CMSC 330: Organization of Programming Languages

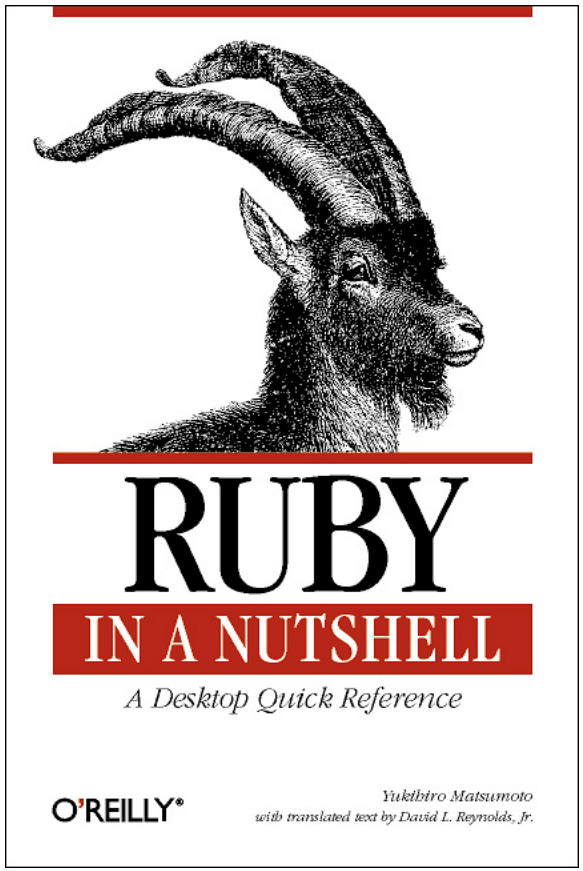
Introduction to Ruby

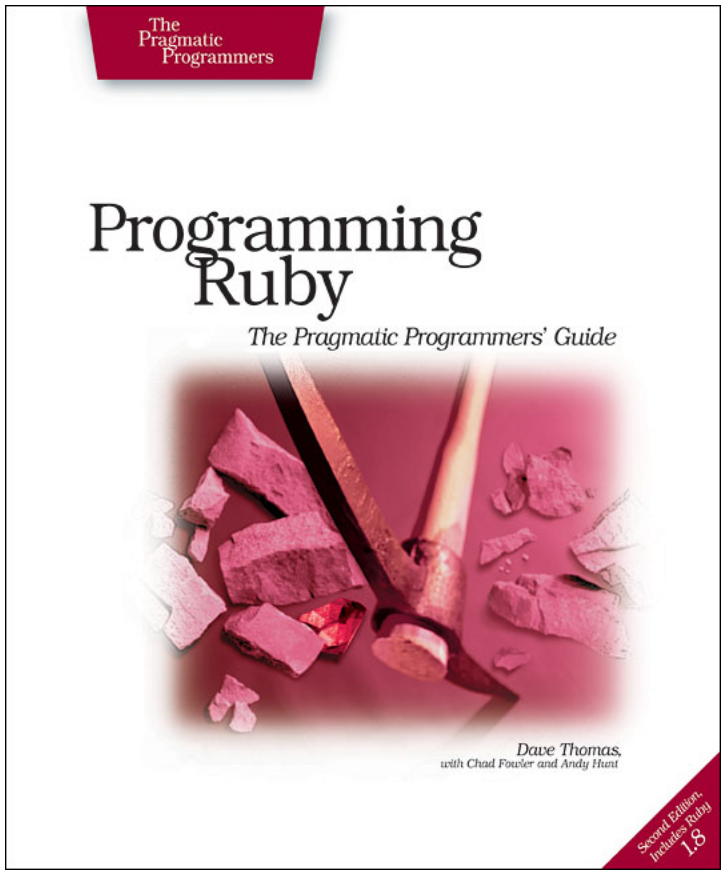
Introduction

- 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")*

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
										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	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

...

