HHLinuxClub

Ruby on Rails Course 2009

Ruby Basics

Sources and Authors:
Page 17

Edited by:
João Cardoso
Matias Korhonen

Table of Contents

Ruby Basics

Overview

Ruby is an object-orientated language first released in 1995 by Yukihiro Matsumoto ("Matz"). Ruby comes preinstalled on Mac OS X and on most major Linux distributions. It is also possible to install Ruby in a Microsoft Windows environment. The Ruby language takes fetures and influences from such languages as Perl, Smalltalk, Eiffel, Ada, Lisp, and Python.

Object Oriented

Ruby goes to great lengths to be a purely object oriented language. Every value in Ruby is an object, even the most primitive things: strings, numbers and even true and false. Every object has a class and every class has one superclass. At the root of the class hiearchy is the class Object, from which all other classes inherit.

Every class has a set of methods which can be called on objects of that class. Methods are always called on an object - there are no "class methods", as there are in many other languages (though Ruby does a great job of faking them).

Every object has a set of instance variables which hold the state of the object. Instance variables are created and accessed from within methods called on the object. Instance variables are completely private to an object. No other object can see them, not even other objects of the same class, or the class itself. All communication between Ruby objects happens through methods.

Mixins

In addition to classes, Ruby has modules. A module has methods, just like a class, but it has no instances. Instead, a module can be included, or "mixed in", to a class, which adds the methods of that module to the class. This is very much like inheritance but far more flexible because a class can include many different modules. By building individual features into separate modules, functionality can be combined in elaborate ways and code easily reused. Mix-ins help keep Ruby code free of complicated and restrictive class hiearchies.

Dynamic

Ruby is a very dynamic programming language. Ruby programs aren't compiled, in the way that C or Java programs are. All of the class, module and method definitions in a program are built by the code when it is run. A program can also modify its own definitions while it's running (commonly known as "monkey patching"). Even the most primitive classes of the language like String and Integer can be opened up and extended.

Variables in Ruby are dynamically typed, which means that any variable can hold any type of object. When you call a method on an object, Ruby looks up the method by name alone - it doesn't care about the type of the object. This is called duck typing and it lets you make classes that can pretend to be other classes, just by implementing the same methods.

Variables and scope

You do not need to declare variables or variable scope in Ruby. The name of the variable automatically determines its scope.

- x is local variable (or something besides a variable)
- $x is a global variable
- @x is an instance variable
- @@x is a class variable

Blocks

Blocks are one of Ruby's most unique and most loved features. A block is a piece of code that can appear after a call to a method, like this:

```
cities.each do |city|
  city.upcase!
end
```

The block is everything between the do and the end. The code in the block is not evaluated right away, rather it is packaged into an object and passed to the each method as an argument. That object can be called at any time, just like calling a method. The each method calls the block whenever it needs to iterate through the array. The block gives you a lot of control over how sort behaves. A block object, like any other object, can be stored in a variable, passed along to other methods, or even copied.

Many programming languages support code objects like this. They're called closures and they are a very powerful feature in any language, but they are typically underused because the code to create them tends to look ugly and unnatural. A Ruby block is simply a special, clean syntax for the common case of creating a closure and passing it to a method. This simple feature has inspired Rubyists to use closures extensively, in all sorts of creative new ways.

Ruby Editors

Ruby can be edited in almost any text editor and there are a number of (mostly) proprietary IDEs available for development in Ruby and Ruby on Rails.

Some examples of editors and IDEs:

- Komodo Edit
- Vim
- TextMate
- NetBeans
- Aptana Studio

Basics of Ruby

Irb

Irb (Interactive Ruby) is an interactive interface to Ruby which is mostly useful for testing small snippets of code. Using irb will save you from having to write lots of small text files when you are testing a feature or trying to see if a particular function will work.

You can also find an online version of irb at http://tryruby.sophrinix.com/

Datatypes

As mentioned earlier, everything in Ruby is an object. As a way of testing this try the following in irb:

```
123.class
999999999999999999999999999999999999999999999.class
nil.class
true.class
"hello world".class
:symbol.class
```

Constants

There are only two basic thing you need to remember about constants:

- Constants **always** start with capital letters (`Constant` would be a constant, but `constant` would not)
- You can change constant in Ruby, but you shouldn't (Ruby will give you a warning)

Symbols

So did you notice something weird about that previous code listing? "What the hell was that colon thingy about?" Well, it just so happens that Ruby's object oriented ways have a cost: lots of objects make for slow code. Every time you type a string, Ruby makes a new object. Regardless of whether two strings are identical, Ruby treats every instance as a new object. You could have `"live long and prosper"` in your code once and then again later on and Ruby wouldn't even realize that they're pretty much the same thing. Here is a sample irb session which demonstrates this fact :

```
irb> "live long and prosper".object_id
=> -606662268
irb> "live long and prosper".object_id
=> -606666538
```

Notice that the object ID returned by irb Ruby is different even for the same two strings.

