



Software Engineering
Rochester Institute
of Technology

Introduction to Ruby

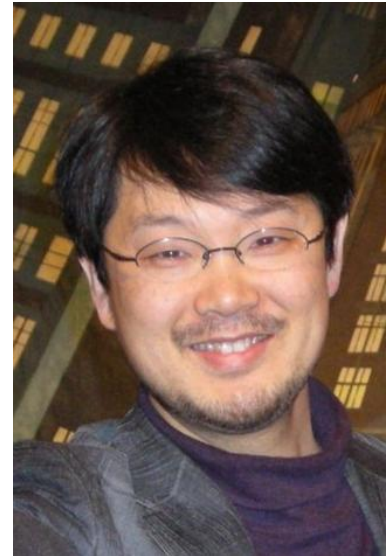
SWEN-250

Personal Software Engineering



A Bit of History

- Yukihiro "Matz" Matsumoto
 - Created a language he liked to work in.
 - Been around since mid-90s.
 - Caught on in early to mid 00s.
- Lineage
 - Smalltalk – dynamic, OO-centric
 - CLU – yield to blocks
 - Pascal – basic concrete syntax
 - AWK / Python / Perl – scripting & regular expressions
 - Matz's own predilections





Ruby Characteristics

- Everything is an object – *everything*.
 - `3.times { puts "hello" }`
 - `"Mike is smart".sub(/Mike/, "Pete")`
 - `str = str[0..9]` unless `str.length < 10`
- Every statement is an expression:
 - Generally the last value computed.
 - No need for `return` – but it's there anyway.
- Rich built in data types:

String

Array

Hash

RegExp

Range

Unbounded numbers (factorial)

Blocks & procs

Anonymous functions



Software Engineering
Rochester Institute
of Technology

Exploring Ruby

- ri – Ruby information
- irb – Interactive Ruby
- Script files: *filename.rb*



Ruby Control Structures: Selection

Software Engineering
Rochester Institute
of Technology

```
if condition  
  statements  
elsif condition  
  statements  
else  
  statements  
end
```

```
unless condition  
  statements  
end
```

Conditions in Ruby

Comparisons, etc., return a boolean:
true (the only member of TrueClass)
false (the only member of FalseClass)

Evaluating conditions

false evaluates to false.

nil evaluates to false.

Everything else is **true** (including 0).

Statement Modifiers (a la Perl)

```
statement if condition
```

```
statement unless condition
```



Ruby Control Structures: Loops

```
while condition  
  statements  
end
```

```
begin  
  statements  
end while condition
```

```
until condition  
  statements  
end
```

```
begin  
  statements  
end until condition
```

```
Early Termination  
next  
break  
redo
```

We don't need
no stinkin'
loops!



Iterators

- Explicit loops are rare in Ruby
- Instead, we usually use iterators
 - Iterators are defined on collection classes
 - "Push" elements into a block one at a time.
 - The basic iterator is **each**.
 - Show with arrays (the simplest collection)

```
fibonacci = [ 1, 2, 3, 5, 8 ]
```

```
fibonacci.each { | value | puts "The next value is #{value}" }
```

```
fibonacci.each_index { | i | puts "fibonacci[#{i}] = #{fibonacci[i]}" }
```

```
fibonacci.select { | value | value % 2 == 1 }
```

```
fibonacci.inject(0) { | sum, value | sum += value }
```

```
puts "Total = #{fibonacci.inject(0) { | s, v | s += v } }"
```



But, For Completeness

Software Engineering
Rochester Institute
of Technology

- loop

```
loop { puts "forever" }
```

```
loop do
```

```
  line = gets
```

```
  break if ! line
```

```
  puts line
```

```
end
```

- for statement

```
for v in  
  collection  
  statements  
end
```



```
collection.each do | v |  
  statements  
end
```




puts "Betty's pie shop" vs. puts 'Betty\'s pie shop'