Raw Census 2000 Data for DC

```
u108_S,DC,
     000,01,0000001,572059,72264,572059,12.6,572059,572059,572059,0,0,0,0,57
     2059,175306,343213,2006,14762,383,21728,14661,572059,527044,158617,3400
     61,1560,14605,291,1638,10272,45015,16689,3152,446,157,92,20090,4389,572
     059,268827,3362,3048,3170,3241,3504,3286,3270,3475,3939,3647,3525,3044,
     2928,2913,2769,2752,2933,2703,4056,5501,5217,4969,13555,24995,24216,237
     26,20721,18802,16523,12318,4345,5810,3423,4690,7105,5739,3260,2347,3032
     32,3329,3057,2935,3429,3326,3456,3257,3754,3192,3523,3336,3276,2989,283
     8,2824,2624,2807,2871,4941,6588,5625,5563,17177,27475,24377,22818,21319
     ,20851,19117,15260,5066,6708,4257,6117,10741,9427,6807,6175,572059,5363
     73,370675,115963,55603,60360,57949,129440,122518,3754,3168,22448,9967,4
     638,14110,16160,165698,61049,47694,13355,71578,60875,10703,33071,35686,
     7573,28113,248590,108569,47694,60875,140021,115963,58050,21654,36396,57
     913,10355,4065,6290,47558,25229,22329,24058,13355,10703,70088,65737,371
     12,21742,12267,9475,9723,2573,2314,760,28625,8207,7469,738,19185,18172,
     1013,1233,4351,3610,741,248590,199456,94221,46274,21443,24831,47947,870
     5,3979,4726,39242,25175,14067,105235,82928,22307,49134,21742,11776,211,
     11565,9966,1650,86,1564,8316,54,8262,27392,25641,1751,248590,115963,499
     9,22466,26165,24062,16529,12409,7594,1739,132627,11670,32445,23225,2166
     1,16234,12795,10563,4034,248590,115963,48738,28914,19259,10312,4748,399
     2,132627,108569,19284,2713,1209,509,218,125
...
```

