CMSC 330: Organization of Programming Languages

Introduction to Ruby

Last Lecture
- Many types of programming languages
  - Imperative, functional, logical, OO, scripting
- Many programming language attributes
  - Clear, orthogonal, natural...
- Programming language implementation
  - Compiled, interpreted

Introduction
- Ruby is an object-oriented, imperative scripting language
  - “I wanted a scripting language that was more powerful than Perl, and more object-oriented than Python. That’s why I decided to design my own language.”
  - “I believe people want to express themselves when they program. They don’t want to fight with the language. Programming languages must feel natural to programmers. I tried to make people enjoy programming and concentrate on the fun and creative part of programming when they use Ruby.”
    - Yukihiro Matsumoto (“Matz”)

Books on Ruby
- Earlier version of Thomas book available on web
  - See course web page
Applications of Scripting Languages

- Scripting languages have many uses
  - Automating system administration
  - Automating user tasks
  - Quick-and-dirty development

- Major application
  - Text processing

Output from Command-Line Tool

```
$ wc *
271     674    5323 AST.c
100     392    3219 AST.h
117    1459  23878 AST.o
810    2328  24589 AST_defs.c
640    3070   33530 AST_defs.h
371     884    9483 AST_parent.c
884    2459   28758 AST_print.c
443    1589  22587 Makefile
271     725    5578 Makefile.am
271     725    5578 Makefile.in
38     175    1154 alloca.c
2035    4516   47721 aloctypes.c
86     350    3286 aloctypes.h
104    1051   66848 aloctypes.o
```

Climate Data for IAD in August, 2005

```
AVG MX 2MIN
DY MAX MIN AVG DEP HDD CDD  WTR  SNW DPTH SPD SPD DIR MIN PSBL S
- S WX    SPD DR
================================================================================
1  87  66  77   1   0  12 0.00  0.0    0  2.5  9 200   M    M   7 18     12 210
2  92  67  80   4   0  15 0.00  0.0    0  3.5 10 10   M    M   3 18     17 320
3  93  69  81   5   0  16 0.00  0.0    0  4.1 13 360   M    M   3 18     17 360
4  95  69  82   6   0  17 0.00  0.0    0  3.6  9 210   M    M   3 18     12 230
5  94  73  84   8   0  19 0.00  0.0    0  5.9 18 10   M    M   3 18     25 360
6  89  70  80   4   0  15 0.00  0.0    0  5.3 20 200   M  6 138     23 210
7  89  69  79   3   0  14 0.00  0.0    0  3.6 9 18   M    M   7 1     16 210
8  86  70  78   3  0 13 0.74  0.0    0  4.4 17 150   M  10  18     23 150
9  78  70  73  -2   0  9 18 0.00  0.0    0  2.3 9 18   M    M   9 18     13 90
10 87  71  79  4  0 14 0.00  0.0    0  2.3 8 240   M    M   8 1      10 210
```

Raw Census 2000 Data for DC

```
u108_S,DC,000,01,0000001,572059,72264,572059,12.6,572059,572059,572059,0,0,
0,0,572059,175306,343213,2006,14762,383,21728,14661,572059,572054,4,15861
7,340041,1560,14605,291,1438,10272,4,5015,5,1689,3,152,466,157,92,2,0090,43
89,572059,266827,3562,3048,3170,3241,3504,3286,3270,3475,3959,3647,3525
,5044,2928,2913,2769,2752,2933,2703,4056,5501,51,52,7,4969,13555,24,995,242
16,23726,20721,38902,1653,12,318,4365,58,10,34,24,400,7125,5739,3,260,234
7,303232,3329,3205,2935,3,429,3326,34,56,3237,3754,3192,3523,3336,2726,2729
89,2838,2834,2624,2807,2871,6,941,6568,56,25,5,5663,1717,7,27475,243777,2,2818
31339,29651,29117,15120,15606,6708,6,257,6117,107,4,9247,6807,6175,5705
9,536733,370675,151933,5,563,6030,6030,5,7949,122549,122518,7,354,3168,22448,
9967,44043,14105,16160,145898,10169,4,7694,123353,71357,6,07805,10703,320,71
35864,7573,28113,248590,1,08569,47694,6085,75,140023,11,5963,56050,2,1654,36
396,59723,10355,4045,6290,47558,52329,22,329,24059,5,13,355,10703,7,0086,657
37,371112,23742,12267,9470,972,2573,2314,760,28625,8207,746,738,19183,
18172,1015,12335,6310,741,248590,13954,4922,4,2674,21443,24,831,479
47,8705,3979,4726,38262,5,3175,16067,10523,82,8228,2237,4,9134,217,62,1177
6,211,11565,966,1650,86,3564,8316,54,82,62,2,7392,256,41,7,1851,4750,90,1159
65,49893,24466,24165,24062,16529,12549,7594,1779,13527,11,6707,32448,5,2322
5,21661,48224,12795,15063,4034,28590,11,5863,49738,8,9814,1052,11,3012,47
48,3892,13627,810859,198,84,7213,1209,50,9,5198,125
```
A Simple Example