Strings

- Literals

"abcdef" vs. 'abcdef' %q{xyz#{1}} → non-interpolate String
"abc #{3 % 2 == 1} def" %Q{xyz#{1}} → interpolate String

- Operators

+ and += s1 = "a" + "b" ; s1 += "c"
* "oops! " * 3
[] should be obvious, but "abcd"[1..2]
== < <=> comparisons
=~ and !~ r.e. match (and not match)

- Some of the methods (many have ! variants)

capitalize sub(*r.e*, *str*)
downcase include?(*str*)
upcase index(*str* or *r.e.*)



Arrays

- Literals

`a = [1, "foo", [6, 7, 8], 9.87]`

`b = %w{ now is the time for all good men }` → Interpolated array of words

- Operators

`&` (intersection) `+` (catenation) `-` (difference)

`*` *int* (repetition) `*` *str* (join w/*str* as separator)

`[] []=` as expected for simple indices

`<< obj` (push on end)

- Some of the methods

`[1, "hello", 3].collect { |v| v * 2 }` # alias map

`[1, 2, 5].include?(2)`

`[1, 2, 5].first` `[1, 2, 5].last`

`[1, 2, 5].length` `[1, 2, 5].empty?`



Hashes

- Literals

```
{ "door" => "puerta", "pencil" => "lapiz" }  
new Hash( default )
```

- Operators

```
h[key]          h[key] = value
```

- Some methods

```
each      each_key      each_value  
empty?    has_key?        has_value?    size  
keys (returns array)      values (returns array)  
sort (returns an array of 2-element arrays)  
sort { |p1, p2| expression returning -1, 0, +1 }
```



I/O

- Class File

`f = File.new(name, mode)`

- *name* is a string giving the file name (host dependent).
- *mode* is an access string: "r", "rw", "w", "w+"

`f.close`

`f.puts`, `f.printf`, `f.gets`, etc.

- `puts`, `printf` are implicitly prefixed by `$stdout`.
- `gets` is implicitly prefixed by `$stdin`

`File.open(name, mode) block` – open the file *name*, call *block* with the open file, close file when block exits.

- Class Dir

`d = Dir.new(name)` – open named directory.

`d.close`

`Dir.foreach(name) block` – pass each file name to *block*.



RegExps

- Literals

/regular expression/

%r@regular expression@

/regular expression/i

delimiter is @

case insensitive

- Resource

https://www.tutorialspoint.com/ruby/ruby_regular_expressions.htm

- Rubular <http://rubular.com/>



RegExps Examples

```
'Some cats here'.gsub(/cats/, 'dogs')
```

```
'xxAAyyBBzz'.gsub(/A+[^B]*B+/, '\&<->\&')
```

```
'xxAAyyBBzz'.gsub(/(A+)([^B]*)(B+)/, '\3\2\1')
```

```
'xx(AA)Azz'.gsub(/\(A+\)/, '###')
```



Miscellaneous (1)

- Functions

- call: `puts "abc"` or `puts("abc")`

- define:

- ```
def putNtimes(string, count)
```

- ```
  puts string * count
```

- ```
end
```

- Requiring modules

- ```
require string
```

- Looks for *string*.rb and imports whatever is in there.
 - Typically service functions, classes, etc.
 - Looks in "standard" locations as well as current directory.

- Example: **require 'pp'**

- Makes a function **pp** available.
 - Similar to **puts**, but presents structures in a nested, easier to read format.



Miscellaneous

- Symbols
 - :foobar, :myname
 - like a string but unique, immutable, and fast
 - Often used as hash keys, identifiers, etc.
- Duck typing: "If it looks like a duck . . ."

```
def putlengths anArray
  anArray.each { |x| puts x.length }
end
putlengths [ [1, 2, 3], "abcde", {"a" => "b", "c" => "d"} ]
```




Software Engineering
Rochester Institute
of Technology

ON TO THE ACTIVITY