**Points to Ponder**

*Look through these now and then use them to test yourself after doing the assignment*

**Note:** This is again a long list of things to understand! Read through the list, breathe deep, do the assignment, and then come back to them.

* Classes and Methods:
  + What is an implicit return?
  + Every block in ruby will return the value of the last line automatically, so it's common to not use thereturn keyword in favor of minimal code (specially if the method fits in one line):
  + def extract\_user\_ids(users)
  + user.map(&:id)
  + end
  + # is the same as
  + def extract\_user\_ids(users)
  + return user.map(&:id)
  + end
  + What is a class?
  + When you find that you want the same method to be run on a bunch of different objects without having to make a bunch of different if/else or case statements, you should start thinking about using a class.
  + When should you use a class?
  + We've been over storing data in hashes, but what happens when you want to treat that data like a real object and make it move? Or if you want to handle 10,000 different instances of it? When you just store your Viking's name, age, health, and strength, it just kind of sits there. What about when you want to make an army of Vikings who can do stuff like #eat, #travel, #sleep and, of course, #attack? For that, you need a slightly more complex structure to make your Viking out of, so you give it its own Viking class:
  + How do you create a class in your script file?
  + class Viking
  + # put your methods and variables here
  + end
  + What is an instance of a class?
  + To be able to amass a horde of 100 Vikings, you need a way to create new ones. Each time you do that, it's called **Instantiating** a newViking. You use a special ::new method to do that. You've done it many times for the Array class already:
  + > my\_arr = Array.new
  + => []
  + What is the difference between the CamelCase and snake\_case styles of naming?
  + Note that the class name is always capitalized and, for multiple words, uses **CamelCase** (capitalized with no spaces) not the **snake\_case**(lowercase with spaces as underscores) you've typically seen for variables.
  + How do you instantiate a class?
  + ::new is a **Class Method**, which means that you call it on the class (Array here) and not the specific instance of that class (which would be my\_arr here). It's also why we designate it with the two colons ::when talking about it here.
  + How do you set the state of your new instance?
  + If you want to give your viking some actions it can do, give it some methods. Since these methods get called on an individual instance of the Viking class, they're called **Instance Methods**, just the same as all your old friends like #each and #sort and #max etc. We just usually don't call them "instance" methods so maybe it wasn't obvious.
  + class Viking
  + def initialize(name, age, health, strength)
  + # codecodecode
  + end
  + def attack(enemy)
  + # code to fight
  + end
  + end
  + What should be done in the #initialize method?
  + Classes share their methods, but what about variables? You don't want all your Vikings to have the same strength, so we use **instance variables** to take care of that. You designate an instance variable using the @variable\_name notation, and you'll be able to use it the same wayfor every instance of Viking but it will have a unique value for each. These instance variables are part of setting up your object's state. When your instance is destroyed, you lose access to its instance variables as well. You'd usually set them up in your #initialize method:
  + class Viking
  + def initialize(name, age, health, strength)
  + @name = name
  + @age = age
  + @health = health
  + @strength = strength
  + end
  + end
  + > oleg = Viking.new("Oleg", 19, 100, 8)
  + => #<Viking:0x007ffc0597bae0>
  + What is a class method?
  + What about class methods? You define a class method by preceding its name with self (e.g. def self.class\_method) or, identically, just the name of the class (e.g. def Viking.class\_method). There's a less common method that puts the line class << self ahead of your method definitions (which won't use self anymore).
  + How is a class method different from an instance method?
  + Class methods have access to other class methods and class variables but don't have access to instance methods or instance variables
  + Instance methods can call other instance methods, instance variables, class methods, or class variables
  + How are methods you already know like #count or #sort etc instance methods?
  + They work with the instance variables
  + What's the difference between how you declare a class method vs an instance method?
  + You define a class method by preceding its name with self (e.g. def self.class\_method) or, identically, just the name of the class (e.g. def Viking.class\_method). There's a less common method that puts the line class << self ahead of your method definitions (which won't use self anymore).
  + What's the difference between how you call a class method vs an instance method?
  + class Foo
  + def self.bar
  + puts 'class method'
  + end
  + def baz
  + puts 'instance method'
  + end
  + end
  + Foo.bar # => "class method"
  + Foo.baz # => NoMethodError: undefined method ‘baz’ for Foo:Class
  + Foo.new.baz # => instance method
  + Foo.new.bar # => NoMethodError: undefined method ‘bar’ for #<Foo:0x1e820>
  + See the difference? bar is a **class method**, so calling bar on the Foo class works fine. baz is an **instance method**, so calling baz on the Foo class raises a NoMethodError. Then, on the next couple lines, we call both methods on an **instance** of Foo (Foo.new).
  + The key difference is instance methods only work with an instance and thus you have to create a new instance to use them (Foo.new).
  + What is an instance variable?
  + Classes share their methods, but what about variables? You don't want all your Vikings to have the same strength, so we use **instance variables** to take care of that. You designate an instance variable using the @variable\_name notation, and you'll be able to use it the same way for every instance of Viking but it will have a unique value for each. These instance variables are part of setting up your object's state. When your instance is destroyed, you lose access to its instance variables as well. You'd usually set them up in your #initialize method:

class Viking

def initialize(name, age, health, strength)

@name = name

@age = age

@health = health

@strength = strength

end

end

> oleg = Viking.new("Oleg", 19, 100, 8)

