MI-RUB Namespaces, Source Files, and Distribution Lecture 8

Pavel Strnad
pavel.strnad@fel.cvut.cz

Dept. of Computer Science, FEE CTU Prague,
Karlovo nám. 13, 121 35
Praha, Czech Republic

MI-RUB, WS 2011/12

Evropský sociální fond
Praha & EU: Investujeme do vaší budoucnosti

Contents

Namespaces

Namespaces

- Great way to organize code into functional units.
- Avoids name clashes.
- Code reusing is easier.

Constants

Constants Example

```
class Triangle
    SIDES = 3
    def area
        # ..
    end
end

class Square
    SIDES = 4
    def area
        # ...
    end
end
puts "A triangle has #{Triangle::SIDES} sides"
sq = Square.new(3)
puts "Area of square = #{sq.area}"
```

Constants

Constants

The double colon(::) is Ruby's namespace resolution operator. The thing to the left must be a class or module, and the thing to the right is a constant defined in that class or module.
The names of classes and modules are themselves just constants. Remember that we said that most everything in Ruby is an object. Well, classes and modules are, too. The name that you use for a class, such as String, is really just a Ruby constant containing the object representing that class.

Constants

```
module Formatters
    class Html
        # ...
    end
    class Pdf
        # ...
    end
end
html_writer = Formatters::Html.new
```

You can nest modules and classes inside other classes as you want. The depth more than 3 is rare.

Organizing Small Programs

Organizing Small Programs

- Small programs are usually in one file.
- You cannot automatically test your code.
- It is better to split a small program into functional parts to be able to reuse and test them.

Small Program Example

```
#!/usr/bin/env ruby
require 'optparse'
dictionary = "/usr/share/dict/words"
OptionParser.new do |opts|
    opts.banner = "Usage: anagram [ options ] word..."
    opts.on("-d", "--dict path", String, "Path to dictionary")
        do |dict|
          dictionary = dict
     end
     opts.on("-h", "--help", "Show this message") do
          puts opts
          exit
     end
```

Small Program Example

```
    begin
        ARGV << "-h"
         if ARGV.empty?
            opts.parse!(ARGV)
    rescue OptionParser::ParseError => e
        STDERR.puts e.message, "\n", opts
        exit(1)
    end
end
# convert "wombat" into "abmotw". All anagrams share a
    signature
def signature_of(word)
    word.unpack("c*").sort.pack("c*")
end

signatures = Hash.new
File.foreach(dictionary) do |line|
    word = line.chomp
    signature = signature_of(word)
    (signatures[signature] ||= []) << word
end
```

Small Program Example

```
ARGV.each do |word|
    signature = signature_of(word)
    if signatures[signature]
        puts "Anagrams of #{word}: #{signatures[signature].join
            (', ')}"
    else
        puts "No anagrams of #{word} in #{dictionary}"
    end
end

#usage
ruby anagram.rb teaching code
A
nagrams of teaching: cheating, teaching
Anagrams of code: code, coed
```

Small Program Example

Decomposition

Then someone asks me for a copy, and I start to feel embarassed.

- It has no tests, and
- it isn't particularly well packaged.

Small Program Decomposition Exercise

Small Exercise 1

Try to decompose the program mentioned above into a functional parts (5 mins).

Small Program Decomposition Exercise

Small Program Decomposition Exercise

There are many solutions. One of them is to decompose it into 4 parts:

- An option parser
- A class to hold the lookup table for anagrams
- A class that looks up words given on the command line
- A trivial command-line interface

The first three of these are effectively library files, used by the fourth.

Project Structure

```
anagram/ <toplevel
    bin/ <commandline interface goes here
    lib/ <three library files go here
    test/ <test files go here
```

Project Structure

```
anagram/
    bin/
        anagram <commandline interface
    lib/
        anagram/ Top level dir and module name
            finder.rb   Anagram::Finder
            options.rb  Anagram::Options
            runner.rb  Anagram::Runner
    test/
            ... various test files
```

We'll create just one top-level module, Anagram, and then place all our classes inside this module. This means that the full name of (say) our options-parsing class will be Anagram::Options.

Project Structure Exercise 2

Exercise 2

Let’s implement classes Anagram::Finder, Anagram::Options and Anagram::Runner. Write tests for them. Consider using *require_relative* method instead *require*. Hint: bin/anagram file:

```
#! /usr/local/rubybook/bin/ruby
require 'anagram/runner'
runner = Anagram::Runner.new(ARGV)
runner.run
```

Using setup.rb

setup.rb

Rather than write this script yourself, you could instead use Minero Aoki's setup.rb. Follow the download link from http://i.loveruby.net/en/projects/setup/, and you'll end up with a gzipped tarball. When you extract the files, you'll find a lot of documentation and other support material. But the key is the file setup.rb that you'll find in the top-level directory.

Using setup.rb

setup.rb

Copy file *setup.rb* into the top-level directory of our new application:

```
anagram/
    bin/
        anagram
    lib/
        anagram/
            finder.rb
            options.rb
            runner.rb
            setup.rb <installer
    test/
         ... various test files
```

RubyGems System

The RubyGems package management system (which is also just called Gems) has become the standard for distributing and managing Ruby code packages. As of Ruby 1.9, it comes bundled with Ruby itself.

```
gem search
gem install
gem list
gem server
```

For more information see the book p. 260.