Let’s start with a simple Ruby program

```
# This is a ruby program
x = 37
y = x + 5
print(y)
print("\n")
```

```ruby
% ruby -w ruby1.rb
42
%
```

Language Basics

- Comments begin with #, go to end of line
- Variables need not be declared
- Line break separates expressions (can also use ";") to be safe
- No special main() function or method

Run Ruby, Run

There are several ways to run a Ruby program

- `ruby -w filename` — execute script in `filename`
  - `--w`: the `-w` will cause Ruby to print a bit more if something bad happens
  - Ruby filenames should end with `.rb` extension
- `irb` — launch interactive Ruby shell
  - Can type in Ruby programs one line at a time, and watch as each line is executed
    ```ruby
    irb(main):001:0> 3+4
    7
    ```
  - Can load Ruby programs via load command
    - Form: `load string`
    - String must be name of file containing Ruby program
    - E.g.: `load 'foo.rb'`
  - Ruby 1.9.3 is installed on linuxlab, Grace clusters

Run Ruby, Run (cont.)

- `fxri` — launch standalone interactive Ruby shell
Run Ruby, Run (cont.)

 Suppose you want to run a Ruby script as if it were an executable (e.g. “double-click”, or as a command)
  • Windows
    ➤ Must associate .rb file extension with ruby command
    ➤ If you installed Ruby using the Windows installer, this was done automatically
    ➤ The Ruby web site has information on how to make this association

Run Ruby, Run (cont.)

 Suppose you want to run a Ruby script as if it were an executable (cont.)
  • *nix (Linux / Unix / etc.)
    ➤ The first line (“shebang”) tells the system where to find the program to interpret this text file
    ➤ Must chmod u+x filename first, or chmod a+x filename so everyone has exec permission
    ➤ Warning: Not very portable: Depends on location of Ruby interpreter
      • /usr/local/bin/ruby vs. /usr/bin/ruby vs. /opt/local/bin/ruby etc.

Creating Ruby Programs

 As with most programming languages, Ruby programs are text files.
  • Note: there are actually different versions of “plain text”! E.g. ASCII, Unicode, Utf-8, etc.
  • You won’t need to worry about this in this course.
  • To create a Ruby program, you can use your favorite text editor, e.g.
    • notepad++ (free, much better than notepad)
    • emacs (free, infinitely configurable)
    • vim
    • Eclipse (see web page for plugin instructions)
    • Many others

Explicit vs. Implicit Declarations

 Java and C/C++ use explicit variable declarations
  • Variables are named and typed before they are used
    ➤ int x, y; x = 37; y = x + 5;

 In Ruby, variables are implicitly declared
  • First use of a variable declares it and determines type
    ➤ x = 37; y = x + 5;
    • x, y exist, will be integers
  • Ruby allows multi-assignment, too
    ➤ x, y = 37, 5; y += x
    • x, y = 37, x+5 would have failed; x was not yet assigned
Tradeoffs?

