Modules and Mixins

Modules are a way to group together methods, classes and constants. Modules were introduced to programming languages in the 1970s, to segregate data and operations into different pieces.

Ruby's modules are are similar to namespaces in languages such as C++. What's a namespace?

Of course, whenever you mix namespaces, you have the possibility of name clashes.

Say you have a graphics library that contains classes for Window and (`window.rb`) and Border (`border.rb`).

You can add the statement `require 'window'` to the program, and then use features of Window:

```
myWindow = Window.new
myWindow.open
```

Both `window.rb` and `border.rb` have a `top` method, which gives the position of the top of the Window and the top of the Border, respectively.

An application programmer wants to lay out windows with borders, and thus loads both `window.rb` and `border.rb` into the program.

What's the problem here?

The window functions can go into one module

```
module Window
    def Window.top
    # ..
    end
end
```

and the border functions can go into another

```
module Border
    def Border.top
    # ...
    end
end
```

When a program needs to use these modules, it can simply load the two files using the Ruby `require` statement, and reference the qualified names.

```
require 'window'
require 'border'
trueTop = Window.top + Border.top
```

Mixins

Modules are different from classes.

- Modules hold all their data within the module itself.
- Classes allow data to be distributed among various objects.

But modules can be "included" in classes. When you `include` a module within a class, all its functionality becomes available to the class.

The methods of the module become instance methods in the class that `include`s it.

[Ruby's `require`, `include`, and `load` statements have similar functionality.

- `include` makes features available, but does not execute the code.
- `require` loads and executes the code one time (somewhat like a C **`#include`**).
- `load` loads and executes the code every time it is encountered.]

Let's contrast mixins with multiple inheritance in (what languages?).

Consider the following code:

```
module Introspect
  def kind
    puts "This object is a #{self.class.name}"
  end
end

class Animal
  include Introspect
  def initialize(name)
     @name = name
  end
end

class Car
  include Introspect
  def initialize(model)
    @model = model
  end
end

d = Animal.new("Cat")
c = Car.new("Ferrari")
d.kind # kind method is available through …
c.kind # .. the mixin Introspect

>>This object is a Animal
```

```
>>This object is a Car
```

Exercise: Select the correct method lookup order. Note that you need to use your personal (not NCSU) Github account for the IDE.

In addition to including modules, classes can also be `extend`ed by modules.

So, what's the difference between `includ`ing a module and `extend`ing a class with it?

Note: It is also possible to `extend` instances, which will add methods of the module to specific objects (not classes).

Comparable

A good example of the power of modules is Comparable, from the Ruby library.

To use comparable in a class, the class needs to define a method called `<=>` (sometimes called "rocket").

Once we define this method, we get a lot of useful comparison functions, such as `<`, `>`, `<=`, `>=`, `==` and the method `between?` for free.

What is this like in Java?

Here is an example. Suppose that we have a `Line` class:

```
class Line
  def initialize(x1, y1, x2, y2)
    @x1, @y1, @x2, @y2 = x1, y1, x2, y2
  end
end
```

We compare two lines on the basis of their lengths.

We add the `Comparable` mixin as follows:

```
class Line
  include Comparable

  def length_squared
    (@x2-@x1) * (@x2-@x1) + (@y2-@y1) * (@y2-@y1)
  end

  def <=>(other)
    self.length_squared <=> other.length_squared
  end
end
```

`<=>` returns `1`,`0`, or `-1`, depending on whether the receiver is greater than, equal to, or less than the argument.

We delegate the call to `<=>` of the `Integer` class, which compares the squares of the lengths.

Now we can use the `Comparable` methods on `Line` objects:

```
l1 = Line.new(1, 0, 4, 3)
l2 = Line.new(0, 0, 10, 10)
puts l1.length_squared
if l1 < l2
  puts "Line 1 is shorter than Line 2"
else if l1 > l2
  puts "Line 1 is longer than Line 2"
else
  puts "Line 1 is just as long as Line 2"
 end
end
```

```
>>Line 1 is shorter than Line 2
```

Exercise: Battle of Minions

This exercise utilizes `Comparable` in Ruby to implement a simple game called "Battle of Minions."