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
%
```

Language Basics

comments begin with #, go to end of line

variables need not be declared

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

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

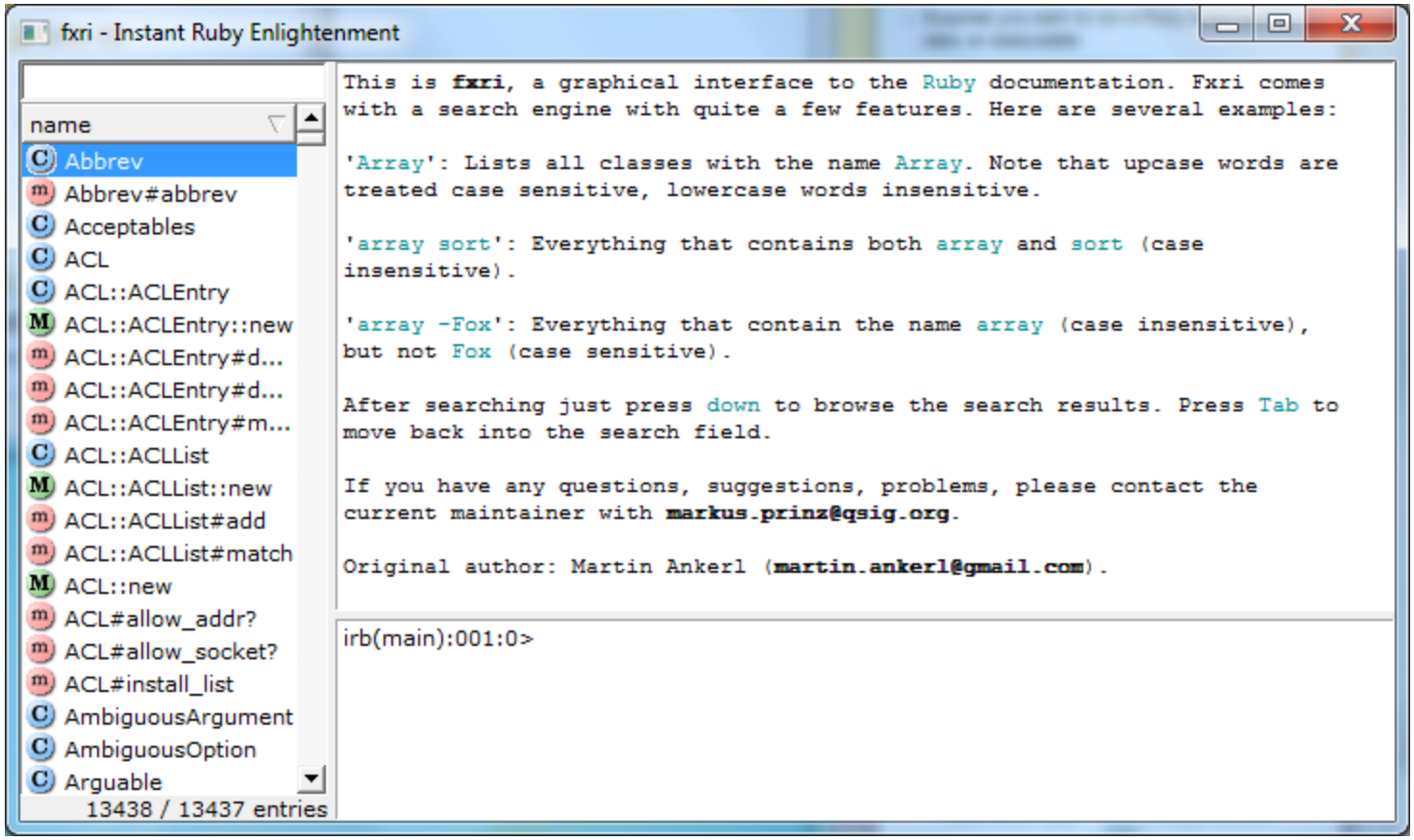
- ruby -w *filename* – execute script in *filename*
  - tip: the -w will cause Ruby to print a bit more if something bad happens
  - Ruby filenames should end with '.rb' extension
- irb – launch interactive Ruby shell inside command window
  - Can type in Ruby programs one line at a time, and watch as each line is executed

    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'

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.)