<table>
<thead>
<tr>
<th>Explicit Declarations</th>
<th>Implicit Declarations</th>
</tr>
</thead>
<tbody>
<tr>
<td>More text to type</td>
<td>Less text to type</td>
</tr>
<tr>
<td>Helps prevent typos</td>
<td>Easy to mistype variable name</td>
</tr>
<tr>
<td>Forces programmer to document types</td>
<td>Variable not held to a fixed type (could imagine variable declarations without types)</td>
</tr>
</tbody>
</table>

Methods in Ruby

```ruby
def sayN(message, n)
  i = 0
  while i < n
    puts message
    i = i + 1
  end
  return i
end
```

```ruby
x = sayN("hello", 3)
puts(x)
```

Method Return Values

- Value of the return is the value of the last executed statement in the method
  - These are the same:
    ```ruby
def add_three(x)
  return x+3
end
def add_three(x)
  x+3
end
```
- Methods can return multiple results (as a list)
  ```ruby
def dup(x)
  return x,x
end
```

Terminology

- Formal parameters
  - Parameters used in the body of the method
    - `def sayN(message, n)` in our example
- Actual parameters
  - Arguments passed in to the method at a call
    - `x = sayN("hello", 3)` in our example
Style

- Names of methods that return a boolean should end in `?`
- Names of methods that modify an object’s state should end in `!`
- Example: suppose `x = [3,1,2]` (this is an array)
  - `x.member?` 3 returns true since 3 is in the array `x`
  - `x.sort` returns a new array that is sorted
  - `x.sort!` modifies `x` in place

Control Statements in Ruby

- A control statement is one that affects which instruction is executed next
  - We’ve seen two so far in Ruby
    - `while` and method call
  - Ruby also has conditionals

```
if grade >= 90 then
  puts "You got an A"
elsif grade >= 80 then
  puts "You got a B"
elsif grade >= 70 then
  puts "You got a C"
else
  puts "You’re not doing so well"
end
```

Ruby Conditionals Must End!

- All Ruby conditional statements must be terminated with the `end` keyword.
- Examples
  - if `grade >= 90` then
    - `puts "You got an A"`
    end
  - if `grade >= 90` then
    - `puts "You got an A"
    else
    - `puts "No A, sorry"
    end

What is True?

- The `guard` of a conditional is the expression that determines which branch is taken

```
if grade >= 90 then
  ...
```
- The `true` branch is taken if the guard evaluates to anything except
  - `false`
  - `nil`
- Warning to C programmers: `0` is not `false`!
Yet More Control Statements in Ruby

- **unless** cond then stmt-f else stmt-t end
  - Same as “if not cond then stmt-t else stmt-f end”

  ```ruby
  unless grade < 90 then
    puts "You got an A"
  else unless grade < 80 then
    puts "You got a B"
  end
  ```

- **until** cond body end
  - Same as “while not cond body end”

  ```ruby
  until i >= n
    puts message
    i = i + 1
  end
  ```

Using If and Unless as Modifiers

- Can write if and unless after an expression
  - puts “You got an A” if grade >= 90
  - puts “You got an A” unless grade < 90

- Why so many control statements?
  - Is this a good idea? Why or why not?
    - **Good**: can make program more readable, expressing programs more directly. In natural language, many ways to say the same thing, which supports brevity and adds style.
    - **Bad**: many ways to do the same thing may lead to confusion and hurt maintainability (if future programmers don’t understand all styles)

Classes and Objects

- Class names begin with an uppercase letter
- The “new” method creates an object
  - s = String.new creates a new String and makes s refer to it
- Every class inherits from **Object**

Everything is an Object

- In Ruby, **everything** is an object
  - (-4).abs
    - integers are instances of **Fixnum**
  - 3 + 4
    - infix notation for “invoke the + method of 3 on argument 4”
  - "programming".length
    - strings are instances of **String**
  - String.new
    - classes are objects with a new method
  - 4.13.class
    - use the class method to get the class for an object
    - floating point numbers are instances of **Float**
Objects and Classes

- Objects are data
- Classes are types (the kind of data which things are)
- But in Ruby, classes themselves are objects!

<table>
<thead>
<tr>
<th>Object</th>
<th>Class (aka type)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Fixnum</td>
</tr>
<tr>
<td>-3.30</td>
<td>Float</td>
</tr>
<tr>
<td>&quot;CMSC 330&quot;</td>
<td>String</td>
</tr>
<tr>
<td>String.new</td>
<td>String</td>
</tr>
<tr>
<td>['a', 'b', 'c']</td>
<td>Array</td>
</tr>
<tr>
<td>Fixnum</td>
<td>Class</td>
</tr>
</tbody>
</table>

- Fixnum, Float, and String are objects of type Class
  - So is Class itself!

Fixing Classes

- Since classes are objects, you can manipulate them however you like
  - Here, the type of y depends on p
    - Either a String or a Time object