To get around this memory hogging, Ruby has provided "symbols." Symbols are lightweight objects best used for comparisons and internal logic. If the user doesn't ever see it, why not use a symbol rather than a string? Your code will thank you for it.

Let us try running the above code using symbols instead of strings:

```
irb> :my_symbol.object_id
=> 150808
irb> :my_symbol.object_id
=> 150808
```

Symbols are denoted by using a colon prefix (`:symbol_name`).

Hashes

Hashes are like dictionaries, in a sense. You have a key, a reference, and you look it up to find the associated object, the definition.

The best way to illustrate this, I think, is simply to demonstrate:

```
hash = { :leia => "Princess from Alderaan", :han => "Rebel without a
cause", :luke => "Farmboy turned Jedi" }
puts hash[:leia]
puts hash[:han]
puts hash[:luke]
```

This code will print out "Princess from Alderaan", "Rebel without a cause", and "Farmboy turned Jedi" in consecutive order.

Arrays

Arrays are a lot like hashes, but the keys are always consecutive numbers. Also, the first key in an array is always 0. So in an array with 5 items, the last element would be found at `array[4]` and the first element would be found at `array[0]`.

Using an out of range index will return `nil`.

```
array = ["This", "is", "an array"]
puts array[0] # => "This"
puts array[3] # => nil
```

Strings

String literals

One way to create a `String` is to use single or double quotes inside a Ruby program to create what is called a string literal.

```
puts 'Hello world'
puts "Hello world"
```

Being able to use either single or double quotes is similar to Perl, but different from languages such as C and Java, which use double quotes for string literals and single quotes for single characters.

So what difference is there between single quotes and double quotes in Ruby? In the above code, there's no difference. However, consider the following code:

```
puts "Betty's pie shop"
puts 'Betty\'s pie shop'
```

Because "Betty's" contains an apostrophe, which is the same character as the single quote, in the second line we need to use a backslash to escape the apostrophe so that Ruby understands that the apostrophe is in the string literal instead of marking the end of the string literal. The backslash followed by the single quote is called an escape sequence.

Single quotes

Single quotes only support two escape sequences.

- `\'` – single quote
- `\\` – single backslash

Except for these two escape sequences, everything else between single quotes is treated literally.

Double quotes

Double quotes allow for many more escape sequences than single quotes. They also allow you to embed variables or Ruby code inside of a string literal – this is commonly referred to as interpolation.

```
name = "bob"
puts "Your name is #{name.capitalize}"
```

Escape sequences

The following escape sequences can be used inside double quotes:

- `\"` – double quote
- `\\` – single backslash
- `\a` – bell/alert
- `\b` – backspace
- `\r` – carriage return
- `\n` – newline
- `\s` – space
- `\t` – tab

Very Long Strings

There are a few ways to use very long (multiple lines) strings in Ruby. Perhaps the simplest of these is:

```
poem = %{My toast has flown from my hand
And my toast has gone to the
moon.
But when I saw it on television,
Planting our flag on Halley's
comet,
More still did I want to eat it.}
```

Numbers (Integers and Floats)

The basic numeric datatypes in Ruby are integers and floats.

The number operators in Ruby are pretty similar to operators in other modern languages:

- + addition
- - subtraction
- / division
- * multiplication
- ** exponent (e.g. 26**3 == 26^3)
- % modulus (the remainder of two divided numbers)

Methods

A method in Ruby is a set of expressions that returns a value. Other languages sometimes refer to this as a function. A method may be defined as a part of a class or separately.

Method Calls

Methods are called using the following syntax:

```
method_name(parameter1, parameter2,…)
```

If the method has no parameters the parentheses can usually be omitted as in the following:

```
method_name
```

If you don't have code that needs to use method result immediately, Ruby allows to specify parameters omitting parentheses:

```
results = method_name parameter1, parameter2
```

You need to use parentheses if you want to work with the result immediately.

```
results = method_name(parameter1, parameter2).reverse
```

Method Definitions

Methods are defined using the keyword `def` followed by the method name. Method parameters are specified between parentheses following the method name. The method body is enclosed by this definition on the top and the word end on the bottom. By convention method names that consist of multiple words have each word separated by an underscore.

```
def output_something(value)
  puts value
end
```

Return Values

Methods return the value of the last statement executed. The following code returns the value x+y.

```
def sum(x,y)
  x + y
end
```

An explicit return statement can also be used to return from a function, if necessary:

```
def authenticate(username)
  return unless username == "bob"
  puts "Welcome, Bob!"
end
```

Default Values

A default parameter value can be specified during method definition to replace the value of a parameter if it is not passed into the method or the parameter's value is `nil`.

```
def greet(username="anonymous")
  puts "Hello " + username + "!"
end
```

Classes

Classes are the basic template from which object instances are created. A class is made up of a collection of variables representing internal state and methods providing behaviors that operate on that state.

