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
```bash
% wc *
  271 674 5323 AST.c
 100 492 3219 AST.h
 159 462 4259 AST_defs.h
 1874 5428 47461 AST_defs.c
 1375 6307 53667 AST_defs.h
 371 884 9483 AST_parent.c
 810 2328 24589 AST_print.c
 640 3070 33530 AST_types.h
 285 846 7081 AST_utils.c
  59 274 2154 AST_utils.h
  50 400 28756 AST_utils.o
 866 2757 25873 Makefile
 270 725 5378 Makefile.am
 866 2743 27320 Makefile.in
  38 175 1154 alloca.c
 2035 4516 26500 alloctypes.c
 38 350 3286 alloctypes.h
 104 1051 66848 alloctypes.o
...
```
Climate Data for IAD in August, 2005

| AVG | MAX | MIN | Dy MAX | Dy MIN | AVG DEP | HDD | CDD | WTR | SNW | DPTH | SPD | SPD DIR | MIN PSBL | S-S WX | SPD DR |
|-----|-----|-----|--------|--------|---------|------|-----|-----|-----|-----|------|-----|---------|-----------|-------|-------|
| 1  | 87  | 66  | 77     | 1      | 0.00    | 0.0  | 0   | 2.5 | 9   | 200  | M    | M    | 7     | 18    | 12    |
| 2  | 92  | 67  | 80     | 4      | 0      | 15   | 0.0 | 0   | 3.5 | 10   | 10   | M    | M    | 3     | 18    | 32    |
| 3  | 93  | 69  | 81     | 5      | 0      | 16   | 0.0 | 0   | 4.1 | 13   | 360  | M    | M    | 2     | 18    | 360   |
| 4  | 95  | 69  | 82     | 6      | 0      | 17   | 0.0 | 0   | 3.6 | 9    | 310  | M    | M    | 3     | 18    | 290   |
| 5  | 94  | 73  | 84     | 8      | 0      | 19   | 0.0 | 0   | 5.9 | 18   | 10   | M    | M    | 3     | 18    | 25    |
| 6  | 89  | 70  | 80     | 4      | 0      | 15   | 0.02| 0   | 5.3 | 20   | 200  | M    | M    | 6     | 138   | 23    |
| 7  | 89  | 69  | 79     | 3      | 0      | 14   | 0.0 | 0   | 3.6 | 14   | 200  | M    | M    | 7     | 1     | 32    |
| 8  | 86  | 70  | 78     | 3      | 0      | 13   | 0.74| 0   | 4.4 | 17   | 150  | M    | M    | 10    | 18    | 23    |
| 9  | 76  | 70  | 73     | -2     | 0      | 8    | 0.19| 0   | 4.1 | 9    | 90   | M    | M    | 9     | 1     | 16    |
| 10 | 87  | 71  | 79     | 4      | 0      | 14   | 0.0 | 0   | 2.3 | 8    | 240  | M    | M    | 8     | 1     | 10    |

Raw Census 2000 Data for DC

```
10 U.S. DC
500.0, 0.000505, 570508, 722294, 570509.12, 6.670509, 570509, 6.0, 6.57
5058, 175056, 1435931, 2066, 14762, 383, 151761, 14661, 570509, 148677, 3605
10, 1540, 14050, 231638, 1072, 4651, 14495, 5532, 46, 157, 20950, 5099, 572
955, 246977, 2502, 4170, 3516, 3938, 3727, 2473, 3924, 325, 3155, 3048, 3048,
2598, 2473, 2769, 2761, 2823, 2753, 6056, 5501.5377, 4569, 13555, 2496, 3214, 237
26, 20573, 18992, 14335, 12558, 4365, 1465, 4459, 1165, 379, 3240, 2347, 3032
32, 3338, 3557, 2957, 3629, 3326, 3864, 2657, 2874, 3252, 3536, 3247, 2899, 283
9, 2824, 2824, 2807, 2871, 2461, 6586, 5652, 1505, 27177, 27475, 28777, 283196,
20858, 19137, 152065, 5678, 6257, 10714, 9477, 6957, 572059, 5363
73, 370475, 115993, 5563, 6038, 79489, 12840, 122518, 3548, 2347, 6445, 9474, 4
628, 13410, 1416446, 61094, 47944, 135350, 711576, 40975, 10703, 35971, 354686,
7073, 32113, 248560, 106869, 47944, 40975, 140021, 115993, 5563, 6038, 20464, 53965,
433, 13355, 6045, 4246, 47558, 22329, 20466, 15355, 7013, 207086, 67077, 371
12, 21742, 122677, 4757, 2573, 32714, 2704, 2947, 7498, 7268, 18165, 18172,
1913, 12331, 4351, 36151, 7482390, 199656, 90221, 46274, 21434, 24803, 67497, 670
1, 3979, 4726, 29042, 77515, 18067, 102535, 59068, 22007, 46814, 21762, 15776, 211,
1813, 12574, 2089, 20631, 14117, 18778, 11256, 48878, 28464, 182058, 103522, 4784, 389
2, 132627, 109569, 190473, 271209, 509, 218129
```

