "Meow!"
            when "dog" then puts "Woof!"
            when "snake" then puts "Hiss!"
            when "fish" then puts "[bubbling sounds]"
            else puts "Rawr?"
            # could go own for a while, but we'll call it there.
        end

    end
end
```



```ruby
class Vehicle

    attr_accessor :make, :model, :year, :color, :quantity

    def initialize(make, model, year, color, quantity)

        @make = make
        @model = model
        @year = year
        @color = color
        @quantity = quantity

    end

    # possible method:
    def full_profile

        "#{@color} #{@year} #{@make} #{@model}"

    end
    # easier to write vehicle.full_profile
    # than all the vehicle.attributes separately
end
```

5.3. Keep Objects Collected

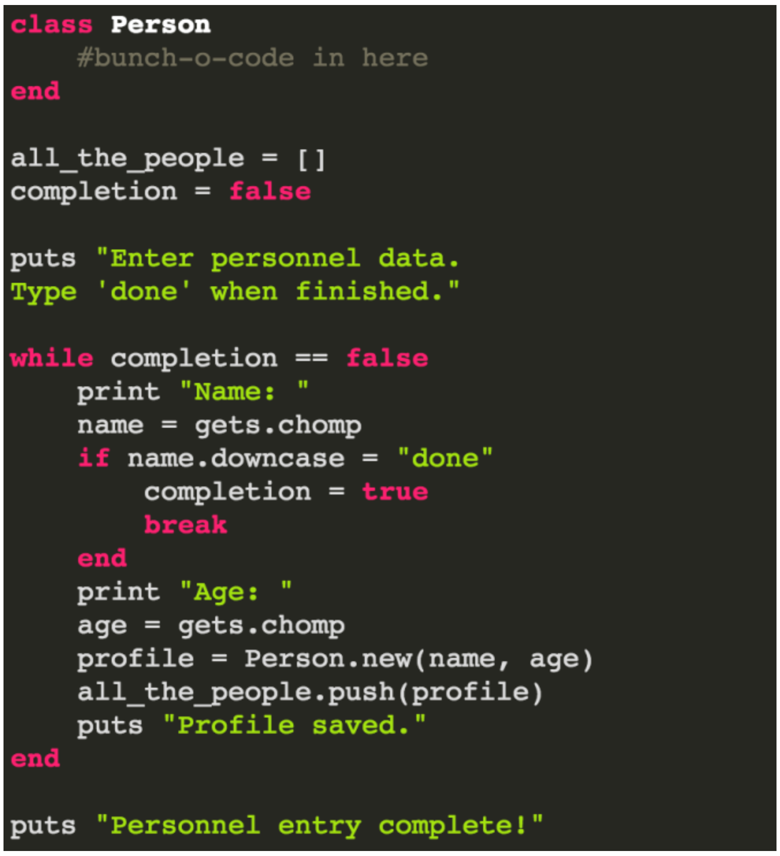
Custom class objects can be handily stored in Arrays!



```ruby
class Person
    #bunch-o-code in here
end

all_the_people = []

new_profile = Person.new("Gayle", 33)

all_the_people.push(my_profile)

new_profile = Person.new("Frank", 59)

