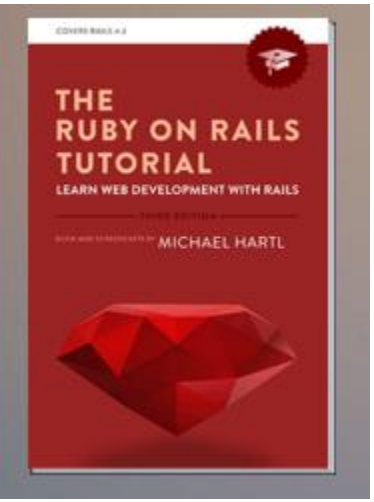
Ruby on Rails

Week 4

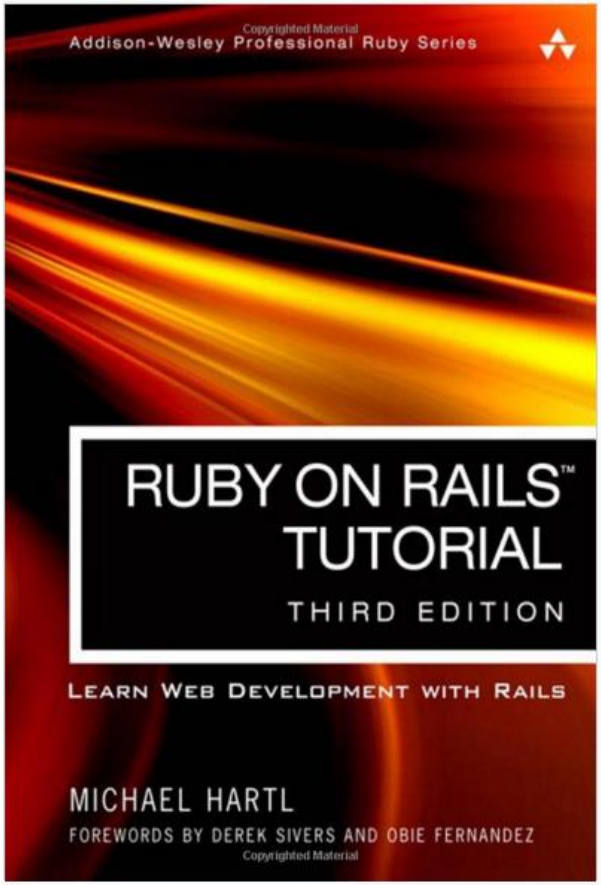
Ruby

Sources



https://www.railstutorial.org/book

http://www.amazon.co.uk/Ruby-Rails-Tutorial-Addison-Wesley-Professional/dp/0134077709/ref=sr_1_1?ie=UTF8&qid=1456138526&sr=8-1&keywords=ruby+on+rails+tutorial



Comments

Comments in Ruby start with the pound sign #.

```
# This is  comment in ruby.
def helloWorld(param)
.
.
.
end
```

Strings

- One of the most important data type for Web Development.
- Strings can use both single and double quotes, in most cases are the same.
- Example Use (Double Quotes):

"foo" + "bar"
=>"foobar"

x="hello"
...
y="world"
...
x + " " + y
⇒ Hello world

"#{x} my beautiful #{y}" *Interpolation, no need to add space like above.*
=> Hello my beautiful World

Strings

- Single quotes strings are also identical to double quotes, except they are entirely literal. They do not support interpolation so are useful when you want to print something that is a special case, like a newline (\n), but don't want to escape it.

- Example Use (Single Quotes):

'foo'+ 'bar'
=>"foobar"

x='hello'
...
y='world'
...
x + " " + y
⇒ Hello world

"#{x} #{y}" *Interpolation, does not work with Single Quoted strings*
=> "#{x} #{y}"

Print

- Most common way is to use the **puts** command.
- Puts automatically appends a new line.
- Notice the **nil** below. This is because the puts command returns nothing.

```
>>puts "foo bar"
foo bar
=>nil
```

Print

- The **print** command can also be used, however, it does not append a new line.
- Puts automatically appends a new line.
- Notice the **nil** below. This is because the puts command returns nothing.

```
>>print “foo bar”
foo bar => nil
```

Objects

- Everything in Ruby is an object.

- This includes:
  - Strings
  - Integers
  - Nil

- Objects respond to message or method calls.
- These can even be called on string literals:

```
>>"foobar".length
=> 6
```

Objects

- Can run checks on objects. Such as, checking if a string is empty:

```
>>“foobar”.empty?
=> false
>> “”.empty?
=> true
```

- Don’t forget the **?** In the above. This states the returned value is a boolean.

if / elsif examples:

```
>> s = "foobar"
>> if s.empty?
>>   "The string is empty"
>> else
>>   "The string is nonempty"
>> end
=> "The string is nonempty"
```

```
>> if s.nil?
>>   "The variable is nil"
>> elsif s.empty?
>>   "The string is empty"
>> elsif s.include?("foo")
>>   "The string includes 'foo'"
>> end
=> "The string includes 'foo'"
```

```
puts "x is not empty" if !x.empty?
```

Method Definition

```
>> def string_message(str = '')
>>   if str.empty?
>>     "It's an empty string!"
>>   else
>>     "The string is nonempty."
>>   end
>> end
=> :string_message
>> puts string_message("foobar")
The string is nonempty.
>> puts string_message("")
It's an empty string!
>> puts string_message
It's an empty string!
```

Default argument.