A Simple Example

Let’s start with a simple Ruby program

```
ruby1.rb:
#
# This is a ruby program
x = 37
y = x + 5

print(y)
print("\n")
```

Language Basics

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

Run Ruby, Run

There are three ways to run a Ruby program

- `ruby -w filename` – execute script in `filename`

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

Suppose you want to run a Ruby script as if it were an executable

```
#!/usr/local/bin/ruby -w
print("Hello, world!\n")
```

Run Ruby, Run (cont.)

```
% ./ruby1.rb
Hello, world!
```
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

Tradeoffs?

<table>
<thead>
<tr>
<th>Explicit Declarations</th>
<th>Implicit Declarations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Higher overhead</td>
<td>Lower overhead</td>
</tr>
<tr>
<td>Helps prevent typos</td>
<td>Easy to mistype variable names</td>
</tr>
<tr>
<td>Forces programmer to document types</td>
<td>Figures out types of variables automatically</td>
</tr>
</tbody>
</table>

Methods in Ruby

Methods are declared with `def...end`

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

(x) = sayHi("hello", 3)

puts(x)

Methods should begin with lowercase letter and be defined before they are called.

Method (and Function) Terminology

- Formal parameters
  - Parameters used in the body of the method
    - `message`, `n` in our example
- Actual parameters
  - Arguments passed in to the method at a call
    - "hello", 3 in our example

More 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 function call
- Ruby also has conditionals

```ruby
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
  if grade >= 60 then
    puts "You’re not doing so well"
  end
end
```

What is True?

- The guard of a conditional is the expression that determines which branch is taken
  - `if grade >= 90 then` ...
  - Guard
- 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-f else stmt-t end”

```
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”

```
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?
  - Advantages? Disadvantages?

Other Useful Control Statements

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

```
while i>n
  break
  next
  puts message
end
```

```
{1..3}.each {
  |elt|
  puts elt
}
```

```
IO.foreach(filename)
  |
  puts x
end
```

```
case x
  when 1, 3..5
    puts message
  when 2, 6..8
    puts message
end
```

Using Ruby Control Statements

Ruby function to print all even numbers from 1 to some given value x

```
def even(x)
  for i in (1..x)
    if i % 2 == 0
      puts i
    end
  end
end
```

```
def even(x)
  (1..x).each
    |i|
    if i % 2 == 0
      puts i
    end
end
```

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 in fact 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!

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixnum</td>
<td>Float</td>
<td>String</td>
</tr>
<tr>
<td>-3.30</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;CMSC 330&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>String</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fixnum</td>
<td></td>
<td></td>
</tr>
<tr>
<td>String</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Fixnum, Float, String, etc., (including Class), are objects of type Class

**Two Cool Things to Do with Classes**
- Since classes are objects, you can manipulate them however you like
  - if p then x = String else x = Time end
    # Time is another class
  - y = x.new
    # creates a String or a Time, depending upon p

- 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 (@ refers to a class field)
    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 that don’t make sense
    irb(main):006:0> @x + 2
    NoMethodError: undefined method `+' for nil:NilClass

**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 a class MyClass that contains 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)

**Ruby is Dynamically Typed**
- Recall we don’t declare types of variables
  - But Ruby does keep track of types at run time
    x = 3; x.foo
    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
# Ruby
x = 3
x = "foo" # gives x a new type

/# C +/
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
  ```java
  Object x = new Object();
  x.println("hello"); // No such method error at compile time
  ```
- But other things are checked dynamically
  ```java
  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 to do when writing code</td>
<td>Less work when writing code</td>
</tr>
<tr>
<td>Helps prevent some subtle errors</td>
<td>Can use objects incorrectly and not realize until execution</td>
</tr>
<tr>
<td>Fewer programs type check</td>
<td>More programs type check</td>
</tr>
</tbody>
</table>

Classes and Objects in Ruby

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

