UNIVERSITY OF SURREY

COMM013: Ruby on Rails

Lab Class 2: Answers Autumn 2014

A Little Practice with Ruby

Setting the Scene

We will develop the book stock example from the lectures a little bit further. As mentioned there, this example is taken from: *Programming Ruby 1.9 & 2.0*, by Dave Thomas, published by Pragmatic Programmers. I have slightly modified and extended it, but still strongly recommend using the book itself for further study of Ruby:

http://pragprog.com/book/ruby4/programming-ruby-1-9-2-0

A reminder of our starting point

In the lectures we had got as far as a basic definition of a BookInStock class:

```
class BookInStock
  attr_reader(:isbn)
  attr_accessor(:price)

  def initialize(isbn, price)
    @isbn = isbn
    @price = Float(price)
  end

  def to_s
    "ISBN: #{@isbn}, price: #{@price}"
  end

end
```

The first thing to do is to use whichever text editor you prefer using to save the above code in a folder called "book_in_stock.rb".

Open a command prompt or terminal window (depending on which operating system you are using) and change directory to the folder where the above file has been saved. Then open up Interactive Ruby (you should remember how to do this from the lecture).

Then load the file you just created. You can do this from the irb by typing at the prompt:

```
> load "book_in_stock.rb"
```

Now create two or three new instances of BookInStock and then print their "internal states" out at the console (use "puts" to do this).

Again, refer to the lecture if you are unsure about this.

The price look a bit funny as there is no currency. Depending on your preference, edit the above file so that a "$" or a "£" sign is in front of the price when it is printed out. Now for something cool. Reload your "book_in_stock.rb" file (simply enter the above command again – you can do this by using the "up arrow" key to recall the command and then hit enter). If you print out the internal states of your book instances, you will see your edit is reflected in the result.

(Virtual) Attributes

In the lectures I introduced the notion of an instance variable (`@isbn` and `@price`, for example). These capture the *internal state* of an object. However (in Ruby) these instance variables are private to an object. And that is private in the very strict sense of Ruby – the only way another object (even one of the same class) can read or write to an instance variable is through an accessor method. We can write these methods explicitly if we wish, or we can declare them using `attr_reader` or `attr_accessible` as above.

What we have now is a way in which the external world can access and manipulate the state of an object. Because of the naming conventions used within Ruby, the names of the externally visible entities are the same as those of the instance variables in this case:

```
my_book.isbn
my_book.price = new_price
```

To make this internal/external distinction clear, the externally visible entities are referred to as the "attributes" of an object. It happens that each attribute of a BookInStock maps onto an internal variable (with the same name). However, this need not necessarily be the case, as we will now see.

Suppose now that I want to refine my model so that it is possible to retain a record of the original publisher's price of a book, but allow the shelf price to be discounted by a certain percentage. To do this, modify the code so that it looks like this (changes in **bold**):

```
class BookInStock
  attr_reader(:isbn)
  attr_accessor(:discount)

  def initialize(isbn, price)
    @isbn = isbn
    @price = Float(price)
    @discount = 1.0
  end

  def shelf_price
    @price * @discount
  end

  def to_s
    "ISBN: #{@isbn}, price: #{@price}"
  end

end
```

Now, although price is still an instance variable, it is no longer an *attribute* of a BookInStock object. In contrast, shelf_price is an attribute of BookInStock but it does not directly map onto an instance variable; in that sense it could be referred to as a *virtual* attribute.

Once you have made the above changes, it is safest to quit the irb and then reopen it. Then load BookInStock's file again.

Create a new instance of BookInStock.

Now, see if you still have access to price as an attribute. Enter:

```
> my_book.price
```

This will generate an error. But you do have access to shelf_price:

```
> my_book.shelf_price
```

You can even assign values to virtual attributes. Make the following modification (in bold) and load the file again:

```
class BookInStock
  attr_reader(:isbn)
  attr_accessor(:discount)

  def initialize(isbn, price)
    @isbn = isbn
    @price = Float(price)
    @discount = 1.0
  end

  def shelf_price
    @price * @discount
  end

  def shelf_price=(price)
    @discount = price / @price
  end

  def to_s
    "ISBN: #{@isbn}, price: #{@price}, discount: #{@discount}"
  end

end
```

Now you can assign a new value to the `shelf_price`, and the value of the `@discount` instance variable will be automatically updated to ensure that the internal state is consistent with the external representation of the object's state.

