Exercise: Modules, Namespaces, and Mixins

CS 132A Ruby Programming, Week 14

Topics

1. **Reading**
2. **Overview**
3. **Ruby classes review**
4. **Scope**
   - **The scope of Constants**
   - **Local scope**
   - **Global scope**
   - **Examples: using global variables**
   - **Object variables**
   - **Class variables**
5. **Class methods**
6. **Nested classes**
7. **Mixins**
8. TODO **Exercises**
   - TODO **Create a module named** `my_math.rb`
   - TODO **Create a module named** `shape.rb`
   - TODO **Create nested classes**
   - TODO **create a script named** `test_shape.rb` **that;**

9. TODO **Turn in your work for credit.**

1. Reading

Beginning Ruby, Chapter 6 — Classes, Objects, and Modules

2. Overview

We've already learned that Ruby is about as object-oriented as a language can be. However, up to this point almost all of the programming we've done has been procedural. Over the next few weeks we'll look at Ruby's object-oriented features in more depth, beginning with Modules, Namespaces, and Mixins.

3. Ruby classes review

- *Classes:* A class is a collection of methods and data that are used as a blueprint to create multiple objects relating to that class.
- *Objects:* An object is a single instance of a class. An object of class Person is a single person. An object of class Dog is a single dog. If you think of objects as real-life objects, a class is the classification, whereas an object is the actual object or "thing" itself.
- *Local variable:* A variable that can only be accessed and used from the current scope.
- *Instance/object variable:* A variable that can be accessed and used from the scope of a single object. An object's methods can all access that object's object variables.
- *Global variable:* A variable that can be accessed and used from anywhere within the current program.
- *Class variable:* A variable that can be accessed and used within the

> scope of a class and all of its child objects.
>
> - *Encapsulation:* The concept of allowing methods to have differing degrees of visibility outside of their class or associated object.
> - *Polymorphism:* The concept of methods being able to deal with different classes of data and offering a more generic implementation (as with the area and perimeter methods offered by your Square and Triangle classes).
> - *Module:* An organizational element that collects together any number of classes, methods, and constants into a single namespace.
> - *Namespace:* A named element of organization that keeps classes, methods, and constants from clashing.
> - *Mix-in:* A module that can mix its methods in to a class to extend that class's functionality.
> - *Enumerable:* A mix-in module, provided as standard with Ruby, that implements iterators and list-related methods for other classes, such as collect, map, min, and max. Ruby uses this module by default with the Array and Hash classes.
> - *Comparable:* A mix-in module, provided as standard with Ruby, that implements comparison operators (such as <, >, and ==) on classes that implement the generic
>
> —Beginning Ruby, p. 158

A few more things to consider:

- Ruby classes can be modified while the program in running, and any changes to the class are available to all existing instances.
- Ruby has *single-inheritance*, meaning that a class can have only one super class.
- Classes can *include* modules; the methods and attributes defined in the module become part of the including class. Mixins allow Ruby classes to mimic multiple inheritance.

4. Scope

The scope (https://en.wikipedia.org/wiki/Scope_(computer_science)#Definition) of a variable describes exactly where, in your program, a certain entity is “bound” to a name. In other words, a name be attached to one value in a function and to another value inside a class. Ruby has four scopes: *local*, *global*, *object*, and *class*.

4.1 The scope of Constants

Constants have slightly different rules than global variables. Where global variables are visible everywhere, no matter where they’re created, Ruby Constants are defined in the scope of the current class and visible in any child classes.

Ruby Constants must always begin with an upper-case alphabetical character, such as `Pi`.

```ruby
module MyMath
  IndianaPi = 3.2  # Indiana Pi definition: see https://en.wikipedia.org/wi
ki/Indiana_Pi_Bill
  Pi = 3.14159265358979323846

  def print_pi
    puts "Real Pi: " + Pi.to_s
  end

  def print_indiana_pi
    puts "Indiana Pi: " + IndianaPi.to_s
  end

end
```

Ruby

```ruby
# Add the current directory to the Ruby Search Path: $:
$:.push('.')
# Require my_math.rb
require 'my_math'
# Include the module into the main scope
include MyMath

# Call functions from my_math.rb
print_pi
print_indiana_pi
puts MyMath::Pi
puts MyMath::IndianaPi

class MyMathClass
  # Mixin the attributes of the MyMath module
  include MyMath
end

puts MyMathClass::Pi
puts MyMathClass::IndianaPi
m = MyMathClass.new
m.print_pi
m.print_indiana_pi
```

```
Real Pi: 3.141592653589793
Indiana Pi: 3.2
3.141592653589793
3.2
3.141592653589793
3.2
Real Pi: 3.141592653589793
Indiana Pi: 3.2
```

