Web Engineering:
Ruby from other languages

The University of Aizu
Quarter 2, AY 2018

Outline

- Running Ruby
- Ruby vs. Java
- View on Ruby from other languages

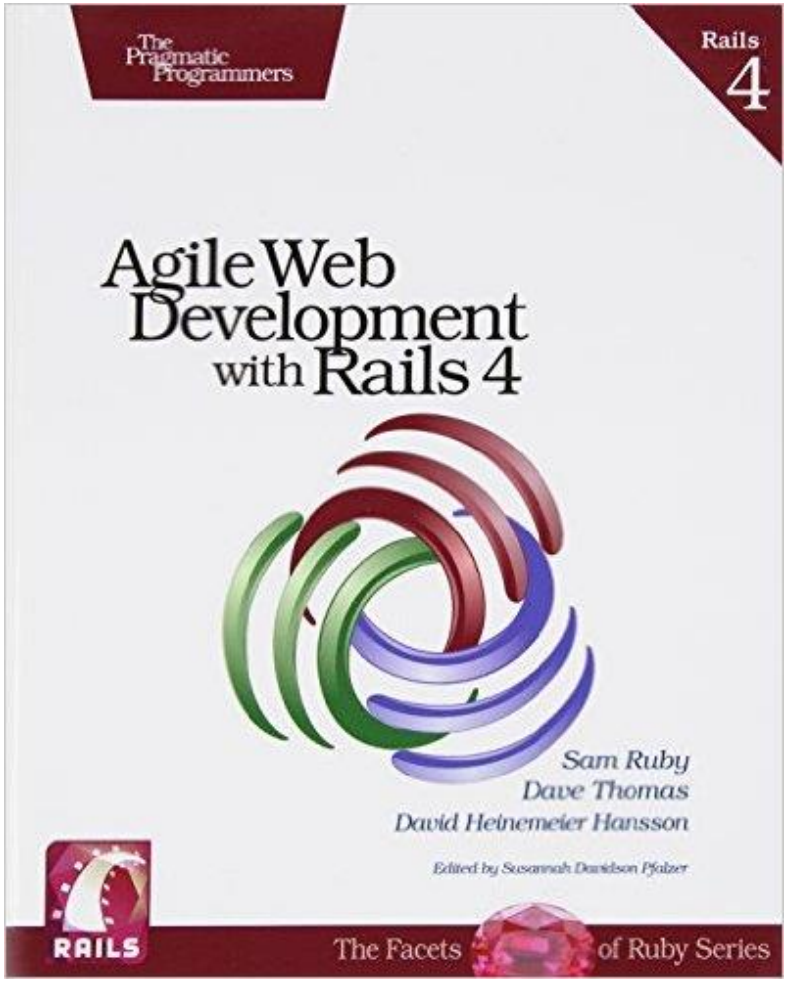
Literature

- Agile Web Development with Rails 4 (1st edition) by Sam Ruby, Dave Thomas and Devid Hansson, The Pragmatic Bookshelf, 2013.
- Web resources:

http://www.buildingwebapps.com/

http://www.ruby-lang.org/en/documentation/quickstart/

http://www.ruby-lang.org/en/documentation/ruby-from-other-languages/



Running Ruby

- Interactively
  - *irb*
  - To stop *irb*, use command *quit*
- In program file
  - *ruby file.rb*

```
prorsv1209{vkluev}109: which irb
/usr/local/bin/irb
prorsv1209{vkluev}110: irb
irb(main):001:0> 3+4
=> 7
irb(main):002:0> puts("Hello")
Hello
=> nil
irb(main):003:0> quit
prorsv1209{vkluev}111: ruby prime_ministers.rb
Prime Ministers of Japan in 21 century
1: Mori
2: Koizumi
3: Abe
4: Fukuda
5: Aso
6: Hatoyama
7: Kan
8: Noda
prorsv1209{vkluev}112:
```

prime_ministers.rb

```
#! /usr/local/bin/ruby
puts("Prime Ministers of Japan in 21 century")
prime_ministers_21_century = ["Mori", "Koizumi", "Abe", "Fukuda", "Aso", "Hatoyama", "Kan", "Noda" ]
prime_ministers_21_century.each_with_index { | value, ind | print ind+1, ": ", value, "\n" }
```

Ruby vs. Java

- **Similarities**, as with Java, in Ruby,...
  - Memory is managed for you via a garbage collector.
  - Objects are strongly typed.
  - There are public, private, and protected methods.
  - There are embedded doc tools (Ruby's is called RDoc). The docs generated by rdoc look very similar to those generated by javadoc.