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

    - 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

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, EBCDIC.
  - 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
  - notepad++ (free, much better than notepad)
  - emacs (free, infinitely configurable)
  - vim
  - Many others

Explicit vs. Implicit Declarations

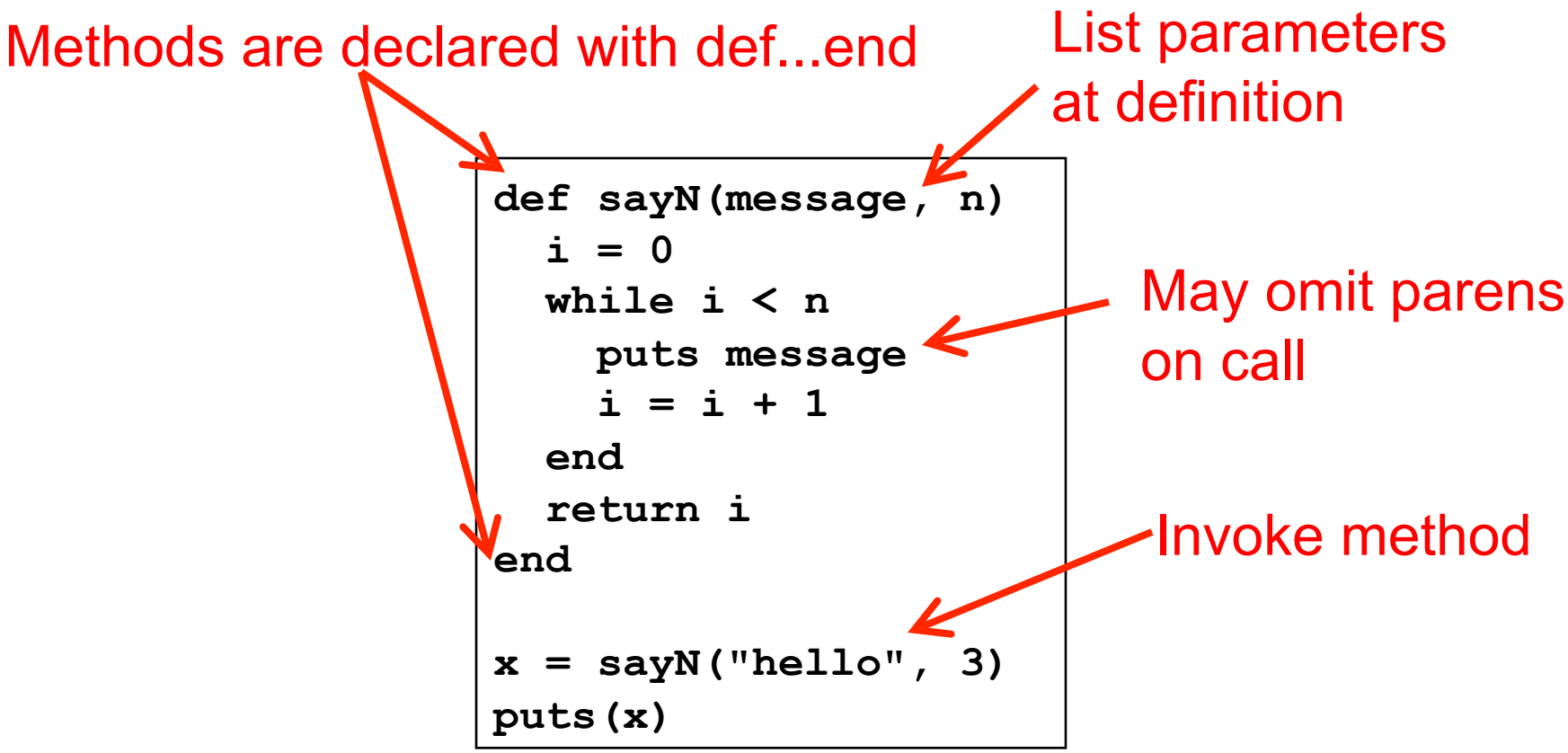
- Java and C/C++ use explicit variable declarations
  - Variables are named and typed before they are used
    - int x, y; x = 37; y = x + 5;

- 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

Tradeoffs?

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

Methods in Ruby



(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