```ruby
p = Point.new(3, 4)
p.addX(4)
puts(p.to_s)
```

Notes For Java Programmers

- Ruby does not support method overloading
  - A typical Java class might have two or more constructors
  - Since Ruby does not support method overloading there can only be one initialize method
- Ruby does issue an exception or warning if classes defines more than one initialize method
  - But last initialize method defined is the valid one

Classes and Objects in Ruby (cont'd)

- Recall classes begin with an uppercase letter
- `inspect` converts any instance to a string
  ```ruby
  #/main/033/0> p.inspect
  => "#<Point:0x954574 @x=4, @y=7>"
  ```
- Instance variables are prefixed with `@`
  - Compare to local variables with no prefix
  - Cannot be accessed outside of class
- 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`
  ```ruby
  class A
    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

Global Variables in Ruby

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

```ruby
class Global
  @@x = 0
  def Global.inc
    @@x = @@x + 1; $x = $x + 1
  end
  def Global.get
    return @@x
  end
end

$x = 0
Global.inc
$x = $x + 1
Global.inc
puts(Global.get)
puts($x)
```

define a class ("singleton") method

Special Global Variables

- Ruby has a bunch of global variables that are implicitly set by methods
- The most insidious one: $_
  - Default method return, argument in many cases
- Example
  ```ruby
  gets # implicitly reads input into $_
  print # implicitly writes $_
  ```
- Using $_ leads to shorter programs
  - And confusion
  - It’s suggested 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
- Here-documents
  ```ruby
  $ = <<END
  This is a text message on multiple lines
  and typing \n is annoying
  END
  ```

Substitution in Ruby Strings

- Writing elt as #{elt} makes it clear that it is a variable to be evaluated, not a literal word to be printed. This is a cleaner way to express output; it builds a single string and presents it as a single argument to puts.

```ruby
ruby> for elt in [100,-9.6,“pickle”]
|    puts "#{elt} #{elt.class}"
| end
100 (Fixnum)
-9.6 (Float)
pickle (String)
```

Creating Strings in Ruby (cont.)

- Ruby also has printf and sprintf
  ```ruby
  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\n"; s.chomp`  # return a new string with `s`'s contents except newline at end of line removed
  - `s = "A line\n"; s.chomp!`  # Destructively removes newline from `s`
  - `"r1\tr2\tr3".each("\t") { |rec| puts rec }`  # Apply code block to each tab-separated substring

**Standard Library: String (cont.)**

- `"hello".index("l", 0)`  # Return index of the first occurrence of string in `s`, starting at `n`
- `"hello".sub("h", "j")`  # Replace first occurrence of "h" by "j" in `string`
- `"r1\tr2\tr3".split("\t")`  # Return array of substrings delimited by tab

**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
  ```ruby
  x = "groundhog" ; y = x
  ```

**Object Copy vs. Reference Copy (cont.)**

- For
  ```ruby
  x = "groundhog" ; y = x
  ```
  - Ruby and Java would both do a reference copy
- But for
  ```ruby
  x = "groundhog"
  y = String.new(x)
  ```
  - Ruby would cause an object copy
  - Unnecessary in Java since Strings are `final`

**Physical vs. Structural Equality**

- Consider these cases again:
  ```ruby
  x = "groundhog" ; y = "groundhog"
  ```

**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><code>a == b</code></td>
<td><code>a.equals(b)</code></td>
</tr>
<tr>
<td>C</td>
<td><code>a == b</code></td>
<td><code>*a == *b</code></td>
</tr>
<tr>
<td>Ruby</td>
<td><code>a.equal?(b)</code></td>
<td><code>a == b</code></td>
</tr>
<tr>
<td>Ocaml</td>
<td><code>a == b</code></td>
<td><code>a ~ b</code></td>
</tr>
<tr>
<td>Python</td>
<td><code>a is b</code></td>
<td><code>a == b</code></td>
</tr>
<tr>
<td>Scheme</td>
<td><code>(eq? a b)</code></td>
<td><code>(equal? a b)</code></td>
</tr>
<tr>
<td>Visual Basic .NET</td>
<td><code>a ls b</code></td>
<td><code>a = b</code></td>
</tr>
</tbody>
</table>

### Summary

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