```ruby
if p then
  x = String
else
  x = Time
End
y = x.new
```

- You can get names of all the methods of a class
  - `Object.methods`
    - `=> 
      
      - "send",
      - "name",
      - "class_eval",
      - "object_id",
      - "new",
      - "autoload?",
      - "singleton_methods", ... ]

The nil Object

- Ruby uses a special object `nil`
  - All uninitialized fields set to nil (@ prefix used for fields)
    - `irb(main):004:0> @x
      => nil`
  - Like NULL or 0 in C/C++ and null in Java
- `nil` is an object of class `NilClass`
  - It’s a singleton object – there is only one instance of it
    - `NilClass` does not have a new method
  - `nil` has methods like `to_s`, but not other methods
    - `irb(main):006:0> nil + 2
      NoMethodError: undefined method `+' for nil:NilClass`

Two Cool Things to Do with Classes

- What is a Program?

  In C/C++, a program is...
  - A collection of declarations and definitions
  - With a distinguished function definition
    - `int main(int argc, char *argv[]) { ... }`
  - When you run a C/C++ program, it’s like the OS calls `main(...)`

  In Java, a program is...
  - A collection of class definitions
  - With some class (say, `MyClass`) containing a method
    - `public static void main(String[] args)`
  - When you run `java MyClass`, the `main` method of class `MyClass` is invoked
A Ruby Program is...

- The class Object
  - When the class is loaded, any expressions not in method bodies are executed
def sayN(message, n)
  i = 0
  while i < n
    puts message
    i = i + 1
  end
  return i
end

x = sayN("hello", 3)
puts(x)

defines a method of Object
invokes self.sayN
invokes self.puts (part of Object)

Ruby is Dynamically Typed

- Recall we don’t declare types of variables
  - But Ruby does keep track of types at run time

```ruby
NoMethodError: undefined method 'foo' for 3:Fixnum
```

- We say that Ruby is dynamically typed
  - Types are determined and checked at run time

- Compare to C, which is statically typed

```ruby
x = 3; x.foo
```

```c
/* C */
int x;
x = 3;
x = "foo"; /* not allowed */
```

Types in Java and C++

- Are Java and C++ statically or dynamically typed?
  - A little of both
  - Many things are checked statically
    Object x = new Object();
    x.println("hello"); // No such method error at compile time
  - But other things are checked dynamically
    Object o = new Object();
    String s = (String) o; // No compiler warning, fails at run time
    // (Some Java compilers may be smart enough to warn about above cast)

Tradeoffs?

<table>
<thead>
<tr>
<th>Static types</th>
<th>Dynamic types</th>
</tr>
</thead>
<tbody>
<tr>
<td>More work when coding</td>
<td>Less work when coding</td>
</tr>
<tr>
<td>Helps prevent some subtle errors</td>
<td>Can use objects incorrectly and not discover until run time</td>
</tr>
<tr>
<td>Fewer programs type check</td>
<td>More programs type check</td>
</tr>
</tbody>
</table>
Arrays and Hashes

- Ruby data structures are typically constructed from Arrays and Hashes
  - Built-in syntax for both
  - Each has a rich set of standard library methods
  - They are integrated/used by methods of other classes

Standard Library: Array

- Arrays of objects are instances of class *Array*
  - Arrays may be heterogeneous
    
    ```ruby
    a = [1, "foo", 2.14]
    ```
  - C-like syntax for accessing elements, indexed from 0
    
    ```ruby
    x = a[0]; a[1] = 37
    ```
  - Arrays are **growable**
    - Increase in size automatically as you access elements
      
      ```ruby
      irb(main):001:0> b = []; b[0] = 0; b[5] = 0; puts b.inspect
      [0, nil, nil, nil, nil, 0]
      ```
    - `[]` is the empty array, same as `Array.new`

Standard Library: Arrays (cont.)

- Arrays can also shrink
  - Contents shift left when you delete elements
    
    ```ruby
    a = [1, 2, 3, 4, 5]
    a.delete_at(3)  # delete at position 3; a = [1,2,3,5]
    a.delete(2)    # delete element = 2; a = [1,3,5]
    ```
- Can use arrays to model stacks and queues
  
  ```ruby
  a = [1, 2, 3]
  a.push("a")  # a = [1, 2, 3, "a"]
  x = a.pop     # x = "a"
  a.unshift("b")  # a = ["b", 1, 2, 3]
  y = a.shift   # y = "b"
  ```

Iterating Through Arrays

- It's easy to iterate over an array with *while*
  
  ```ruby
  a = [1, 2, 3, 4, 5]
  i = 0
  while i < a.length
      puts a[i]
      i = i + 1
  end
  ```
- Looping through all elements of an array is very common
  - And there's a better way to do it in Ruby
Iteration and Code Blocks

- The Array class also has an each method
  - Takes a code block as an argument