all_the_people.push(my_profile)
```

We can continue to use the same variable to create new instances of the Person object, and then push them into an array for storage.

How could we use this approach, along with a loop, to create a collection of objects created by the user?

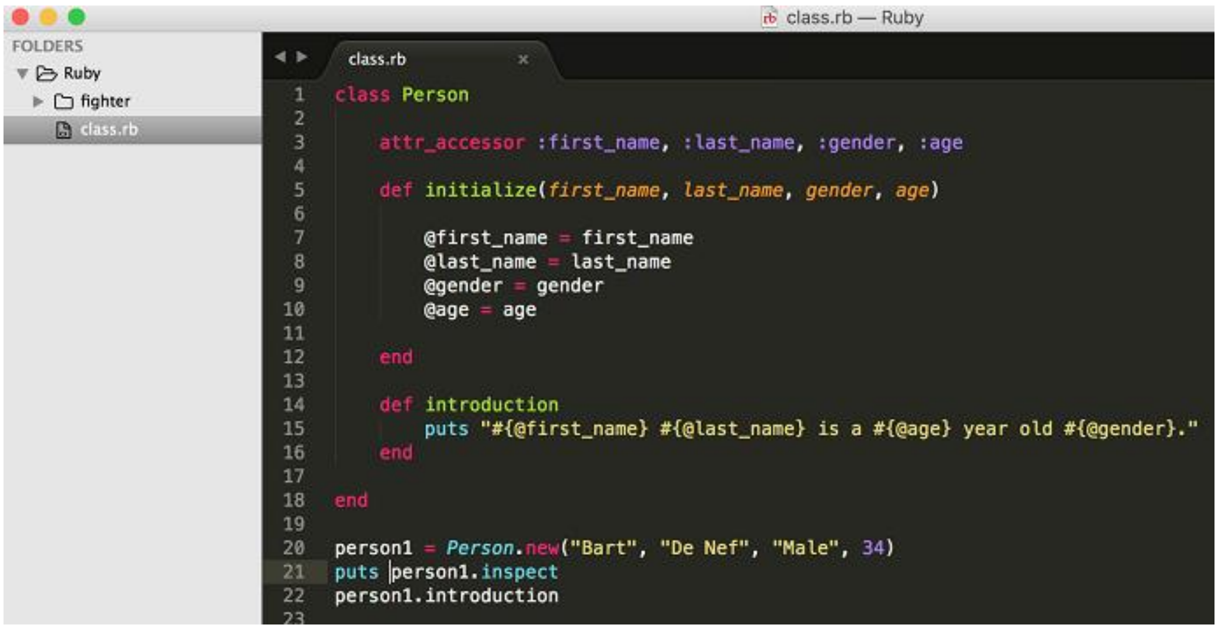


```ruby
class Person
    #bunch-o-code in here
end

all_the_people = []
completion = false

puts "Enter personnel data.
Type 'done' when finished."

while completion == false
    print "Name: "
    name = gets.chomp
    if name.downcase = "done"
        completion = true
        break
    end
    print "Age: "
    age = gets.chomp
    profile = Person.new(name, age)
    all_the_people.push(profile)
    puts "Profile saved."
end