```
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

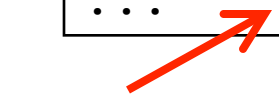
```
if grade >= 90 then
  puts "You got an A"
end
```

```
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
...
```



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-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
```

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
```

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

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

generates a string (cf. to_i)

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

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

code block

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

does not need 'break'

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!

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

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 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

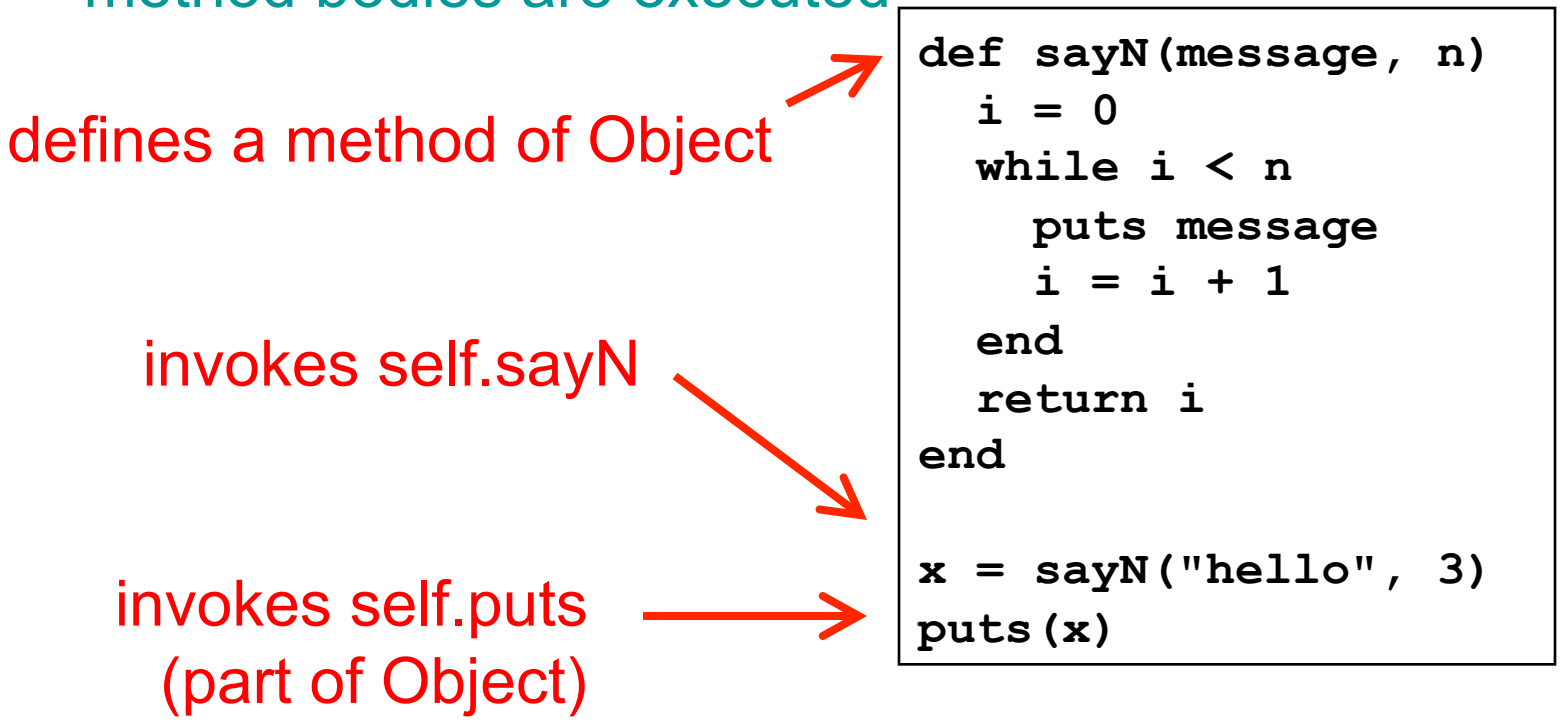
A Ruby Program is...

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

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)
```

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 */
```

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
```

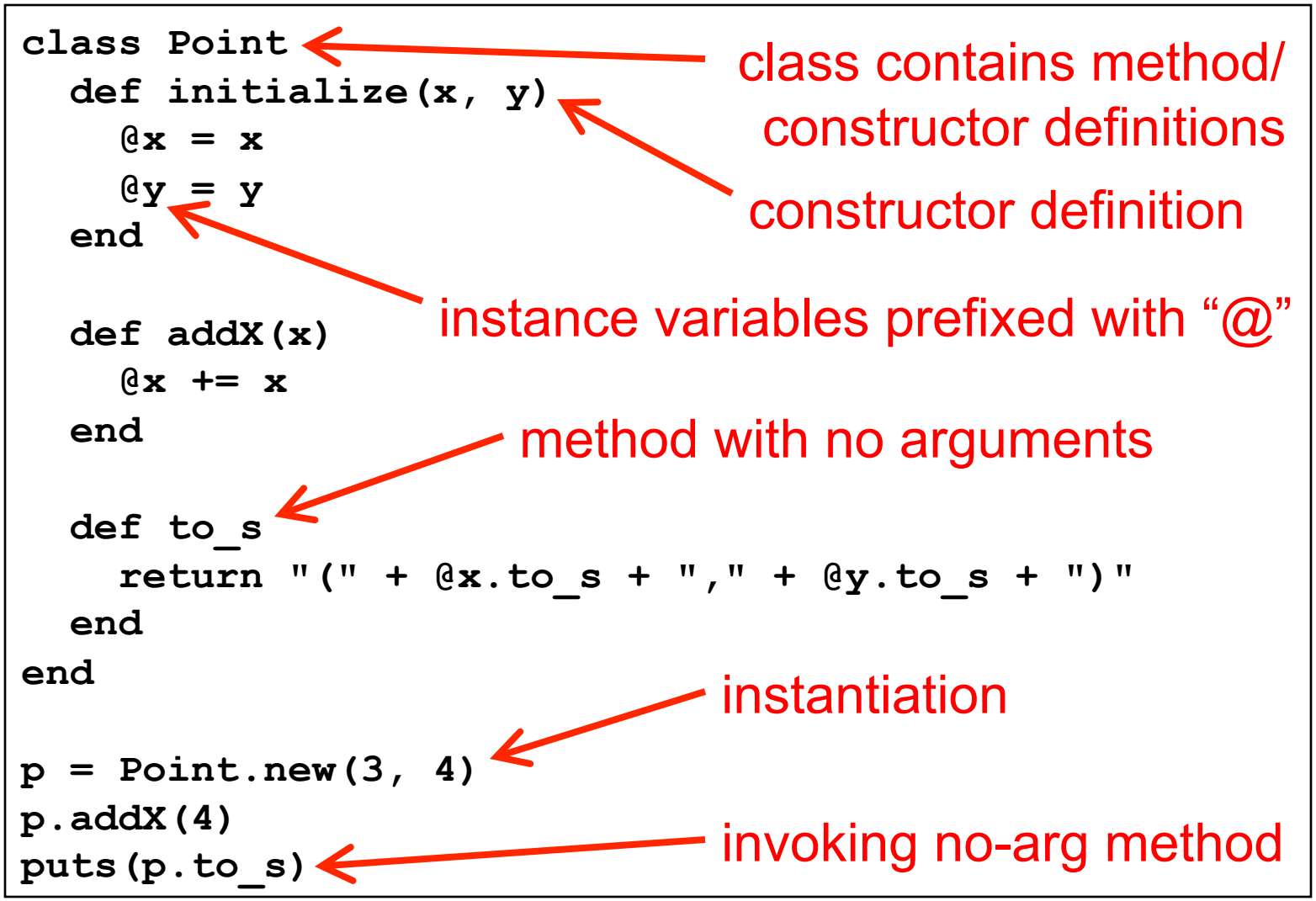
  - But other things are checked dynamically