```ruby
a = [1, 2, 3, 4, 5]
a.each { |x| puts x }
```

- We'll consider code blocks generally a bit later

Ranges

- 1..3 is an object of class Range
  - Integers between 1 and 3 inclusively

- 1...3 also has class Range
  - Integers between 1 and 3 but not including 3 itself.

- Not just for integers
  - 'a'..'z' represents the range of letters 'a' to 'z'
  - 1.3...2.7 is the continuous range [1.3, 2.7)

- Discrete ranges offer the each method to iterate
  - And can convert to an array via to_a; e.g., (1..2).to_a

Other Useful Control Statements

```ruby
for elt in [1, "math", 3.4]
puts elt.to_s
end

for i in (1..3)
puts i
end

(1..3).each { |elt| puts elt }

IO.foreach(filename)
{ |x| puts x }
```

More Data-driven Control Statements

Ruby function to print all even numbers from 0 up to (but not including) some given number `x`

```ruby
def even(x)
  x.times { |i|
    if i % 2 == 0
      puts i
    end
  }
end
```
Standard Library: Hash

- A hash acts like an associative array
  - Elements can be indexed by any kind of values
  - Every Ruby object can be used as a hash key, because the Object class has a hash method

- Elements are referred to using [] like array elements, but Hash.new is the Hash constructor

```ruby
italy("population") = 58103033
italy("continent") = "europe"
italy[1861] = "independence"
```

Hash (cont.)

- Hash methods
  - values returns array of a hash’s values (in some order)
  - keys returns an array of a hash’s keys (in some order)

- Iterating over a hash

```ruby
italy.keys.each { |k|    
  print "key: ", k, " value: ", italy[k]
}

italy.each { |k,v|    
  print "key: ", k, " value: ", v
}
```

Convenient syntax for creating literal hashes

- Use { key => value, ... } to create hash table

```ruby
credits = {    
  "cmsc131" => 4,    
  "cmsc330" => 3,
}
x = credits["cmsc330"]  # x now 3
credits["cmsc311"] = 3
```

Defining Your Own Classes

```ruby
class Point    
  def initialize(x, y)    
    @x = x    
    @y = y    
  end    
  def add_x(x)    
    @x += x    
  end    
  def to_s    
    return "(" + @x.to_s + "," + @y.to_s + ")"
  end
end

p = Point.new(3, 4)  # instantiation
p.add_x(4)  # invoking no-arg method
puts(p.to_s)
```
No Access To Internal State

- Instance variables (with @) can be directly accessed only by instance methods.
- Outside class, they require accessors:

<table>
<thead>
<tr>
<th>A typical getter</th>
<th>A typical setter</th>
</tr>
</thead>
<tbody>
<tr>
<td>def x</td>
<td>def x=(value)</td>
</tr>
<tr>
<td>@x</td>
<td>@x = value</td>
</tr>
<tr>
<td>end</td>
<td>end</td>
</tr>
</tbody>
</table>

- Very common, so Ruby provides a shortcut:

```
class ClassWithXandY
  attr_accessor "x", "y"
end
```

No Method Overloading in Ruby

- Thus there can only be one `initialize` method:
  - A typical Java class might have two or more constructors.
  - You can code up your own overloading by using a variable number of arguments, and checking at runtime the number/types of arguments.
- Ruby does issue an exception or warning if a class defines more than one `initialize` method.
  - But last `initialize` method defined is the valid one.

Classes and Objects in Ruby (cont.)

- Recall classes begin with an uppercase letter.
- `inspect` converts any instance to a string:

```
irb(main):033:0> p.inspect
=> "#<Point:0x54574 @y=4, @x=7>"
```
- The `to_s` method can be invoked implicitly:
  - Could have written `puts(p)`
    - Like Java’s `toString()` methods

Inheritance

- Recall that every class inherits from `Object`:

```
class A < Object
  def add(x)
    return x + 1
  end
end

class B < A
  def add(y)
    return (super(y) + 1)
  end
end
```

```
b = B.new
puts(b.add(3))
```
super() in Ruby

- Within the body of a method
  - Call to super() acts just like a call to that original method
  - Except that search for method body starts in the superclass of the object that was found to contain the original method

Mixins

- Another form of code reuse is “mix-in” inclusion
  - include A “inlines” A’s methods at that point
    - Referred-to variables/methods captured from context
    - In effect: it adds those methods to the current class

Global Variables in Ruby

- Ruby has two kinds of global variables
  - Class variables beginning with @@ (static in Java)
  - Global variables across classes beginning with $

Special Global Variables

- Ruby has a special set of global variables that are implicitly set by methods
- The most insidious one: $_
  - Last line of input read by gets or readline
- Example program

```ruby
gets    # implicitly reads input line into $_
p.print # implicitly prints out $_
```

- Using $_ leads to shorter programs
  - And confusion
  - We suggest you avoid using it
Creating Strings in Ruby

- Substitution in double-quoted strings with `#{ }`
  - `course = "330"`; `msg = "Welcome to #{course}"`
  - "It is now #{Time.now}"
  - The contents of `#{ }` may be an arbitrary expression
  - Can also use single-quote as delimiter
  - No expression substitution, fewer escaping characters

- Here-documents
  - `s = <<END
    This is a text message on multiple lines
    and typing \n is annoying
    END`

Creating Strings in Ruby (cont.)

- Ruby also has `printf` and `sprintf`
  - `printf("Hello, %s\n", name);`
  - `sprintf("%d: %s", count, Time.now)`
    - Returns a string

- The `to_s` method returns a `String` representation of a class object

Standard Library: String

- The `String` class has many useful methods
  - `s.length`  # length of string
  - `s1 == s2`  # structural equality (string contents)
  - `s = "A line
  s.chomp`  # returns "A line"
    - Return new string with s's contents except newline at end of line removed
  - `s = "A line
  s.chomp!`
    - Destructively removes newline from s

- Consider these three examples again
  - All involve searching in a string for a certain pattern
  - What if we want to find more complicated patterns?
    - Find first occurrence of "a" or "b"
Object Copy vs. Reference Copy

Consider the following code

- Assume an object/reference model like Java or Ruby
  - Or even two pointers pointing to the same structure
  ```
  x = "groundhog" ; y = x
  ```

Which of these occur?

- Object copy
- Reference copy

Physical vs. Structural Equality

Consider these cases again:

- If we compare `x` and `y`, what is compared?
  - The references, or the contents of the objects they point to?
- If references are compared (physical equality) the first would return false but the second true
- If objects are compared both would return true

String Equality

- In Java, `x == y` is physical equality, always
  - Compares references, not string contents
- In Ruby, `x == y` for strings uses structural equality
  - Compares contents, not references
  - `==` is a method that can be overridden in Ruby!
  - To check physical equality, use the `equal?` method
    - Inherited from the `Object` class
- It’s always important to know whether you’re doing a reference or object copy
  - And physical or structural comparison
### Comparing Equality

<table>
<thead>
<tr>
<th>Language</th>
<th>Physical equality</th>
<th>Structural equality</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java</td>
<td>a == b</td>
<td>a.equals(b)</td>
</tr>
<tr>
<td>C</td>
<td>a == b</td>
<td>*a == *b</td>
</tr>
<tr>
<td>Ruby</td>
<td>a.equal?(b)</td>
<td>a == b</td>
</tr>
<tr>
<td>Ocaml</td>
<td>a == b</td>
<td>a = b</td>
</tr>
<tr>
<td>Python</td>
<td>a is b</td>
<td>a == b</td>
</tr>
<tr>
<td>Scheme</td>
<td>(eq? a b)</td>
<td>(equal? a b)</td>
</tr>
<tr>
<td>Visual Basic .NET</td>
<td>a ls b</td>
<td>a = b</td>
</tr>
</tbody>
</table>

### Summary

- **Scripting languages**
- **Ruby language**
  - Implicit variable declarations
  - Dynamic typing
  - Many control statements
  - Classes & objects
  - Strings