Do you know how to write an internal DSL in Ruby?

A Domain-Specific Language (DSL) is a description language designed for a fairly narrow purpose.

by RubyLearning
http://rubylearning.com/blog/

IMPORTANT

You Do NOT Have Rights to Edit, Resell or Claim Ownership to this Document!

However...You DO Have the Right to Pass this Document Along to Others Who Might Benefit from it!

Feel free to share it with your blog readers and give it away as a freebie.

DISCLAIMER: All information contained within this document is strictly the views represented by Satish Talim, hereafter referred to as the author or author, at time of publication. The author reserves all rights. Reasonable attempt has been made to accurately substantiate all information in this document. However, the author, does not, and cannot, take responsibility for any errors or exclusions that may be contained in within. No suitability of use is meant or implied in regard to any system or program, use at your own risk. It is recommended the reader contact the appropriate qualified professional for advice in these and any other areas should it be needed.

Do you know how to write an internal DSL in Ruby?

Almost all Ruby programming newbies would love to get their hands wet writing a Ruby DSL. This article explains how you can write a simple Ruby DSL.

Introduction

A **Domain-Specific Language (DSL)** is a (usually small) programming or description language designed for a fairly narrow purpose. DSLs are targeted at end users or domain specialists who are not expert programmers. Martin Fowler classifies DSLs into two styles – external and internal. An external DSL is a language that is different from the main programming language for an application, but that is interpreted by or translated into a program in the main language. An internal DSL transforms the main programming language itself into the DSL (our simple DSL is tied to the Ruby programming language).

Ruby code blocks

Ruby's *support for blocks* (i.e., closures) is useful in defining internal DSLs.
Ruby code blocks (called closures in other languages) are chunks of code between braces or between `do-end` that you can associate with method invocations, almost as if they were parameters. A Ruby block is a way of grouping statements, and may appear only in the

source next to a method call; the block is written starting on the same line as the method call's last parameter (or the closing parenthesis of the parameter list). The code in the block is not executed at the time it is encountered. Instead, Ruby remembers the context in which the block appears (the local variables, the current object, and so on) and then enters the method. Matz says that any method can be called with a block as an implicit argument. Inside the method, you can call the block using the `yield` keyword with a value. Blocks are not objects, but they can be converted into objects of class `Proc`. One way a block can be converted to a `Proc` object is by passing a block to a method whose last parameter is prefixed with an ampersand. That parameter will receive the block as a `Proc` object:

```
def my_method(p1, &block)
  ...
end
```

instance_eval

The class `Object` has an `instance_eval` public method which can be called from a specific object. It provides access to the instance variables of that object. It can be called either with a block or with a string:

```
class Rubyist
  def initialize
    @geek = "Matz"
  end
end
obj = Rubyist.new
# instance_eval can access obj's private methods
# and instance variables
obj.instance_eval do
  puts self  # => #<Rubyist:0x2ef83d0>
  puts @geek # => Matz
end
```

The block that you pass to `instance_eval` helps you dip inside an object to do something in there. You can wreak havoc on encapsulation! No data is private data anymore.
`instance_eval` can also be used to add class methods as shown below:

```
class Rubyist
end
Rubyist.instance_eval do
  def who
    "Geek"
  end
end
puts Rubyist.who # => Geek
```

Deciding on a simple DSL

You are an expert Ruby programmer and your friends Victor, Michael and Satoshi (all 3 are novice chess players) have requested you to write a Ruby program for them, that could help them with a listing of the best black opening chess moves.

You tell your chess friends that if they need help they should individually send you a text file containing the white's first move, as follows:

```
h4
a3
e4
```

h4, a3, e4 would be Ruby methods in your DSL program. Once we get the DSL to follow valid Ruby syntax, Ruby does all the work to parse the file and hold the data in a way that we can operate on it.

Victor is playing the black pieces and his opponent plays the opening white piece (say h4). Victor would like to know - what's the best strategy to counter white's opening move of h4. He also would like to know, what if his opponent would have played a3.

Victor decides to send a text file to you.

The DSL program – chess_opener.rb

Being a Ruby expert, you dish out your first version of the DSL program – chess_opener.rb:

```
class ChessOpener
  def initialize
    @data = {}
    load_data
  end

  def self.load(filename)
    dsl = new
    dsl.instance_eval(File.read(filename))
  end

  def h4
    puts "=========="
    puts @data.assoc("h4")
    puts "=========="
  end

  def a3
    puts "=========="
    puts @data.assoc("a3")
    puts "=========="
  end

  def method_missing(method_name, *args, &block)
    msg = "You tried to call the method #{method_name}. There is
no such method."
    raise msg
  end

  private
  def load_data
    @data = {"a3" => ["Anderssen's Opening Polish Gambit: 1. a3 a5
2. b4",
                      "Anderssen's Opening Creepy Crawly
Formation: 1. a3 e5 2. h3 d5",
                      "Anderssen's Opening Andersspike: 1. a3 g6
2. g4"],
             "h4" => ["Koola-Koola continues 1.h4 a5",
                      "Wulumulu continues 1.h4 e5 2. d4",
                      "Crab Variation continues 1.h4 any 2. a4",
                      "Borg Gambit continues 1.h4 g5.",
                      "Symmetric Variation continues 1.h4 h5"]}
  end

end
```

Some explanation of code

The `initialize` method of your class `ChessOpener` creates a `Hash` object `@data` and populates it by calling the `private` method `load_data`. You have referred to the online list of chess openingsto create the hash `@data`. The current program has the openings only for a3 and h4 moves, but you plan to add the other moves soon.
You want a simple and straightforward way to parse the DSL file. Something like:

```
my_dsl = ChessOpener.load(filename)
```

Also, you would like to accept the DSL file from the command line, something like:

```
my_dsl = ChessOpener.load(ARGV[0])
```

You write a *class method* `load`:

```
def self.load(filename)
  dsl = new
  dsl.instance_eval(File.read(filename))
end
```

The class method `load` creates a `ChessOpener` object and calls `instance_eval` on the DSL file (chess_opener_test.txt above). If you feed `instance_eval` a string, `instance_eval` will evaluate the string as Ruby code. In fact, this Ruby code is nothing but calls to the methods h4 and a3 which are respectively called. The

methods h4 and a3 make use of Ruby Hash's `assoc` method to extract the information about the particular (say h4) move.
The program also provides a method_missing method, in case the program fails to find a method say h5 (assuming Victor has typed that by mistake in the file chess_opener_test.txt.)

Running the DSL program

You next write the program – chess_opener_test.rb, ensuring that the files chess_opener.rb, chess_opener_test.rb and chess_opener_test.txt are in the same folder on your computer.
You now run your Ruby code as follows:

```
ruby chess_opener_test.rb chess_opener_test.txt
```

Here's the sample output:

```
==========
h4
Koola-Koola continues 1.h4 a5
Wulumulu continues 1.h4 e5 2. d4
Crab Variation continues 1.h4 any 2. a4
Borg Gambit continues 1.h4 g5.
Symmetric Variation continues 1.h4 h5
==========
==========
a3
Anderssen's Opening Polish Gambit: 1. a3 a5 2. b4
Anderssen's Opening Creepy Crawly Formation: 1. a3 e5 2. h3
d5
Anderssen's Opening Andersspike: 1. a3 g6 2. g4
==========
```

In fact, in the next version of your DSL program, you plan to write the output to a file and send the same to Victor. Why don't you fork this project and add-on some more functionality?

That's it!

If you have any questions, feel free to leave a comment on my blog post **here**.