```
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?

Static types	Dynamic types
More work to do when writing code	Less work when writing code
Helps prevent some subtle errors	Can use objects incorrectly and not realize until execution
Fewer programs type check	More programs type check

Classes and Objects in Ruby



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

```
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

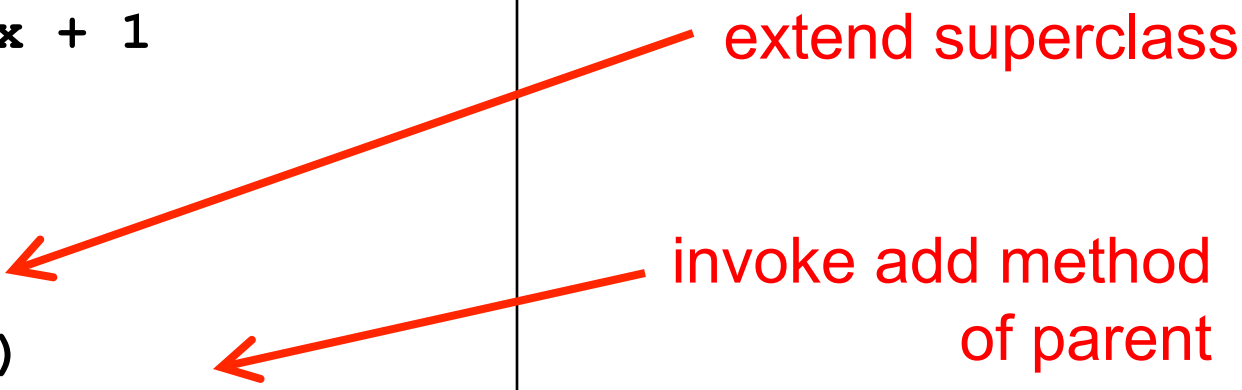
Inheritance

- 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))
```



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 $

```
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 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

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

- 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.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
```

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\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\tr2\t\tr4".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
  - Use gsub ("global" sub) to replace all occurrences
- "r1\tr2\t\tr3".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