Ruby vs. Java

- **Differences**, unlike Java, in Ruby,...
  - You don't need to compile your code. You just run it directly.
  - There are several different popular third-party GUI toolkits. Ruby users can try WxRuby, FXRuby, Ruby-GNOME2, or the bundled-in Ruby Tk for example.
  - You use the *end* keyword after defining things like classes, instead of having to put braces around blocks of code.
  - You have *require* instead of *import*.

Ruby vs. Java

- **Differences**, unlike Java, in Ruby,...
  - All member variables are private. From the outside, you access everything via methods.
  - Parentheses in method calls are usually optional and often omitted.
  - Everything is an object, including numbers like 2 and 3.14159.
  - There's no *static* type checking.
  - Variable names are just labels. They don't have a type associated with them.

Ruby vs. Java

- **Differences**, unlike Java, in Ruby,...
  - There are no type declarations. You just assign to new variable names as-needed and they just "spring up" (i.e. *a = [1,2,3]* rather than *int[] a = {1,2,3};*).
  - There's no casting. Just call the methods. Your unit tests should tell you before you even run the code if you're going to see an exception.
  - It's *foo = Foo.new("hi")* instead of *Foo foo = new Foo("hi");*.

Ruby vs. Java

- **Differences**, unlike Java, in Ruby,...
  - The constructor is always named "*initialize*" instead of the name of the class.
  - You have "*mixin's*" instead of interfaces.
  - YAML tends to be favored over XML.
  - It's *nil* instead of *null*.
  - == and *equals()* are handled differently in Ruby.
    - Use == when you want to test equivalence in Ruby (*equals()* is Java).
    - Use *equal?()* when you want to know if two objects are the same (== in Java).

View on Ruby from other languages

- Iteration
- Everything has a value
- Symbols are not lightweight Strings
- Everything is an Object
- Variable Constants
- Naming conventions
- Fake keyword parameters
- The universal truth
- Access modifiers apply until the end of scope
- Method access
- Classes are open
- Funny method names
- Missing methods
- Message passing, not function calls
- Blocks are Objects
- Overriding Operators

Iteration

- Two Ruby features that are a bit unlike what you may have seen before, and which take some getting used to, are "blocks" and iterators. Instead of looping over an index (like with C, C++, or pre-1.5 Java), or looping over a list (like Perl's *for (@a) {...}*, or Python's *for i in aList: ...*), with Ruby you'll very often instead see

```
some_list.each do |this_item|
  # We're inside the block.
  # deal with this_item.
end
```

Everything has a value

- There's no difference between an expression and a statement. Everything has a value, even if that value is **nil**. This is possible:

```
x = 10
y = 11
z = if x < y
  true
else
  false
end
z # => true
```

Symbols are not lightweight Strings

- Symbols can best be described as identities. A symbol is all about **who** it is, not **what** it is. Start *irb* and see the difference:

```
irb(main):001:0> :george.object_id == :george.object_id
=> true
irb(main):002:0> "george".object_id == "george".object_id
=> false
irb(main):003:0>
```

- The object_id methods returns the identity of an Object. If two objects have the same object_id, they are the same (point to the same Object in memory).

Symbols are not lightweight Strings

- Once you have used a Symbol once, any Symbol with the same characters references the same Object in memory. For any given two Symbols that represent the same characters, the *object_ids* match.
- Look at the String ("*george*"). *The object_ids* don't match. That means they're referencing two different objects in memory. Whenever you use a new String, Ruby allocates memory for it.
- If you're in doubt whether to use a Symbol or a String, consider what's more important: the identity of an object (i.e. a Hash key), or the contents (in the example above, "*george*").

Everything is an Object

- Even classes and integers are objects, and you can do the same things with them as with any other object:

```
# This is the same as
# class MyClass
# attr_accessor :instance_var
# end
MyClass = Class.new do
  attr_accessor :instance_var
end
```

Variable Constants

- Constants are not really constant. If you modify an already initialized constant, it will trigger a warning, but not halt your program. That isn't to say you **should** redefine constants, though.