Implicit return type.

```
>> def string_message(str = '')
>>   return "It's an empty string!" if str.empty?
>>   return "The string is nonempty."
>> end
```

Explicit return.

Arrays

- List of elements in order.
- Ruby has a nice easy way to go from a string of words to an Array.

```
>>  "foo bar     baz".split     # Split a string into a three-element array.
=> ["foo", "bar", "baz"]
```

- Can split by a given char (note char not included):

```
>> "fooxbarxbazx".split('x')
=> ["foo", "bar", "baz"]
```

Arrays (Continued)

- As with most programming languages, Arrays start at index 0.

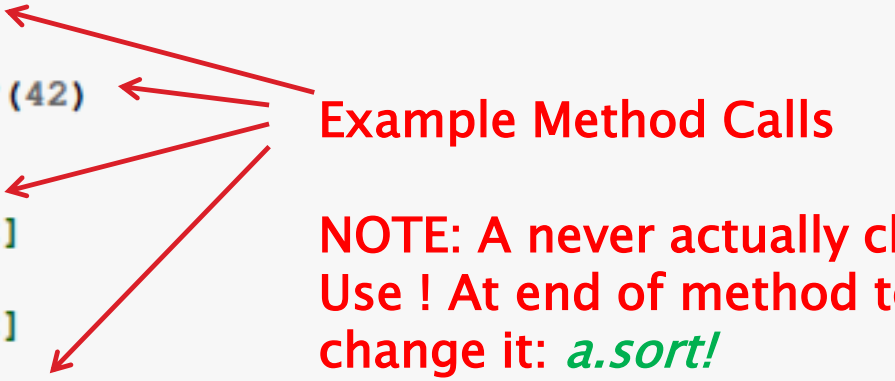
```
>> a = [42, 8, 17]
=> [42, 8, 17]
>> a[0]                 # Ruby uses square brackets for array access.
=> 42
>> a[1]
=> 8
>> a[2]
=> 17
>> a[-1]                # Indices can even be negative!
=> 17
```

```
>> a                    # Just a reminder of what 'a' is
=> [42, 8, 17]
>> a.first
=> 42
>> a.second
=> 8
>> a.last
=> 17
>> a.last == a[-1]      # Comparison using ==
=> true
```

Example Method Calls

Arrays (Continued)

```
>> a
=> [42, 8, 17]
>> a.empty?
=> false
>> a.include?(42)
=> true
>> a.sort
=> [8, 17, 42]
>> a.reverse
=> [17, 8, 42]
>> a.shuffle
=> [17, 42, 8]
>> a
=> [42, 8, 17]
```



Example Method Calls

NOTE: A never actually changes. Use ! At end of method to change it: *a.sort!*

Arrays (Continued)

- Adding Elements:

```
>> a.push(6)                    # Pushing 6 onto an array
=> [42, 8, 17, 6]
>> a << 7                       # Pushing 7 onto an array
=> [42, 8, 17, 6, 7]
>> a << "foo" << "bar"          # Chaining array pushes
=> [42, 8, 17, 6, 7, "foo", "bar"]
```

- Joining Elements

```
>> a
=> [42, 8, 17, 7, "foo", "bar"]
>> a.join                       # Join on nothing.
=> "428177foobar"
>> a.join(', ')                 # Join on comma-space.
=> "42, 8, 17, 7, foo, bar"
```

Ranges

- Adding Range to Array (use parentheses):

```
>> 0..9
=> 0..9
>> 0..9.to_a              # Oops, call to_a on 9.
NoMethodError: undefined method `to_a' for 9:Fixnum
>> (0..9).to_a            # Use parentheses to call to_a on the range.
=> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

- Call Array Range:

```
>> a = %w[foo bar baz quux]          # Use %w to make a string array.
=> ["foo", "bar", "baz", "quux"]
>> a[0..2]
=> ["foo", "bar", "baz"]
```

- Ranges and Characters:

```
>> ('a'..'e').to_a
=> ["a", "b", "c", "d", "e"]
```

Blocks

- Adding Range to Array (use parentheses):

```
>> (1..5).each do |number|
?>   puts 2 * number
>>   puts '--'
>> end
2
--
4
--
6
--
8
--
10
--
=> 1..5
```

Blocks

```
>> 3.times { puts "Betelgeuse!" }    # 3.times takes a block with no variables.
"Betelgeuse!"
"Betelgeuse!"
"Betelgeuse!"
=> 3
>> (1..5).map { |i| i**2 }           # The ** notation is for 'power'.
=> [1, 4, 9, 16, 25]
>> %w[a b c]                         # Recall that %w makes string arrays.
=> ["a", "b", "c"]
>> %w[a b c].map { |char| char.upcase }
=> ["A", "B", "C"]
>> %w[A B C].map { |char| char.downcase }
=> ["a", "b", "c"]
```

Hashes / Symbols

