

# CMSC 330: Organization of Programming Languages

## Introduction to Ruby

### Introduction

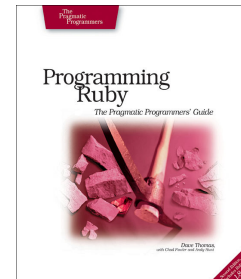
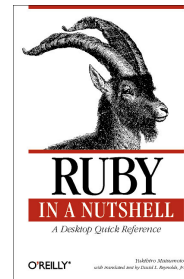
- ▶ Ruby is an *object-oriented, imperative scripting language*
  - "I wanted a scripting language that was more powerful than Perl, and more object-oriented than Python. That's why I decided to design my own language."
  - "I believe people want to express themselves when they program. They don't want 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")

### Last lecture

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

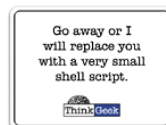
### Books on Ruby



- 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 238788 AST.o
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 5578 Makefile.am
866 2743 27320 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

1	2	3	4	5	6A	6B	7	8	9	10	11	12	13	14	15	16	17	18
=====																		
DY MAX MIN AVG DEP HDD CDD WTR SNW DPTH SPD SPD DIR MIN PSBL S-S WX																		
=====																		
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	2	18	17	360
4	95	69	82	6	0	17	0.00	0.0	0	3.6	9	310	M	M	3	18	12	290
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	0.02	0.0	0	5.3	20	200	M	M	6	138	23	210
7	89	69	79	3	0	14	0.00	0.0	0	3.6	14	200	M	M	7	1	16	210
8	86	70	78	3	0	13	0.74	0.0	0	4.4	17	150	M	M	10	18	23	150
9	76	70	73	-2	0	8	0.19	0.0	0	4.1	9	90	M	M	9	18	13	90
10	87	71	79	4	0	14	0.00	0.0	0	2.3	8	260	M	M	8	1	10	210
...																		

CMSC 330

7

## 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, 527044, 15861
7, 340061, 1560, 14605, 291, 1638, 10272, 45015, 16689, 3152, 446, 157, 92, 20090, 43
89, 572059, 268827, 3362, 3048, 3170, 3241, 3504, 3286, 3270, 3475, 3939, 3647, 3525
, 3044, 2928, 2913, 2769, 2752, 2933, 2703, 4056, 5501, 5217, 4969, 13555, 24995, 242
16, 23726, 20721, 18802, 16523, 12318, 4345, 5810, 3423, 4690, 7105, 5739, 3260, 234
7, 303232, 3329, 3057, 2935, 3429, 3326, 3456, 3257, 3754, 3192, 3523, 3336, 3276, 29
89, 2838, 2824, 2624, 2807, 2871, 4941, 6588, 5625, 5563, 17177, 27475, 24377, 22818
, 21319, 20851, 19117, 15260, 5066, 6708, 4257, 6117, 10741, 9427, 6807, 6175, 57205
9, 536373, 370675, 115963, 55603, 60360, 57949, 129440, 122518, 3754, 3168, 22448,
9967, 4638, 14110, 16160, 165698, 61049, 47694, 13355, 71578, 60875, 10703, 33071,
35686, 7573, 28113, 248590, 108569, 47694, 60875, 140021, 115963, 58050, 21654, 36
396, 57913, 10355, 4065, 6290, 47558, 25229, 22329, 24058, 13355, 10703, 70088, 657
37, 37112, 21742, 12267, 9475, 9723, 2573, 2314, 760, 28625, 8207, 7469, 738, 19185,
18172, 1013, 1233, 4351, 3610, 741, 248590, 199456, 94221, 46274, 21443, 24831, 479
47, 8705, 3979, 4726, 39242, 25175, 14067, 105235, 82928, 22307, 49134, 21742, 1177
6, 211, 11565, 9966, 1650, 86, 1564, 8316, 54, 8262, 27392, 25641, 1751, 248590, 1159
63, 4999, 22466, 26165, 24062, 16529, 12409, 7594, 1739, 132627, 11670, 32445, 2322
5, 21661, 16234, 12795, 10563, 4034, 248590, 115963, 48738, 28914, 19259, 10312, 47
48, 3992, 132627, 108569, 19284, 2713, 1209, 509, 218, 125
...
```

CMSC 330

8

## 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")
```