Class Definition

Classes are defined in Ruby using the `class` keyword followed by a name. The name must begin with a capital letter and by convention names that contain more than one word are run together with each word capitalized and no separating characters (CamelCase). The class definition may contain method, class variable, and instance variable declarations as well as calls to methods that execute in the class context, such as `attr_accessor`. The class declaration is terminated by the end keyword.

```
class SomeClass
  def some_method
  end
end
```

Instance Variables

Instance variables are created for each class instance and are accessible only within that instance or through the methods provided by that instance. They are accessed using the @ operator.

```
class Person
  @name = "Bob"

  def greet
    "Welcome " + @name + "!"
  end

  def change_to_fred
    @name = "Fred"
  end
end

person = Person.new
puts person.greet # => "Welcome Bob!"

person.change_to_fred
puts person.greet # => "Welcome Fred!"
```

Instantiation

An object instance is created from a class through the a process called instantiation. In Ruby this takes place through the Class method new.

Example:

```
person = Person.new("Joe")
```

This function sets up the object in memory and then delegates control to the initialize function of the class if it is present. Parameters passed to the new function are passed into the initialize function.

```
class Person
  def initialize(name)
    @name = name
  end
end
```

Attribute Accessors

Attribute accessors provide an easy way to read and write instance variables. This means that you do not need to write your own getter and setter methods.

```
class Person
  attr_accessor :name
end
```

The previous example could also be written as:

```
class Person
  def name
    @name
  end

  def name=(name)
    @name = name
  end
end
```

The Ruby language has three attribute accessors:

- `attr_accessor` will give you get/set functionality
- `attr_reader` will give only getter
- `attr_writer` will give only setter

Inheritance

A class can inherit functionality and variables from a superclass, sometimes referred to as a parent class or base class. Ruby does not support multiple inheritance and so a class in Ruby can have only one superclass.

```
class Person
  attr_accessor :name
end

class Employee < Person
  attr_accessor :position
end
```

Access Control

By default, all methods in Ruby classes are public – accessible by anyone. If desired, this access can be restricted by public, private, protected object methods. It is interesting that these are not actually keywords, but actual methods that operate on the class, dynamically altering the visibility of the methods.

As a result of that fact these “keywords” influence the visibility of all following declarations until a new visibility is set or the end of the declaration-body is reached.

- `public` is the default accessibility level for class methods
- `protected` methods are accessible also to child objects
- `private` methods are accessible only to the class that defines them

Example:

```
class Person
  def name
    @name
  end

  protected
    def name=(name)
      @name = name
    end

  private
    def change_to_bob
      @name = "Bob"
    end
end
```

Comments

Single line comments are denoted using a hash character (#):

```
# this is a comment line
```

Operators

Assignment

Assignment in Ruby is done using the equals sign (=). This is both for variables and objects, but since strings, floats, and integers are actually objects in Ruby, you're always assigning objects.

Examples:

```
colour = "red"
other_colour = String.new("blue")
number = 123
```

Self assignment

```
x = 1           # => 1
x += x          # => 2
x -= x          # => 0
x += 4          # => 4
x *= x          # => 16
x **= x         # => 18446744073709551616 # Raise to the power
x /= x          # => 1
```

If you're used to C-type languages, you might be wondering where are the increment and decrement operators (++ and `--`), well in Ruby the closest equivalents are `x += 1` and `x -= 1`.

Multiple assignments

In Ruby you can assign values to multiple variables on one line:

```
colour1, colour2, colour3 = "red", "blue", "green"
colours, city = ["red", "blue", "green"], "Helsinki"
```

Conditional assignment

The operator `||=` checks whether a variable is nil (or inexistent) and assigns an object to it if it is.

```
configuration ||= "default value"
```

Comparison Operators

Operator	Description
==	Equivalency
!=	Inequality
<	Less than
>	Greater than
>=	Greater than or equal to
<=	Less than or equal to
<=>	Combined comparison operator. Returns: • 0 if the values are equal • -1 if the second is greater than the first • 1 if the first is greater than the second

Examples:

```
5 == 5      # => true
5 != 4      # => true
4 > 5       # => false
4 < 5       # => true
5 <=> 5     # => 0
5 <=> 2     # => 1
5 <=> 6     # => -1
```

And (&&), Or (||), and Not (!)