```
>> user = {}                                # {} is an empty hash.
=> {}
>> user["first_name"] = "Michael"           # Key "first_name", value "Michael"
=> "Michael"
>> user["last_name"] = "Hartl"              # Key "last_name", value "Hartl"
=> "Hartl"
>> user["first_name"]                       # Element access is like arrays.
=> "Michael"
>> user                                     # A literal representation of the hash
=> {"last_name"=>"Hartl", "first_name"=>"Michael"}
```

Hashes / Symbols

- Special kind of Ruby data type.
- In Rails, Symbols are used in Hashes far more often than Strings.
- Symbols similar to strings, but are prefixed with a colon (:)
  Eg: :name
- Has to start with a letter.

```
>> "name".split('')
=> ["n", "a", "m", "e"]
>> :name.split('')
NoMethodError: undefined method `split' for :name:Symbol
>> "foobar".reverse
=> "raboof"
>> :foobar.reverse
NoMethodError: undefined method `reverse' for :foobar:Symbol
```

Hashes / Symbols

- Example of Symbols in use in Hashes:

```
>> user = { :name => "Michael Hartl", :email => "michael@example.com" }
=> {:name=>"Michael Hartl", :email=>"michael@example.com"}
>> user[:name]              # Access the value corresponding to :name.
=> "Michael Hartl"
>> user[:password]          # Access the value of an undefined key.
=> nil
```

- Because of the common use of symbols as keys Ruby introduced an easier syntax for Symbols:

```
>> h1 = { :name => "Michael Hartl", :email => "michael@example.com" }
=> {:name=>"Michael Hartl", :email=>"michael@example.com"}
>> h2 = { name: "Michael Hartl", email: "michael@example.com" }
=> {:name=>"Michael Hartl", :email=>"michael@example.com"}
>> h1 == h2
=> true
```

OLD

NEW – Since v1.9

Note the position of the Colon in the two versions

Hashes / Symbols

- Iterating through a Hash with Blocks:

```
>> flash = { success: "It worked!", danger: "It failed." }
=> {:success=>"It worked!", :danger=>"It failed."}
>> flash.each do |key, value|
?>   puts "Key #{key.inspect} has value #{value.inspect}"
>> end
Key :success has value "It worked!"
Key :danger has value "It failed."
```

Hashes / Symbols

- The inspect method returns a literal interpretation of the object, brackets (arrays) and quotes (strings) included:

```
>> puts (1..5).to_a             # Put an array as a string.
1
2
3
4
5
>> puts (1..5).to_a.inspect     # Put a literal array.
[1, 2, 3, 4, 5]
>> puts :name, :name.inspect
name
:name
>> puts "It worked!", "It worked!".inspect
It worked!
"It worked!"
```

- So common, there is a shortcut:

```
>> p :name              # Same output as 'puts :name.inspect'
:name
```

Classes

- Example Class:

```
>> class Word
>>   def palindrome?(string)
>>     string == string.reverse
>>   end
>> end
=> :palindrome?
```

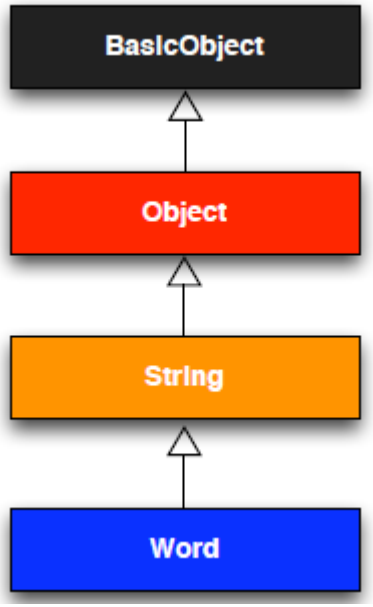
- Usage:

```
>> w = Word.new                 # Make a new Word object.
=> #<Word:0x22d0b20>
>> w.palindrome?("foobar")
=> false
>> w.palindrome?("level")
=> true
```

Classes – Inheritance

- Example Class:

```
class Word < String              # Word inherits from String.
  # Returns true if the string is its own reverse.
  def palindrome?
    self == self.reverse         # self is the string itself.
  end
end
nil
```



- Usage:

```
>> s = Word.new("level")     # Make a new Word, initialized with "level".
=> "level"
>> s.palindrome?             # Words have the palindrome? method.
=> true
>> s.length                  # Words also inherit all the normal string methods.
=> 5
```

```
>> s.class
=> Word
>> s.class.superclass
=> String
>> s.class.superclass.superclass
=> Object
```

Classes – Modify Built in Classes

```
>> "level".palindrome?
NoMethodError: undefined method `palindrome?' for "level":String
```

- Modify String Class:

```
>> class String
>>   # Returns true if the string is its own reverse.
>>   def palindrome?
>>     self == self.reverse
>>   end
>> end
=> nil
>> "deified".palindrome?
=> true
```