4.2 Local scope

(p. 119)

Local variables are used in the place where they're defined. A *place* would in a function or method, or in block of code. The advantage of local variables is that they don't clash with local variables of the same name in other scopes.

```ruby
# main scope
color = 'reddish blue'
def mycolor(somecolor)
  # local scope
  color = somecolor
  puts "In the local scope of the mycolor function, color is \"#{color}\"."
end
mycolor('bluish gray')
puts "In the main scope, color is \"#{color}\"."
```

```
In the local scope of the mycolor function, color is "bluish gray".
In the main scope, color is "reddish blue".
```

4.3 Global scope

(p. 121)

Global are visible everywhere. Globals begin with a `$`. Ruby has quite a view built-in global variables, but so far we've used only one: `$:`, the directory search path. Global variables aren't used much in Ruby, as global variables are a poor fit with the object oriented programming paradigm. Although there are no Variable Police enforcing the use of global variables, you really don't want to use them in your programs. You can view a complete list of Ruby's global variables in `Kernel#global_variables` using IRB.

```ruby
$ irb
>> Kernel.global_variables
=> [:$;, :$-F, :$@, :$!, :$SAFE, :$~, :$&, :$`, :$', :$+, :$=, :$KCODE, :$-
K, :$,,
:$/, :$-0, :$\, :$_, :$stdin, :$stdout, :$stderr, :$>, :$<, :$., :$FILENAME
, :$-i,
:$*, :$:, :$-I, :$LOAD_PATH, :$", :$LOADED_FEATURES, :$?, :$$, :$VERBOSE, :
$-v,
:$-w, :$-W, :$DEBUG, :$-d, :$0, :$PROGRAM_NAME, :$-p, :$-l, :$-a, :$binding
,
:$1, :$2, :$3, :$4, :$5, :$6, :$7, :$8, :$9]
```

Many of these variables were adopted from Perl or from Shell.

Variable	Value
`$!`	latest error message
`$@`	location of error
`$_`	string last read by gets
`$.`	line number last read by interpreter
`$&`	string last matched by regexp
`$~`	the last regexp match, as an array of subexpressions
`$n`	the nth subexpression in the last match (same as $~[n])
`$=`	case-insensitivity flag
`$/`	input record separator
`$\`	output record separator
`$0`	the name of the ruby script file
`$*`	the command line arguments
`$$`	interpreter's process ID
`$?`	exit status of last executed child process

4.4 Examples: using global variables

Ruby

```
#!/usr/local/bin/ruby
# File: globals.rb

# Print the process ID.
puts "The sub-shell process ID is #{$$}."

# Match a string to a regex with subexpressions}."
"this is a string" =~ /(this) (is) (a) (string)/

# Print the subexpression matches in reverse order
puts "The sub-expression matches, in reverse order: #{$4}, #{$3}, #{$2}, #{
$1}"

# Print the name of the ruby script file
puts "The name of the script is: #{$0}."
```

```
$ ruby globals.rb
The sub-shell process ID is 9906.
The sub-expression matches, in reverse order: string, a, is, this
The name of the script is: globals.rb.
```

4.5 Object variables

(p. 121)

Object variables are accessible inside of an object. Object variables start with `@`. Each instance of a class has the same variables, but their values are only accessible within the object. For example

Ruby

```
class Shape
  def initialize(x,y)
    @x = x
    @y = y
  end

  def to_s
    self.class.to_s + " at #{@x}x#{@y}"
  end
end

class Circle < Shape
  def initialize(radius,x,y)
    @radius = radius
    # Initialize x and y in parent class
    super(x,y)
  end
  def to_s
    super.to_s + ", radius #{@radius}"
  end
end

c = Circle.new(10, 0, 0)
d = Circle.new(100, 100, 100)
puts c
puts d
```

```
Circle at 0x0, radius 10
Circle at 100x100, radius 100
```

The example show that the values for `@x` , `@y` , and `@radius` are restricted to the objects in which they are defined.