Naming conventions

- Ruby enforces some naming conventions. If an identifier starts with a capital letter, it is a constant. If it starts with a dollar sign ($), it is a global variable. If it starts with @, it is an instance variable. If it starts with @@, it is a class variable.
- Method names, however, are allowed to start with capital letters. This can lead to confusion, as the example below shows:

```
Constant = 10
def Constant
  11
end
```

- Now *Constant* is 10, but *Constant()* is 11.

Fake keyword parameters

- Ruby doesn't have keyword parameters, like Python has. However, it can be faked by using symbols and hashes. Ruby on Rails, among others, uses this heavily.
- Example:

```
def some_keyword_params( params )
  params
end
some_keyword_params( :param_one => 10, :param_two => 42 )
# => {:param_one=>10, :param_two=>42}
```

The universal truth

- In Ruby, everything except **nil** and **false** is considered true. In C, Python and many other languages, 0 and possibly other values, such as empty lists, are consided false.
- Example:

```
# in Ruby
if 0
  puts "0 is true"
else
  puts "0 is false"
end
```

- Prints “0 is true”.

Access modifiers apply until the end of scope

```
class MyClass
  private
  def a_method; true; end
  def another_method; false; end
end
```

- You might expect another_method to be public. Not so. The 'private' access modifier continues until the end of the scope, or until another access modifier pops up, whichever comes first.
- By default, methods are public.

Method access

- Methods may be
  - Public (by default)
  - Private
  - protected
- Access rules are slightly different from Java
  - Examples see at (Section Method Access)

http://www.ruby-lang.org/en/documentation/ruby-from-other-languages/

Classes are open

- Ruby classes are open. You can open them up, add to them, and change them at any time. Even core classes, like *Fixnum* or even Object, the parent of all objects. Ruby on Rails defines a bunch of methods for dealing with time on *Fixnum*.

```
class Fixnum
  def hours
    self * 3600 # number of seconds in an hour
  end
  alias hour hours
end

# 14 hours from 00:00 January 1st
# (aka when you finally wake up ;)
Time.mktime(2006, 01, 01) + 14.hours # => Sun Jan 01 14:00:00
```

Funny method names

- Methods are allowed to end with question marks or exclamation marks. By convention, methods that answer questions (i.e. *Array#empty?* returns **true** if the receiver is empty) end in question marks. Potentially "dangerous" methods (ie methods that modify **self** or the arguments, *exit!* etc.) by convention end with exclamation marks.
- All methods that change their arguments don't end with exclamation marks, though. *Array#replace* replaces the contents of an array with the contents of another array. It doesn't make much sense to have a method like that that **doesn't** modify self.

Missing methods

- If Ruby can't find a method that responds to a particular message. It calls the *method_missing* method with the name of the method it couldn't find and the arguments.
- By default, *method_missing* raises a *NameError* exception, but you can redefine it to better fit your application.
- This code below prints the details of the call, but you are free to handle the message in any way.

```
# id is the name of the method called, the * syntax collects
# all the arguments in an array named 'arguments'
def method_missing( id, *arguments )
  puts "Method #{id} was called, but not found. It has " +
       "these arguments: #{arguments.join(", ")}"
end

__ :a, :b, 10
# => Method __ was called, but not found. It has these
# arguments: a, b, 10
```

Message passing, not function calls

- A method call is really a **message** to another object:

```
# This
1 + 2
# Is the same as this ...
1.+(2)
# Which is the same as this:
1.send "+", 2
```

Blocks are Objects

- Blocks (closures, really) are heavily used by the standard library. To call a block, you can either use *yield*, or make it a *Proc* by appending a special argument to the argument list, like so:

```
def block( &the_block )
  # Inside here, the_block is the block passed to the method
  the_block # return the block
end
adder = block { |a, b| a + b }
# adder is now a Proc object
adder.class # => Proc
```

Overriding operators

- Example:

```
class Fixnum
  # You can, but please don't do this
  def +( other )
    self - other
  end
end
```

Conclusion

- When you look at any new programming languages, you analyse it from the view of the languages you studed.
- We compared Ruby with Java.
- We looked at the key features of Ruby from the views of other languages.
- To feel the language, you should solve at least one progamming problem by yourself.
- To get quick understanding on how to use Ruby, use Google with the queries:
  - Ruby examples
  - Ruby examples code