Suppose in a game, players can summon their minions to battle.

Every minion has four attributes:

- `name`
- `atk`
- `defense`
- `HP`

The rules for which minion wins a battle are …

1. The damage that one minion does to the other minion is equal to its `atk` minus the other's `defense`.
2. The winning minion is the one that has a higher HP compared to the damage done by the other minion, except that …
    a. If the `atk` of minion A is less than or equal to the `defense` of minion B, minion B automatically wins.
    b. If both `atk`'s are less than or equal to the other's `defense`, they automatically tie.

Assume that

- `minion A > minion B` means minion A defeats minion B,
- `minion A < minion B` means minion B defeats minion A,
- `minion A == minion B` means they tie.

To execute the program from command line, please use this command:

```
ruby battle_of_minions.rb
```

Exercise

By filling in the blanks, complete the method `<=>(other)` in the `Minion` class to implement the battle of two minions.

Expected output

Upon executing the program, you should see the following output (assuming all the blanks are filled in correctly):

```
rider
```

If you finish early, implement the additional rule that if a minion has an HP ≤ 0, it loses, and if both minions have an HP ≤ 0, they tie. Does this require additional code?

Composing Modules

`Enumerable` [SAAS §3.7] is a standard mixin, which can be included in any class.

It has a very useful method `inject`, which can be used to repeatedly apply an operation to adjacent elements in a set:

```
[1, 2, 3, 4, 5].inject {|v, n| v+n }
>>15
```

Many built-in classes include `Enumerable`, including `Array` and `Range`.

```
('a' .. 'z').inject {|v, n| v+n }
```

Exercise: Use inject to define a factorial method.

Let's define a `VowelFinder` class that includes `Enumerable`. It will have an `each` method for returning successive vowels from a string. This method yields each time it encounters a vowel.

```
class VowelFinder
  include Enumerable

  def initialize(string)
    @string = string
  end

  def each
    @string.scan(/[aeiou]/) do |vowel|
      yield vowel
    end
  end
end
```

Here's an example of its use:

```
VowelFinder.new("abacadabra").inject {|v, n| v+n}
>>
```

Exercise: Use yield to return every third element of an Array.

Reflection

We've already seen a few examples of where Ruby programs can discover things about themselves at run time.

For example, we have seen calls like

```
3.14159.methods
```

Why do we call this "discovering things about [3.14159] at run time"?

Reflection allows program entities to discover things about themselves through introspection.

For example, an object can ask what its methods are, and a class can tell what its ancestors are.

While Java also provides reflection, it does so much more verbosely than Ruby.

The related technique of *metaprogramming* allows one to create new program entities, such as methods or classes, at run time. Why is this called *meta*programming?

```
puts [1, 2, 3, 4, 5].length
>> 5
puts "Hey".class
>> String
puts "John".class.superclass # print the superclass of a String
```

```
>> Object
```

Peruse the Ruby documentation for Object and examine methods such as `instance_of?` and `kind_of?`.

Note: While it may be useful, in debugging, to print out the class of an object, it is almost always a mistake to *test* the class of an object:

```
if s.kind_of? Integer then this_method else
that_method end
```

Why?

Now look at `Object` methods like `instance_variables`, `methods`, `private_methods`.

Then look at `Module` and examine such methods as `class_variables` and `instance_methods`.

Exercise: Test your knowledge of reflection.

Methods of Array

Ruby's `Array` class contains many methods for treating arrays as collections. One example is `collect`. Applied to an array with a block as parameter, it returns a new array that consists of the results of applying the block to each element of the array.

```
a = [1, 2, 3]
a.collect {|x| x**2}
>> [1, 4, 9]
```

`Array` has several other methods for working with collections, such as `select` and `reverse`. Read about them in the `Array` documentation.

Exercise: Test your knowledge of collection operators.

Now, time to put it all together and work on a more involved program, a Blackjack game.

Exercise: Deck and Player