4.6 Class variables

(p. 122)

Class variables have a scope that makes them visible everywhere inside the class, including inside methods. Class variable names begin with `@@` . Class variables are useful for storing data that must be accessible to all instances of a class.

The `Student` class in `week7.cgi` used a class variable named `@@count` to keep track of the number of student objects:

Ruby

```
# The Student class
class Student

  # A class variable to count the number of students
  @@count = 0;

  # Create the accessors all at once.
  attr_accessor :number,:user_name, :password, :uid, :gid, :gcos_field,:hom
e_directory, :login_shell, :number
  def initialize(data)
    # Give each student a @number; increase @@count by 1
    @number = @@count += 1;

    # Use parallel assignment to an array to a list to initialize each obje
ct

    @user_name,@password,@uid,@gid,@gcos_field,@home_directory,@login_shell
  = data
  end

  def to_s
"
#:       : #{@number}
User name: #{@user_name}
Password : N/A
UserID   : #{@uid}
GroupID  : #{@gid}
GCOS     : #{@gcos_field}
HomeDir  : #{@home_directory}
Shell    : #{login_shell}
"
  end
end

s1 = Student.new(['joe','secret',1,11,"Joe Smith, cool dude!", '/home/joe',
 '/bin/bash'])
s2 = Student.new(['tom','topsecret',2,22,"Tom Brown, awesome guy!", '/home/
tom', '/bin/zsh'])
puts s1
puts s2
```

```
#:        : 1
User name: joe
Password : N/A
UserID   : 1
GroupID  : 11
GCOS     : Joe Smith, cool dude!
HomeDir  : /home/joe
Shell    : /bin/bash

#:        : 2
User name: tom
Password : N/A
UserID   : 2
GroupID  : 22
GCOS     : Tom Brown, awesome guy!
HomeDir  : /home/tom
Shell    : /bin/zsh
```

5. Class methods

Class methods belong to the class and can be invoked directly, without creating an instance:

```ruby
Ruby
class ClassMethod
  # This is an object method
  def obj_method
    "To run me,  you need to create an instance."
  end

  # This is a class method
  def self.klass_method
    puts "I'm a class method. You can run me without creating an instance."
    self.methods.sort
  end
end

print ClassMethod::klass_method
```

```
I'm a class method. You can run me without creating an instance.
[:!, :!=, :!~, :<, :<=, :<=>, :==, :===, :=~, :>, :>=, :__id__, :__send__,
:allocate, :ancestors, :autoload, :autoload?, :class, :class_eval,
:class_exec, :class_variable_defined?, :class_variable_get, :class_variable
_set,
:class_variables, :clone, :const_defined?, :const_get, :const_missing,
:const_set, :constants, :define_singleton_method, :display, :dup, :enum_for
,
:eql?, :equal?, :extend, :freeze, :frozen?, :hash, :include, :include?,
:included_modules, :inspect, :instance_eval, :instance_exec, :instance_meth
od,
:instance_methods, :instance_of?, :instance_variable_defined?, :instance_va
riable_get,
:instance_variable_set, :instance_variables, :is_a?, :itself, :kind_of?,
:klass_method, <<<<<<<<< There it is
:method, :method_defined?, :methods, :module_eval, :module_exec, :name, :ne
w,
:nil?, :object_id, :prepend, :private_class_method, :private_constant,
:private_instance_methods, :private_method_defined?, :private_methods,
:protected_instance_methods, :protected_method_defined?, :protected_methods
,
:public_class_method, :public_constant, :public_instance_method, :public_in
stance_methods,
:public_method, :public_method_defined?, :public_methods, :public_send, :re
move_class_variable,
:remove_instance_variable, :respond_to?, :send, :singleton_class, :singleto
n_class?, :singleton_method,
:singleton_methods, :superclass, :taint, :tainted?, :tap, :to_enum, :to_s,
:trust,
:untaint, :untrust, :untrusted?]
```

If you try to run `s1.klass_method` you will get an error. Instances only have access to class methods and variables inside of the class, that is to say, in the *lexical* scope of the class.

```
class_vars.rb:36:in `<main>': undefined method `klass_method' for #<Student
:0x00000001e24fb8> (NoMethodError)
```