Make sure you try this out, and print the internal state of the object out to check this!

As always, do note that nothing magic ever happens – it is your responsibility as a programmer to make sure that the internal and external representations are consistent.

This hiding of the difference between instance variables and calculated values is an important principle of object-oriented software engineering. As a programmer, you are free to change the way an object's internal state is represented as much as you like so long as the externally visible attributes are not changed.

Exercise 1

Modify the `to_s` method so that it prints out the `shelf_price` instead of the price.

Ans:

```
  def to_s
   "ISBN: #{@isbn}, price: $#{self.shelf_price}, discount: \
            #{@discount}"
  end
```

Reading the Data

The next thing we want to do is to be able to read in all the data about our books from a file. We will use CSV files for our data. Fortunately, Ruby provides support for CSV with a very useful library and we will use that to build our CsvReader class.

We will build this class up incrementally. The first thing we want to do is to open a file and then use each row after the header to instantiate a new instance of BookInStock. We will collect all this instances into an instance variable called BooksinStock (remember what I said about collections being conventionally named as pluralisations of the name of their instances). Create a new file called "csv_reader.rb" in the same folder as "book_in_stock.rb" and enter the following code:

```
require 'csv'
require_relative 'book_in_stock.rb'

class CsvReader

  def initialize
    @books_in_stock = []
  end

  def read_in_csv_data(csv_file_name)
    CSV.foreach(csv_file_name, headers: true) do |row|
      @books_in_stock << BookInStock.new(row["ISBN"], row["Price"])
    end
  end
end
```

First, we need to require the csv library. We also need to require "book_in_stock.rb", and I have used `require_relative` on the assumption that this file will always be in the same folder as my new file.

The initialisation method should be clear to you now, so I will not explain that. The definition of `read_in_csv_data` takes a file name as parameter (you will need to provide the full path from the current working directory – so putting all the csv files in the current working directory is simplest!). It then uses the CSV class from our library to read in successive rows from that file, with the

exception of the first row which it uses as a header to set up the key:value hashes that are the internal representation of each row.

Once a row has been read, it is used to instantiate a new instance of BookInStock and append that to the instance variable (using another nice intuitive piece of Ruby syntax).

Testing

We are not going to do any formal testing for a couple of weeks. But at this stage it is worth informally (unit) testing your class definition to make sure it does what it says. To save you a bit of typing you can download a ready prepared csv file from here:

https://dl.dropboxusercontent.com/u/23490033/file1.csv

Save this into your current working folder.

Then in the irb type:

```
> load 'csv_reader.rb'
> csv = CsvReader.new
> csv.read_in_csv_data('file1.csv')
```

That should work without failure if you have followed the above carefully. Sort out the errors if there are any (ask if you are stuck!). The trouble is, you will not know for sure if it has actually built up the `@books_in_stock` instance variable correctly.

Exercise:

The reason for not being able to easily access this instance variable is that we have not defined a corresponding externally visible attribute. Other than for testing this is not part of the required public interface for CsvReader. A way to handle this is to override the default `to_s` method in order to have it summarise the internal state of a CsvReader object. Try and do this now. Then reload the class and check it has indeed updated the internal state correctly.

Answer:

```
def to_s
  puts "The following books are in stock"
  for book in @books_in_stock
    puts book.to_s
  end
end
```

Summing the stock value

Now we can add the attribute that we actually require. This is `total_value_of_stock`. Note that we will implement this as a calculated (or virtual) attribute, but that does not matter to any client of the class. For performance reasons if we are going to access this attribute many times with the stock unchanged then we might decide to change the implementation so that its value is cached as an instance variable. We can do this without breaking any code external to the class. But for our purposes the following code is satisfactory. Update your implementation by adding the following method and reload the class (additions in bold):

```
def total_value_in_stock
  sum = 0.0
  @books_in_stock.each {|book| sum += book.shelf_price}
  sum
end
```

Remember that Ruby implicitly returns the evaluation of the last statement in a method definition. Hence we simply need to evaluate the `sum` before closing the method.