puts "Personnel entry complete!"
```

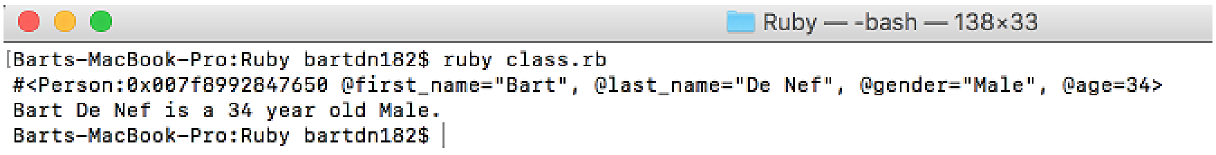
- Let the user provide data for our objects, and save the objects in an array.
- **break** will knock us back to the top of the loop without completing the rest of it.

6. Class Inheritance

Remember when we made our **Word class** and then set it to inherit all the functionality of the **String class**? We can now create some classes that inherit all the functionality of the Person class:

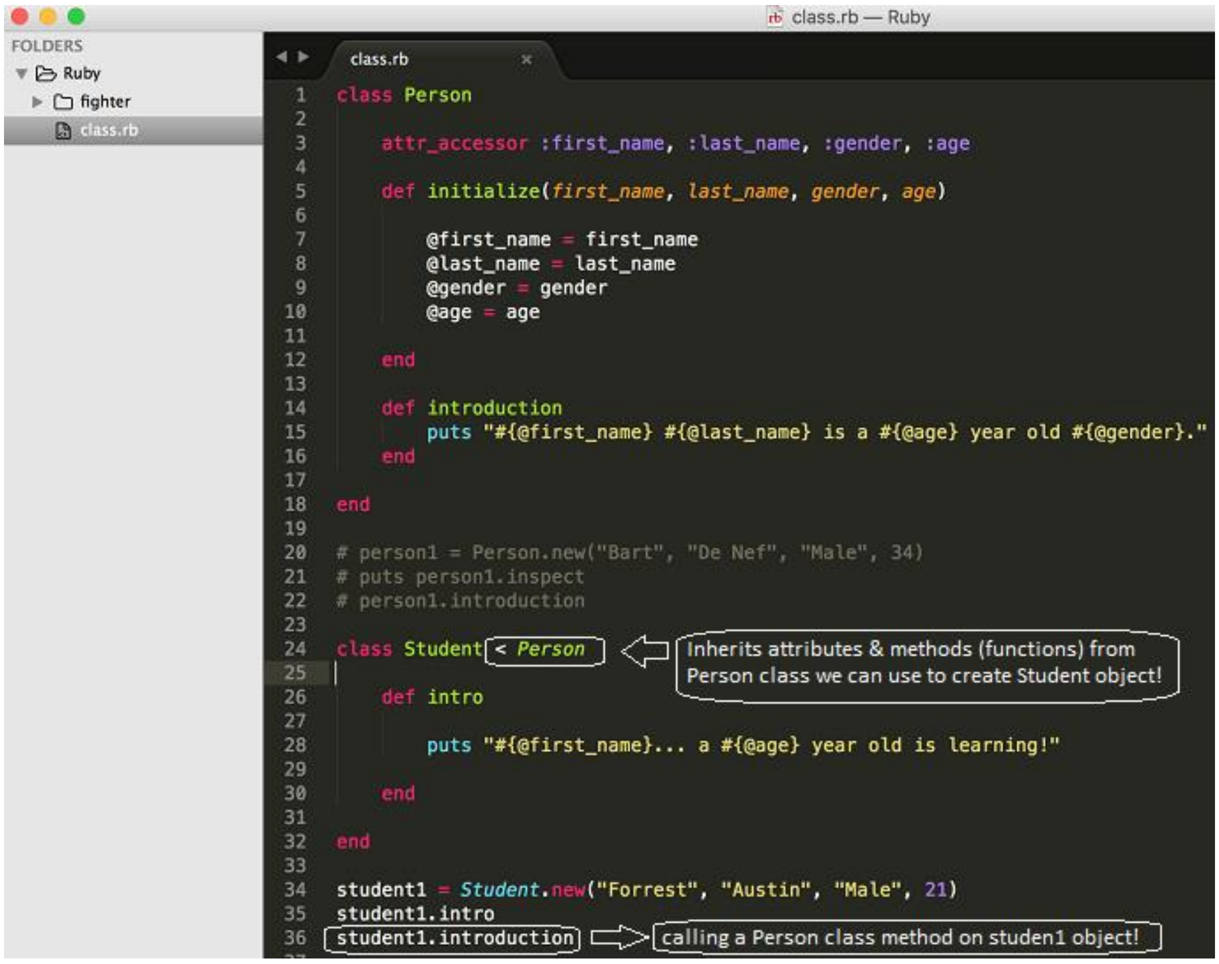


```ruby
class Person

    attr_accessor :first_name, :last_name, :gender, :age

    def initialize(first_name, last_name, gender, age)

        @first_name = first_name
        @last_name = last_name
        @gender = gender
        @age = age

    end

    def introduction
        puts "#{@first_name} #{@last_name} is a #{@age} year old #{@gender}."
    end

end

person1 = Person.new("Bart", "De Nef", "Male", 34)
puts person1.inspect
person1.introduction
```



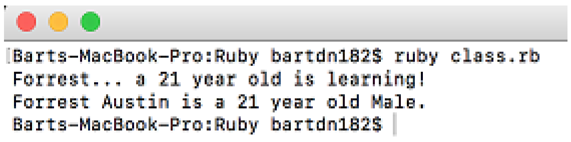
```
Barts-MacBook-Pro:Ruby bartdn182$ ruby class.rb
#<Person:0x007f8992847650 @first_name="Bart", @last_name="De Nef", @gender="Male", @age=34>
Bart De Nef is a 34 year old Male.
Barts-MacBook-Pro:Ruby bartdn182$
```

Create now a Student class in the same file as the Person class!