6. Nested classes

Ruby allows us to define classes within classes. This facility is useful when a class is dependent on other classes.

```ruby
class Shape
  def initialize(x,y)
    @x = x
    @y = y
  end

  def location
    [@x, @y]
  end

  def to_s
    self.class.to_s + " #{@x}x#{@y}"
  end

  class Circle < Shape
    attr_accessor :radius
    def initialize(radius,x,y)
      @radius = radius
      super(x,y)
    end

  end

  def create_circle(radius, x, y)
    Circle.new(radius, x, y)
  end
end

c = Shape::Circle.new(100,100,100)
x,y = c.location
puts "Circle, radius #{c.radius}, located at #{x}x#{y}!"

s = Shape.new(0,0)
puts "Create a circle within the Shape class."
circle = s.create_circle(1000, 500, 500)
puts circle
```

```
Circle, radius 100, located at 100x100!
Create a circle within the Shape class.
Shape::Circle 500x500
```

7. Mixins

(p. 142)

Mixins allow Ruby classes to incorporate methods and classes defined in other namespaces such as those created in modules. We “mix in” methods from a module with the `include` keyword.

```ruby
module CgiHelper
  def head
   <<HTML
<!DOCTYPE HTML>
<html>
  <head>
    <meta charset="utf-8">
    <title>%s</title>
  </head>
HTML
  end
end

class HTMLHelper
  # Mix the CgiHelper methods into the HTMLHelper class
  include CgiHelper
end

h = HTMLHelper.new
puts h.head
```

```
<!DOCTYPE HTML>
<head>
  <meta charset="utf-8">
  <title>%s</title>
```

8. TODO Exercises

8.1 TODO Create a module named `my_math.rb`

1. In the `my_math.rb` file, create a constant named `Pi` of the value of Pi to 20 digits.
2. Require your `my_math.rb` file in your program where necessary.
3. Use your `my_math.rb` `Pi` contant for all calculataions involving *Pi*.

8.2 TODO Create a module named `shape.rb`

1. In the module, create a class named `Shape` with these methods. Refer to the code above to get started.
   1. `initialize` that has to arguments: x, and y
   2. `location` Returns an array containing the x and y values of the object
   3. `to_s` Prints a description of the object: class name and x,y coordinates
   4. Write a script that uses the `Shape` class. It should:
      - Create a shape object.
      - Print the object to invoke the `to_s` method.
   5. Run the script and capture the output in a file names `week14_exercises.txt`

8.3 TODO Create nested classes

In the `Shape` class, create the nested classes below. See the example of nested classes above for ideas.

- `Circle`
- `Rectangle`
- `Triangle`

1. In each subclass of `Shape`

- Create an `area` method appropriate to the class
- Create a `to_s` method that will print out a description of the object and its properties. Your `to_s` method must call `super` and include the output of the `Shape#to_s` method.
- Run the script and capture the output in a file names `week14_exercises.txt`

2. In the `Shape` class, create a method named `create_shape` that creates a shape object of either the Circle, Rectangle, or Triangle classes, like this:

```ruby
circle = Shape.create_shape(:circle)
circle.radius = 100
puts circle.area # 31415.93

triangle = Shape.create_shape(:triangle)
triangle.height = 13
triangle.base = 13
puts triangle.area # 84.5

rectangle = Shape.create_shape(:rectangle)
rectangle.height = 10
rectangle.width  = 13
puts rectangle.area  # 130
```

- Run the script and capture the output in a file names `week14_exercises.txt`

8.4 TODO create a script named `test_shape.rb` that;

- Requires `shape.rb`, `my_math.rb` as necessary.
- Creates a `Shape` object that in turn creates `Circle`, `Rectangle`, and `Triangle` objects.
  - Prints the object to invoke the `to_s` method that will display the class, location, and area of the object

- Creates `Circle` , `Rectangle` , and `Triangle` objects using this syntax and including any parameters that your classes require.

```ruby
triangle  = Shape::Triangle.new
circle    = Shape::Circle.new
rectangle = Shape::Rectangle.new
```

- Run the script and capture the output in a file names `week14_exercises.txt`

9. TODO Turn in your work for credit.

- Create a zip file named `week14_exercises.zip` .
- Include your `my_math.rb` , `shape.rb` , `test_shape.rb` , and `week14_exercises.txt` scripts in the zip file.
- Upload the zip file to Insight using the **Week 14 Exercises UPLOAD LINK**.

And, if you find errors or confusion in this exercise, please let me know in the forums. Have fun! :)

❧