RUBY FOR PROGRAMMERS

This cheat-sheet accompanies the Bitwise Courses course on *Ruby For Programmers* by Huw Collingbourne.

Attributes

Attributes are Ruby's equivalent of 'properties' in many other programming languages. They provide a simple way to 'get' and 'set' the values of an object's instance variables. Here I create a pair of *get* and *set* attributes called `description` to return and assign a value to the instance variable, `@description`:

```
def description
   return @description
end

def description=( aDescription )
   @description = aDescription
end
```

Ruby provides a quicker way to create simple *get* and *set* accessors. You can simply specify a symbol (an identifier that begins with a `:` character) as an argument to the `attr_reader` (to create a getter), `attr_writer` (to create a setter) or `attr_accessor` (to create a pair of getter and setter) methods. This is how I create a getter accessor called `name`, a setter accessor called `description` and a pair of getter/setter accessor attributes called `value` inside the `Thing` class:

```
class Thing
  attr_reader :name
  attr_writer :description
  attr_accessor :value
end
```

And now given a `Thing` object `t`, I can use these accessors like this:

```
aname = t.name
t.description = "A soft, furry wotsit"
t.value = t.value + 2
```

Modules

Modules are like classes but they do not allow **instances** (objects) to be created from them and they do not permit **inheritance**. Modules are often used to store 'libraries' of methods and constants that you may want to use in a variety of unrelated classes. In other words, modules allow you to share code without using inheritance.

Let's suppose you have this code in a file called *mymodule.rb*:

```
module MyModule
   GOODMOOD = "happy"
   BADMOOD = "grumpy"

    def greet
      return "I'm #{GOODMOOD}. How are you?"
    end

    def MyModule.greet
      return "I'm #{BADMOOD}. How are you?"
    end
end
```

Here `GOODMOOD` and `BADMOOD` are constants. They are accessed from outside the module using the `::` operator:

```
puts(MyModule::GOODMOOD)
```

This displays:

```
happy
```

`MyModule.greet` is a module method and can be accessed using the syntax shown below:

```
puts( MyModule.greet )
```

This displays:

```
I'm grumpy. How are you?
```

The other `greet` method (the one not preceded by `MyModule.`) is an instance method. Modules cannot have instances so in order to access this method I must include (or 'mix in') this module in another class. Here I include the module and call the `greet` method:

```
include MyModule
puts( greet )
```

This displays:

```
I'm happy. How are you?
```

When including modules into separate source code files you may first need to ‘require’ (that is, to load) the module code file before including the module, like this:

```
require( "./ mymodule.rb")
include MyModule
```