Lesson 4:
Introducing object orientation

Objectives

- Describe and use object oriented programming
- Create and use instances of built in classes
- Create custom classes
- Control the scope of class functions and variables

Objects & classes

- An object is an instance of a class
- *class*: the blueprint for the attributes and behaviors of an object
  - *attributes*: simple and/or complex variables
  - *behaviors*: methods
- *object*: one complex variable comprised of other simple and/or complex variables and methods, as defined by its class

```
class Laptop
    attr_accessor :brand
    attr_accessor :screen_size
    attr_accessor :retina

    def initialize(b, s)
        @brand = b
        @screen_size = s
        @retina = false
    end

    def display
        “#{@brand} laptop: #{@screen_size} inches”
    end
end
```

```
# Create new object from class
my_first_laptop = Laptop.new(“Dell”, 13)

# Run method from class
my_first_laptop.display
=> “Dell laptop: 13 inches”
```

```
# Create another new object from class
current_laptop = Laptop.new(“Apple Mac”, 15)

# Set attribute of object
current_laptop.retina = true
current_laptop.display
=> “Apple Mac laptop: 15 inches”
```

Objects can be thought of as a container or collection of other variables, both simple and complex (arrays, hashes, other objects) methods, called to interact with the object

The dot operator . separates the object (or class) name from its variables and methods

```
# Array class method .new called to create new array object
# Same as writing take_that = []
take_that = Array.new

# Array object method .push used to add value to array
take_that.push("Robbie")

# .length variable gives count of values in array
puts take_that.length
=> 1
```

Object Oriented Programming (*OOP*) provides:
– a modular way to build and update code
– support for code reuse
– a way to model "real world" interactions in code

OOP is a *very* widely used and supported development approach

Ruby, Python, C++, Java, C#, PHP, Perl and more support OOP development

Docs:
http://www.ruby-lang.org/en/documentation/
http://www.ruby-doc.org/core-1.9.3/
http://www.ruby-doc.org/stdlib-1.9.3/

The *File* class reads and writes files on the file system:

.new(file_name, mode): open or create specified file to append
.write(string): append string value to end of file
.close: close the file object

```
secret_file = File.new("demo.txt", "w")
secret_file.write("Oh man, I really love Take That")
secret_file.close
```

The *Time* class displays and formats date/time information

```
t = Time.now

puts t
=> “2012-10-03 14:22:30 +0100”
```

```
puts t.sunday?
=> false

puts t.day
=> 3
```

Exercise:

Creating and using instances of built-in classes

Classes

Commonly, but not necessarily, in a separate .rb file

File name should match the class name
e.g. laptop.rb

Ruby file names are generally lower-cased
Ruby class names are upper-cased

```
class Laptop
    # Code goes here
end
```

Class definition may include variables and methods
– variables defined in classes are generally preceded by @
– methods defined in classes using def and end

```
class Laptop
    @brand = "Asus"
    def get_brand
        return @brand
    end
end
```

How do you create an object of your class?

Call the .new method of your class and assign the result to a variable

```
class Laptop
    @brand = "Asus"
    def get_brand
        return @brand
    end
end

laptop = Laptop.new
puts laptop.get_brand
=> Asus
```

How do you define initial values when creating an object?

A method named initialize will be automatically called when creating a new object

```
class Laptop
    @brand = nil
    def initialize(brand)
        @brand = brand
    end
    def get_brand
        return @brand
    end
end

laptop = Laptop.new("Asus")
puts laptop.get_brand
=> Asus
```

Can you reach and display class instance variables?

No, variables in classes are private to their class (@@) or instance (@)

Only code inside the class can read/write them and code outside an object cannot access its instance variables

```
class Laptop
    @brand
    def initialize(brand)
        @brand = brand
    end
end

laptop = Laptop.new("Asus")
puts laptop.brand
=> undefined method or variable 'brand' error
```

So how do you get data out of the object?