=> #<Viking:0x007ffc0597bae0>

* + What's the difference between an instance variable and a 'regular' variable?
  + The instance variables are a part of oleg but you can't access them from outside him because it's really nobody's business but his. So you have to create a method specifically to get that variable, called a **getter** method, and just name it the same thing as the variable you want:
    - def health
    - @health
    - end
    - > oleg.health
    - => 87
  + What are "getter" and "setter" methods used for?
  + To access instance variables
  + What is the difference between a "getter" and a "setter" method?
  + To get and set instance variables
  + How do you make instance variables readable outside your class? Writeable? Both at the same time?
  + Well, you can imagine that you'll probably be writing a whole lot of those methods, so Ruby gives you a helper method calledattr\_accessor, which will create those getters and setters for you. Just pass it the symbols for whatever variables you want to make accessible and POOF! those methods will now exist for you to use:
  + class Viking
  + attr\_accessor :name, :age, :health, :strength
  + # codecodecode
  + end
  + attr\_accessor isn't magical, it just uses Ruby's ability to create methods from within your script (part of "metaprogramming") to set up #name and #name=(new\_name) and #age and #age=(new\_age) etc.
  + You shouldn't make anything readable and certainly not writeable without a good reason. If you only want one or the other, Ruby gives you the similar attr\_reader and attr\_writer methods. They should be pretty self explanatory.
  + Can a class call its own class methods?

|  |  |
| --- | --- |
|  | That is correct. self inside a class method is the class itself. (And also inside the class definition, such as the self in def self.coolpost.)  You can easily test these tidbits with irb:  class Foo  def self.bar  puts self.inspect  end  end  Foo.bar # => Foo |

* + What's the difference between when you would use a class variable and when you would use a constant?
  + If you're thinking that class variables seem pretty similar to constants, they are only similar in that all instances have access to them. If you've got something that will never, CAN never change, use a constant. If you might ever change it, stick with a class variable. At the very least, it makes your code much more legible.
  + What's the difference between a class and a module?
  + Basically, a class can be instantiated but a module cannot.
  + When would you use a class but not a module?
  + A module will never be anything other than a library of methods. A class can be so much more -- it can hold its state (by keeping track of instance variables) and be duplicated as many times as you want. It's all about objects. If you need to instantiate something or otherwise have it exist over time, that's when you need to use a class instead of a module.
  + How does inheritance work?
  + **Inheritance** is the ability of one class to be a child of another class and therefore inherit all its characteristics, including methods and variables.
  + Why is inheritance really useful?
  + Why do all this inheritance? To keep our code as DRY as possible. It lets us not have to repeat a bunch of methods (say, #to\_s, which is implemented in the Object class) for every different subclass.
  + How do you extend a class? What does that mean?
  + In Ruby, a class inherits from another class using the < notation. Unlike some other languages, a class can only have ONE parent.
  + class Viking < Person
  + Now Viking has access to all of Person's methods. You say thatViking **Extends** Person.
  + What does #super do? Why use it?
  + #super lets you call the superclass's version of your current method.
  + class Viking < Person
  + ...
  + def heal
  + 2.times { super }
  + puts "Ready for battle!"
  + end
  + end
* Scope:
  + What is scope?
  + **Scope** is the formal term that represents when you can access a variable or method and when you can't. It's nothing explicit in the code (you're never calling a method named scope or anything like that); it's just a concept. If your variable is "in scope" then it's available for use, otherwise it's "out of scope".
  + When can you start using a variable?
  + When you’re in the scope of the variable
  + When is a new scope defined?
  + **A new scope** is created when you first define a variable. That variable is then accessible by anything "downstream" of it in the code, until the current scope is exited (by leaving a method or loop, for instance):
  + def launch\_longships(longships)
  + # Here we can't yet use `longship`, `longships\_count` or `longship\_name`. We CAN use `longships` because it was passed in above this point.
  + launched\_ships = 0
  + # Now launched\_ships is in scope so we can use it
  + longships.each do |longship|
  + # Now `longship` is in scope, so we can use it
  + longship\_name = "#{longship.owner.name}'s Reaver"
  + # Now `longship\_name` is in scope so we can use it
  + longship.launch
  + launched\_ships += 1
  + puts "#{longship\_name} successfully launched!"
  + end
  + # Now we've exited the loop so `longship` and `longship\_name` are no longer in scope so we cannot use them.
  + puts "Excellent news! We've launched #{launched\_ships} ships!"
  + end
  + A good rule of thumb for scope is that you create a new scope any time you should indent your code and any time within that indent a new variable is defined
  + When are methods in scope?
  + Method scope is similar but has some important differences because it deals much more explicitly with the notion of **privacy**. You still can't call a method until the Ruby interpreter has had the chance to define it. By default, instance methods can be called by any instance of a class (e.g. oleg.sleep) and class methods can be called directly on the class itself (e.g. Viking.new)
  + What is a private method?
  + If you create methods that should only be accessible by other methods within your class, make them private. This is the default setting for instance variables unless you expose them using the afore-mentionedattr\_accessor.
  + What is a protected method?
  + Since we don't want #take\_damage to be visible to anyone on the command line but we DO want it to be visible to the methods inside other instances of Viking, we call that **protected**.
  + How are private and protected methods different?
  + protected provides most of the privacy of private but lets the methods inside other instances of the same class or its descendents also access it:
  + What does "encapsulation" mean?
  + Encapsulation means that the internal representation of an object is hidden from the outside. Only the object can interact with its internal data. Public methods can be created to open a defined way to access the logic inside an object.