Lab 2: Ruby Modules, Mixins, and Gems

Learning Objectives:

1. Introduction to Modules in Ruby
2. Introduction to Mixins in Ruby
3. A detailed look into RubyGems

Supporting files:

1. Fruits.rb
2. ModuleTest.rb
3. Number.rb
4. apple.rb
5. apple.gemspec

---

1. Introduction to Modules in Ruby

Ruby Modules are a way of grouping together methods, classes, and constants. They are similar to classes in that they hold a collection of methods, constants, and other module and class definitions. Modules are defined much like classes are, but the module keyword is used in place of the class keyword. Modules give us two major benefits.

- Modules provide a namespace and prevent name clashes.
- Modules implement the mixin facility.

Modules define a namespace, a sandbox in which your methods and constants can play without having to worry about being stepped on by other methods and constants. One or more modules can be defined with the same function name but different functionalities.

Syntax of a Ruby module:

```
module Identifier
  statement1
  statement2
  ...........
end
```

Once defined modules can be loaded using the keyword **require** as follows:

```
require '<filename>'
```

Note that, the .rb extension is not needed to be mentioned when loading the module. Next, in order to embed a module from the loaded file, in your class the **include** keyword can be used, as follows:

```
include <modulename>
```

Note that, if multiple required files contain the same function name, the resulting code ambiguity in the calling program can be avoided using modules and appropriate functions can be called using module name.

Let us go through and execute the attached files Fruit.rb and ModuleTest.rb for examples.

---

2. Introduction to Mixins in Ruby

When a class can inherit features from more than one parent class, the class is supposed to show multiple inheritance. Ruby does not support multiple inheritance directly but Ruby Modules have another wonderful use. At a stroke, they pretty much eliminate the need for multiple inheritance by providing a facility called **Mixin**. A Mixin is just a module that is included into a Class. When you "mixin" a Module into a Class, the Class will have access to the methods of the Module. Mixins give you a wonderfully controlled way of adding functionality to classes. However, their true power comes out when the code in the mixin starts to interact with code in the class that uses it.

Overall Mixins in Ruby behave as follows:

```
module Test1          ## Defining the first module
  def method1
  end
end

module Test2          ## Defining the second module
  def method2
  end
end

class FinalTest
include Test1         ## Including both the modules Test1 and Test2
include Test2
  def method3
  end
end

final = FinalTest.new         ## The object of FinalTest class has access to all the methods belonging
final.method1                 ## to the two included modules
final.method2
final.method3
```

Let us open and execute the attached file Number.rb for an example.

---

3. A detailed look into RubyGems

A Ruby Gem can be thought of as a library. At its most basic form, a Ruby gem is a package. It has the necessary files and information for being installed on the system. They can easily be used to extend or change functionality within Ruby applications. RubyGems is a package manager which became part of the standard library in Ruby 1.9. It allows developers to search, install and build gems, among other features. All of this is done by using the gem command-line utility. You can find, install, and publish your own RubyGems here.

Previously we have discussed how to install RubyGems in your computers (Ref: Lab1- ECE421 – Using Ruby Version Manager.pdf). In order to use an installed gem, you need to require it in the beginning of the document as follows:

```
require '<RubyGemName>'
```

Making your own Gem:

Creating and publishing your own gem is easy. This webpage gives a detailed and very organized overview of almost everything you need to know about creating and using gems.

Creating a gem primarily consists of two steps. At first, you need to place the code for your package within the 'lib' directory of your gem folder. The convention is to have one Ruby file with the same name as your gem, since that gets loaded when 'require' statement is executed. That one file is in charge of setting up your gem's code and API.
For an example, open to the gem_example/lib/apple.rb file among the attached files.

Next, you need to create a **gemspec** file for your gem. The gemspec defines what's in the gem, who made it, and the version of the gem. It's also your interface to RubyGems.org. Refer to the apple.gemspec file among the supporting files, for an example of how to create the gemspec. After you have created a gemspec, you can build a gem from it. Then you can install the generated gem locally to test it out.

For example: in order to **build** our sample gem '**apple**' from the attached files, we need to type the following in Terminal:

```
$ gem build apple.gemspec
```

Output:

```
Successfully built RubyGem
 Name: apple
Version: 1.0.0
File: apple-1.0.0.gem
```

To **install** the gem, type the following in Terminal:

```
$ gem install ./apple-1.0.0.gem
```

Output:

```
Successfully installed apple-1.0.0
Parsing documentation for apple-1.0.0
Installing ri documentation for apple-1.0.0
Done installing documentation for apple after 0 seconds
1 gem installed
```

To **test** the gem with IRB Shell in Terminal:

```
$ irb
irb(main) :001:0> require 'apple'
```

Output:

```
=> true
```

```
irb(main) :002:0> Apple.sayName
```

Output:

Hello! I am an apple.
=> nil

```
irb(main) :003:0> Apple.sayColor
```

Output:

I am red in color!
=> nil

To uninstall the gem, quit the IRB shell and type in Terminal:

```
$ gem uninstall apple
```

Output:

Successfully uninstalled apple-1.0.0