```ruby
class Person

  attr_accessor :first_name, :last_name, :gender, :age

  def initialize(first_name, last_name, gender, age)

    @first_name = first_name
    @last_name = last_name
    @gender = gender
    @age = age

  end

  def introduction
    puts "#{@first_name} #{@last_name} is a #{@age} year old #{@gender}."
  end

end

# person1 = Person.new("Bart", "De Nef", "Male", 34)
# puts person1.inspect
# person1.introduction

class Student < Person


  def intro

    puts "#{@first_name}... a #{@age} year old is learning!"

  end

end

student1 = Student.new("Forrest", "Austin", "Male", 21)
student1.intro
student1.introduction
```

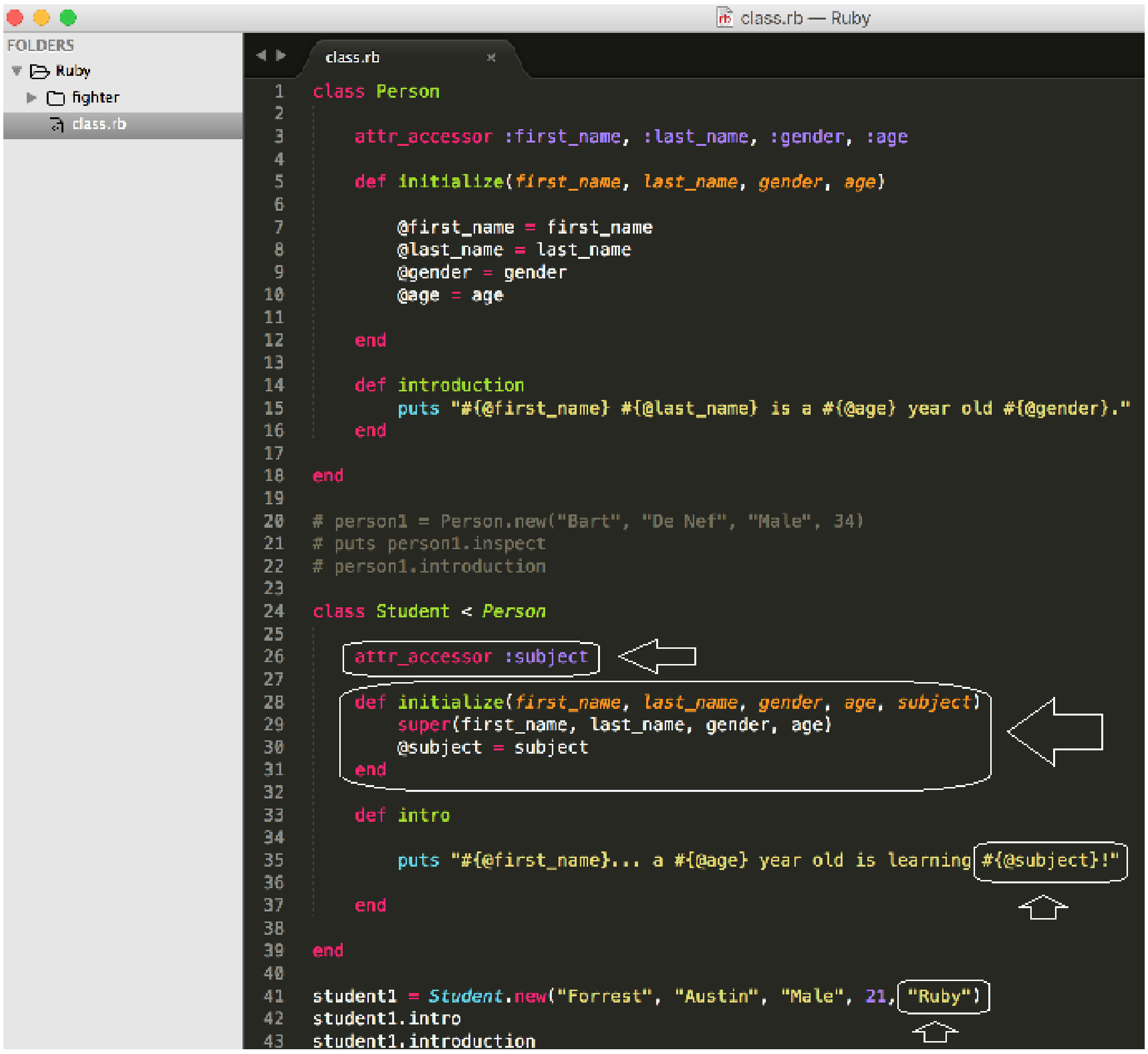
class.rb — Ruby
FOLDERS
Ruby
fighter
class.rb
Inherits attributes & methods (functions) from Person class we can use to create Student object!
calling a Person class method on studen1 object!



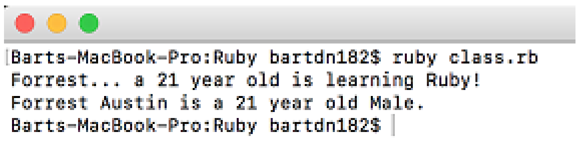
```
Barts-MacBook-Pro:Ruby bartdn182$ ruby class.rb
Forrest... a 21 year old is learning!
Forrest Austin is a 21 year old Male.
Barts-MacBook-Pro:Ruby bartdn182$
```

⇨ So you see that you can call the **.intro** function on your new **Student class**, but you can ALSO call the **.introduction** function, since the **Student class** inherited all the attributes and functionality of the **Person class**. Feel free to remove the **< Person** to prove this to yourself.

Let’s add a subject to the student what he is learning!