Write a method which returns the desired data or, use
attr_accessor :my_name

This creates @my_name variable and exposes it outside the class

```
class Laptop
    attr_accessor :brand

    def initialize(brand)
        @brand = brand
    end
end

laptop = Laptop.new("Asus")
puts laptop.brand
=> Asus
```

How do you use a class defined in a different file?

Ruby looks for classes in paths defined in a global array called `$LOAD_PATH`

Determine current directory using the auto-created `__FILE__` constant
add the current directory to `$LOAD_PATH`

```
$LOAD_PATH.unshift(File.dirname(__FILE__))
```

How do you use a class defined in a different file?

require makes a class available to your code

Do not use the file extension (.rb) in a require statement so, if laptop.rb is in a folder called lib.

Notice you require the file name of the class, not the class name itself.

```
$LOAD_PATH.unshift(File.dirname(__FILE__))
require 'lib/laptop'

laptop = Laptop.new("Asus")
puts laptop.brand
=> Asus
```

Exercise:

Declaring custom classes

Scope

Scope	Example	Explanation
Local	name	available in the same script or method
Instance	@name	unique value for each instance of a class, available from any method of that class
Class	@@name	same shared value for all instances of a class, available from any method of that class
Global	$name	same shared value for all code running within a single Ruby program

It's generally bad to create global ($) variables, because they end up being used faaaaaar from where they're created, making code hard to follow. Ruby uses them to expose system info.

Class variables (@@) are also risky, for reasons explained here: http://www.oreillynet.com/pub/a/ruby/excerpts/ruby-best-practices/worst-practices.html

Local variables in a method are visible ***only*** in that method

```
def set_local_name
    name = "Fred"
end

set_local_name

puts name
=> Undefined method 'name' error
```

Instance variables are visible ***anywhere*** in the current script or instance

```
def set_instance_name
    @name = "Fred"
end

set_instance_name

puts @name
=> Fred
```

Instance and local variables are ***private*** within an object

```
class User
    @name
end

person = User.new
person.name = "Fred"
=> Undefined method ‘name’
```

Create an attribute accessor to ***expose*** data from an object

```
class User
  attr_accessor :name
end

person = User.new
person.name = "Fred"
=> Fred
```

Inheritance

One class can inherit the variables and methods of another class using the < operator

Child class inherits from parent class, or sub-class inherits from super-class attributes are assigned in the class as instance (@) variables

```
class User
    attr_accessor :name
    def initialize(name)
        @name = name
    end
end

class Admin < User
end

the_admin = Admin.new("Rik")
puts the_admin.name
=> Rik
```

```
class User
    attr_accessor :name
    def initialize(name)
        @name = name
    end
end

class Admin < User
    # attr_accessor :name inherited from parent
    attr_accessor :email
    def initialize(name, email)
        @name = name
        @email = email
    end
end

the_admin = Admin.new("Rik")
puts the_admin.name
=> Rik
```

A child can override a parent variable or method by re-using its name

```
class User
    def say_something(name)
        "I'm a user named #{name}"
    end
end

class Admin < User
    # overrides same-named parent method
    def say_something(name)
        "I'm an admin named #{name}"
    end
end

the_admin = Admin.new
puts the_admin.say_something("Rik")
=> I'm an admin named Rik
```

super keyword calls parent method from overriding child method

```
class User
    def say_something(name)
        "I'm a user named #{name}"
    end
end

class Admin < User
    def say_something(name)
        # passes to parent then modifies result
        super(name) + " and an admin too"
    end
end

the_admin = Admin.new
puts the_admin.say_something("Rik")
=> I'm a user named Rik and an admin too
```

If defined in different physical files, a child **must** require its parent

In file: lib/user.rb

```
class User

end
```

In file: lib/admin.rb

```
require 'lib/user'

class Admin < User

end
```

Remember that a require statement is relative to the paths defined in $LOAD_PATH ***not the physical location of the file*** using the require statement

Exercise:

Controlling the scope of methods and variables in a class

Lab:
Introducing object orientation