Both `and` and && evaluate to `true` only if both operands are true, their only difference being their precedence.

Both `or` and `||` evaluate to `true` if either operand is true, again their only difference is in operator precedence.

As you might guess, both `not` and `!` only differ in precedence.

As a general rule you should only use the **non-word** operators (&&, `||`, and `!`) as they have a higher precedence than the word operators (`and`, `or`, `not`).

Examples:

```
x = false
y = true

x && y    # => false
x || y    # => true
!x        # => true
!y        # => false
```

Conditionals and Flow Control

Ruby can control the execution of code using conditional branches. A conditional branch takes the result of a test expression and executes a block of code depending whether the test expression is true or false. If the test expression evaluates to the constant `false` or `nil`, the test is false; otherwise, it is true. Note that the number zero is considered true, whereas many other programming languages consider it false.

IF, ELSIF, ELSE

The behaviour of `if` .. `elsif` .. `else` in Ruby is fairly consistant with most major programming languages, as seen in the following examples:

```
a = 3
if a == 4
  a = 7
elsif a < 0
  a = 0
else
  a = 5
end
print a # => 5
```

If the block executed as result of a test expression is only one line of code, you can write the if-statement after the code. For example:

```
a = 5
a = 7 if a == 4
print a # => 5
```

UNLESS

The unless-expression is the opposite of the if-expression, the code-block it contains will only be executed if the test expression is false.

```
my_city = "Helsinki"
unless my_city == "Helsinki"
  print "I don't live in Helsinki"
end
```

As with the short if-statement shown before, the previous example could be written using only two lines of code:

```
my_city = "Helsinki"
print "I don't live in Helsinki" unless my_city == "Helsinki"
```

CASE

The case expressions are very useful when testing the same variable multiple times.

```
case n
  when 0
    puts 'You typed zero'
  when 1, 9
    puts 'n is a perfect square'
  when 2
    puts 'n is a prime number'
    puts 'n is an even number'
  when 3, 5, 7
    puts 'n is a prime number'
  when 4, 6, 8
    puts 'n is an even number'
  else
    puts 'Only single-digit numbers are allowed'
end
```

The Ruby Ternary Operator

Ruby has a handy shorhand operator for making decisions:

```
[condition] ? [true expression] : [false expression]
```

This means that writing this:

```
customer = "Bob"
puts customer == "Bob" ? "Hello Bob!" : "You're not Bob!"
```

is equivalent to:

```
customer = "Bob"

if customer == "Bob"
  puts "Hello Bob!"
else
  puts "You're not Bob!"
end
```

Loops

WHILE

The while statement in Ruby is very similar to if and to other languages' while (syntactically):

```
while <expression>
  <...code block...>
end
```

The code block will be executed again and again, as long as the expression evaluates to true.

UNTIL
The until statement is similar to the while statement in functionality.

```
until <expression>
  <...code block...>
end
```

Unlike the while statement, the code block for the until loop will execute as long as the expression evaluates to `false`.

FOR
Example:

```
for i in 1..10 do
  puts i
end
```

TIMES
Example:

```
# Display 'Hello' 10 times
10.times do
  puts "Hello"
end

# Display powers of two
10.times do |i|
  puts 2**i
end
```

BREAK
The `break` keyword causes the loop or conditional in which it appears to exit at that point.

RETURN
`return value` causes the method in which it appears to exit at that point and return the value specified. Just calling `return` without supplying an argument will return `nil`.

Ruby Ranges
Ranges are used in Ruby to create a range of successive values:

```
1..5    # Creates a range from 1 to 5, inclusive
1...5   # Creates a range from 1 to 5, exclusive
```

Ranges can also be used with strings, for example:

```
letters = 'a'..'z'
words = 'cab'..'car'
```

Sources

Wikibooks

Most of the text was copied verbatim from the Wikibooks “Ruby Programming” online book. Some code examples were rewritten and some sections were written from scratch.

http://en.wikibooks.org/wiki/Ruby_Programming

Other

The sources below were only used for reference and no content was copied.

- Techotopia: Ruby Essentials. http://www.techotopia.com/index.php/Ruby_Essentials
- Programming Ruby 1.9: The Pragmatic Programmers' Guide. Dave Thomas, with Chad Fowler and Andy Hunt. ISBN-13: 978-1-93435-608-1. http://pragprog.com/titles/ruby3/programming-ruby-1-9
- Learn to Program. Chris Pine. http://pine.fm/LearnToProgram/