```ruby
class Person

    attr_accessor :first_name, :last_name, :gender, :age

    def initialize(first_name, last_name, gender, age)

        @first_name = first_name
        @last_name = last_name
        @gender = gender
        @age = age

    end

    def introduction
        puts "#{@first_name} #{@last_name} is a #{@age} year old #{@gender}."
    end

end

# person1 = Person.new("Bart", "De Nef", "Male", 34)
# puts person1.inspect
# person1.introduction

class Student < Person

    attr_accessor :subject

    def initialize(first_name, last_name, gender, age, subject)
        super(first_name, last_name, gender, age)
        @subject = subject
    end

    def intro

        puts "#{@first_name}... a #{@age} year old is learning #{@subject}!"

    end

end

student1 = Student.new("Forrest", "Austin", "Male", 21, "Ruby")
student1.intro
student1.introduction
```

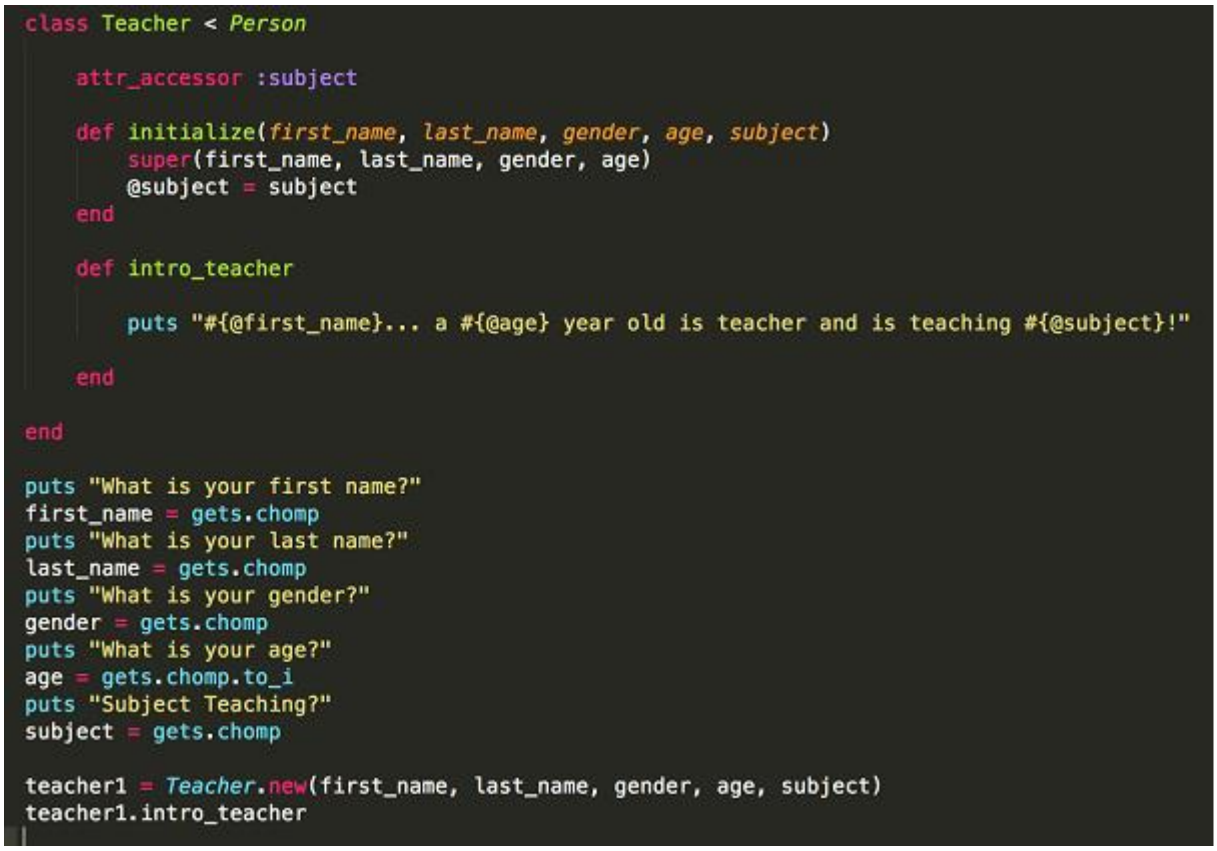


```
Barts-MacBook-Pro:Ruby bartdn182$ ruby class.rb
Forrest... a 21 year old is learning Ruby!
Forrest Austin is a 21 year old Male.
Barts-MacBook-Pro:Ruby bartdn182$
```

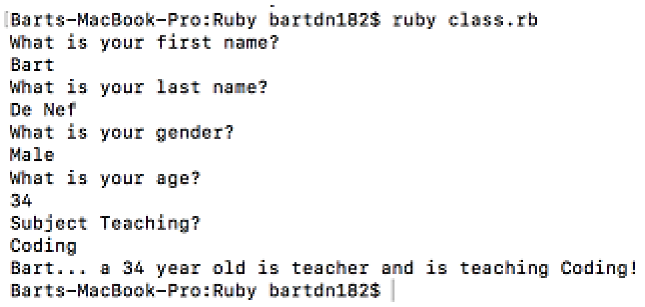
⇨ With the **super** method we don't have to rewrite our instance variables from **Person class**.

Exercise:

Create another class called **Teacher** in that inherits from the **Person class**. Create a function and test it! Create this in the same file as the **Person class**.



```ruby
class Teacher < Person

    attr_accessor :subject

    def initialize(first_name, last_name, gender, age, subject)
        super(first_name, last_name, gender, age)
        @subject = subject
    end

    def intro_teacher

        puts "#{@first_name}... a #{@age} year old is teacher and is teaching #{@subject}!"

    end

end

puts "What is your first name?"
first_name = gets.chomp
puts "What is your last name?"
last_name = gets.chomp
puts "What is your gender?"
gender = gets.chomp
puts "What is your age?"
age = gets.chomp.to_i
puts "Subject Teaching?"
subject = gets.chomp

teacher1 = Teacher.new(first_name, last_name, gender, age, subject)
teacher1.intro_teacher
```



```
Barts-MacBook-Pro:Ruby bartdn182$ ruby class.rb
What is your first name?
Bart
What is your last name?
De Nef
What is your gender?
Male
What is your age?
34
Subject Teaching?
Coding
Bart... a 34 year old is teacher and is teaching Coding!
Barts-MacBook-Pro:Ruby bartdn182$
```