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

CMSC 330

9

## 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)

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

CMSC 330

10

## Run Ruby, Run

- There are three ways to run a Ruby program
  - `ruby -w filename` – execute script in *filename*
    - tip: the `-w` will cause Ruby to print a bit more if something bad happens
  - `irb` – launch interactive Ruby shell
    - can type in Ruby programs one line at a time, and watch as each line is executed
 

```
irb(main):001:0> 3+4
=> 7
irb(main):002:0> print("hello\n")
hello
=> nil
```

CMSC 330

11

## Run Ruby, Run (cont.)

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

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

`./filename` # run program

- 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 `/usr/local/bin/ruby`

CMSC 330

12

## 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

CMSC 330

13

## Tradeoffs?

<u>Explicit Declarations</u>	<u>Implicit Declarations</u>
Higher overhead	Lower overhead
Helps prevent typos	Easy to mistype variable name
Forces programmer to document types	Figures out types of variables automatically

CMSC 330

14

## Methods in Ruby

Methods are declared with `def...end` List parameters at definition

```
def sayN(message, n)
  i = 0
  while i < n
    puts message
    i = i + 1
  end
  return i
end
x = sayN("hello", 3)
puts(x)
```

May omit parens on call

Invoke method

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

CMSC 330

15

## 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

CMSC 330

16

## 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

```
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
```

CMSC 330

17

## 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!

CMSC 330

18

## 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`”

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

- ▶ `until` `cond` `body` `end`
  - Same as “`while not cond body end`”

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

CMSC 330

19

## 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?

CMSC 330

20

## Other Useful Control Statements

The diagram illustrates several Ruby control statements:

- `for elt in [1, "math", 3.4]` `puts elt.to_s` `end`
- `for i in (1..3)` `puts i` `end` (Callout: generates a string (cf. `to_i`))
- `(1..3).each { |elt| puts elt }`
- `IO.foreach(filename) { |x| puts x }` (Callout: code block)
- `while i > n` `break` `next` `puts message` `redo` `end` (Callout: does not need 'break')
- `case x` `when 1, 3..5` `when 2, 6..8` `end`

CMSC 330

## 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
```

CMSC 330

22

## 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`

CMSC 330

23

## 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`

CMSC 330

24

## Objects and Classes

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

Object	Class
10	Fixnum
-3.30	Float
"CMSC 330"	String
String.new	String
Fixnum	Class
String	Class

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

CMSC 330

25

## 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
  - ... # creates a String or a Time, # depending upon p
  - 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", ...]

CMSC 330

26

## 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
```

CMSC 330

27

## 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

CMSC 330

28

## A Ruby Program is...

- The class Object
  - When the class is loaded, any expressions not in method bodies are executed

defines a method of Object

invokes self.sayN

invokes self.puts  
(part of Object)

```
def sayN(message, n)
  i = 0
  while i < n
    puts message
    i = i + 1
  end
  return i
end
x = sayN("hello", 3)
puts(x)
```

CMSC 330

29

## 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
x = 3
x = "foo" # gives x a
          # new type
```

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

CMSC 330

30

## 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)

CMSC 330

31

## Tradeoffs?

### Static types

More work to do when writing code  
 Helps prevent some subtle errors  
 Fewer programs type check

### Dynamic types

Less work when writing code  
 Can use objects incorrectly and not realize until execution  
 More programs type check

CMSC 330

32

## Classes and Objects in Ruby