Exercise:

Ruby has a rather nice, although obscurely named, method called inject that can be used to accumulate a value across members of a collection. The general pattern is:

```
collection_instance.inject(init) {|acc,element| acc opp element2 }
```

The value `init` is used to initialise the accumulator acc. The operator `opp` is typically one of `{ +, *, << }`. `element2` could either be `element` itself or an attribute of `element`.

See if you can use inject to reduce the above method definition down from three lines to one.

Ans:

```
def total_value_in_stock
  @books_in_stock.inject(0.0) {|sum, book| sum + book.shelf_price}
end
```

Counting book copies: Exercise

It would be useful to implement a second virtual attribute – a record of the number of instances in stock of books with a specific isbn.

```
def number_of_each_isbn
  #...
end
```

I'm going to leave this for you to do as an exercise, but here are a couple of hints. A hash is a good structure for doing this. In this case, we would use the isbn as a key and the number of books with that isbn as the corresponding value. You will need to iterate through books_in_stock in a similar way to the preceding example (but *not* using inject) and each time you find a book of a specific isbn you should increment its associated value by 1.

There is a trick you need to know, though. You will remember from the lectures that if you try to access the value of a key that has not so far been explicitly recorded in a hash, Ruby will return `nil`. This is not the behaviour we want in this case as we need the starting point for the book counters to be `0`, not `nil` ("+" is not defined as a method on `nil`). Fear not, Ruby can sort this for you. You can specify the default value for a value as follows:

```
book_counts = Hash.new(0)
```

Now try and implement this method yourself. It is not too difficult, but I will provide an answer tomorrow.

Make sure you test that this is working correctly before moving on to the final step.

Ans:

```
def number_of_each_isbn
  book_counts = Hash.new(0)
  for book in @books_in_stock
    book_counts[book.isbn] = book_counts[book.isbn] + 1
  end
  book_counts
end
```

Defining the application file

No real thinking is needed now, provided that everything you have done so far is working. You can exit the irb and create the main application class. What we want to do now is to be able to take a list of csv files from the command line and process them to print out the total value of the books in stock. This should all be quite straightforward. Only one thing needs explanation. The ARGV variable in the code below accesses the command line arguments to create a list of csv files that are then processed sequentially.

Create a file called "stock_stats.rb" in the same folder as all the other files and then edit it so that it contains the following contents:

```
require_relative 'csv_reader'

reader = CsvReader.new

ARGV.each do |csv_file_name|
  STDERR.puts "Processing #{csv_file_name}"
  reader.read_in_csv_data(csv_file_name)
end

puts "Total value = #{reader.total_value_in_stock}"
```

You can copy down a couple more csv files if you like:

https://dl.dropboxusercontent.com/u/23490033/file2.csv

https://dl.dropboxusercontent.com/u/23490033/file3.csv

Now enter this at the command line:

```
$ ruby stock_stats.rb file1.csv file2.csv file3.csv
```

You should see the processing message printed out for each file, followed by a statement of the total value of the stock.

A little discussion

Apart from a slight variation in the BookInStock class, the content of all the files (apart from the csv ones) is taken from Dave Thomas' book *Programming Ruby 1.9 & 2.0.*

As mentioned, please refer to this book for further insights. However, there is one point I would like to discuss. Before introducing the definition of the CsvReader class, Dave Thomas states that whilst in OO design the classes normally map onto external entities, there is "another source of classes in your designs. These are the classes that correspond to the things inside your code itself." He then introduces CsvReader. However, I would assert that this does indeed map onto an external entity. Although I have retained the same name in the above in order to make it very clear that I have reused his example, I personally believe that this class is in fact a representation of the BookShop. It has an internal instance variable that maintains a catalogue of the books in stock, and two attributes that capture important aspects of the external state of a shop: the total value of books in stock, and the numbers of copies of books for each isbn in stock. It does not represent anything about a Csv reader – the `read_in_csv` method is merely a utility method for initialising the state of a BookShop using a set of csv files. This issue will become clearer when we move to Rails where we can dispense with the csv files and build a model that persists in a database. Then we will see that a "BookShop has many BooksInStock" and we can maintain a clean representation of the internal and external views of the state of a BookShop without needing the read_in_csv method; Rails' Active Record will populate the models with data in the background.

This is a fine point and do not worry about it too much at this stage. We will, however, talk a lot more about modelling later in the course.

Paul J Krause

Professor of Software Engineering

University of Surrey

12th October 2013