```

class Point
  def initialize(x, y)
    @x = x
    @y = y
  end

  def addX(x)
    @x += x
  end

  def to_s
    return "(" + @x.to_s + ", " + @y.to_s + ")"
  end
end

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

← class contains method/constructor definitions  
← constructor definition  
← instance variables prefixed with "@"  
← method with no arguments  
← instantiation  
← invoking no-arg method

CMSC 330

33

## 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

CMSC 330

34

## Classes and Objects in Ruby (cont'd)

- 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>"`
- 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

CMSC 330

35

## Inheritance

- Recall that every class inherits from `Object`

```

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))
    
```

← extend superclass  
← invoke add method of parent

CMSC 330

36

## 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

CMSC 330

37

## Global Variables in Ruby

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

```
class Global
  @@x = 0

  def Global.inc
    @@x = @@x + 1; $x = $x + 1
  end

  def Global.get
    return @@x
  end
end
```

define a class ("singleton") method

CMSC 330

38

## 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

```
gets # implicitly reads input into $_
print # implicitly writes $_
```

- ▶ Using \$\_
  - And confusion
  - It's suggested you avoid using it

CMSC 330

39

## Creating Strings in Ruby

- ▶ Substitution in double-quoted strings with #{ }
  - course = "330"; msg = "Welcome to #{course}"
  - "It is now #{Time.new}"
  - 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
```

CMSC 330

40

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

CMSC 330

41

## 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

CMSC 330

42

## 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` # returns "A line"
    - > Return new string with s's contents except newline at end of line removed
  - `s = "A line\n"; s.chomp!`
    - > Destructively removes newline from `s`
    - > *Convention:* methods ending in ! modify the object
    - > *Another convention:* methods ending in ? observe the object
  - `"r1\r2\t\r4".each("\t") { |rec| puts rec }`
    - > Apply code block to each tab-separated substring

CMSC 330

43

## 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
  - > Use `gsub` ("global" sub) to replace all occurrences
- `"r1\r2\t\r3".split("\t")`
  - > Return array of substrings delimited by tab
- ▶ 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"
    - > Split string at tabs, spaces, and newlines

Regular Expressions!

CMSC 330

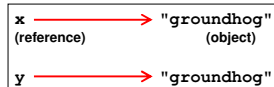
44

## Object Copy vs. Reference Copy

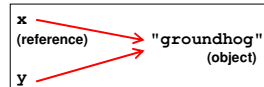
- ▶ Consider the following code
  - Assume an object/reference model like Java or Ruby
    - > Or even two pointers pointing to the same structure

```
x = "groundhog" ; y = x
```

- ▶ Which of these occur?



Object copy



Reference copy

CMSC 330

45

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

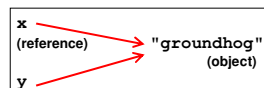
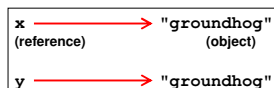
- ▶ For `x = "groundhog" ; y = x`
  - Ruby and Java would both do a reference copy
- ▶ But for `x = "groundhog" ; y = String.new(x)`
  - Ruby would cause an object copy
  - Unnecessary in Java since Strings are **final**

CMSC 330

46

## 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

CMSC 330

47

## 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

CMSC 330

48

## Comparing Equality

---

Language	Physical equality	Structural equality
<a href="#">Java</a>	<code>a == b</code>	<code>a.equals(b)</code>
<a href="#">C</a>	<code>a == b</code>	<code>*a == *b</code>
<a href="#">Ruby</a>	<code>a.equal?(b)</code>	<code>a == b</code>
<a href="#">Ocaml</a>	<code>a == b</code>	<code>a = b</code>
<a href="#">Python</a>	<code>a is b</code>	<code>a == b</code>
<a href="#">Scheme</a>	<code>(eq? a b)</code>	<code>(equal? a b)</code>
<a href="#">Visual Basic .NET</a>	<code>a Is b</code>	<code>a = b</code>

CMSC 330

49

## Summary

---

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

CMSC 330

50