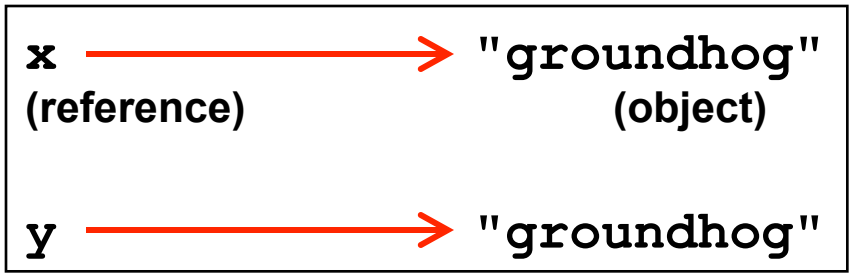
Regular Expressions!

Object Copy vs. Reference Copy

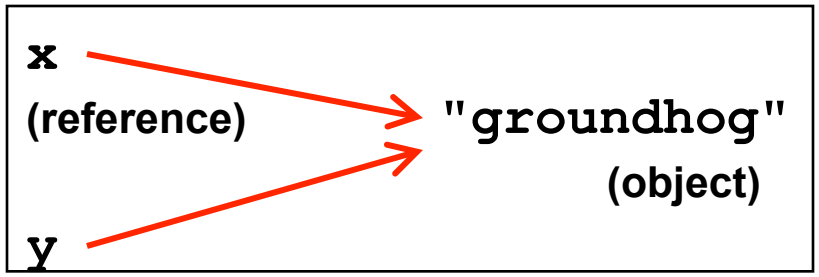
- Consider the following code
  - Assume an object/reference model like Java or Ruby
    - Or even two pointers pointing to the same structure

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

- Which of these occur?



Object copy



Reference copy

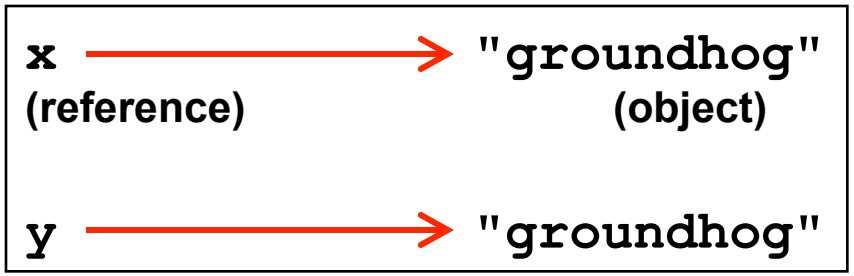
Object Copy vs. Reference Copy (cont.)

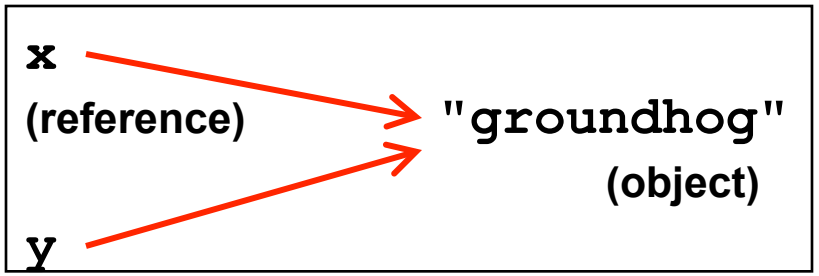
- For
```
x = "groundhog" ; y = x
```
  - Ruby and Java would both do a reference copy

- But for
```
x = "groundhog"
y = String.new(x)
```
  - Ruby would cause an object copy
  - Unnecessary in Java since Strings are final

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

Language	Physical equality	Structural equality
Java	**a == b**	**a.equals(b)**
C	**a == b**	***a == *b**
Ruby	**a.equal?(b)**	**a == b**
Ocaml	**a == b**	**a = b**
Python	**a is b**	**a == b**
Scheme	**(eq? a b)**	**(equal? a b)**
Visual Basic .NET	**a Is b**	**a = b**

String Operations in Ruby

- "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\tr2\t\tr3".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

- A way of describing patterns or sets of strings
  - Searching and matching
  - Formally describing strings
    - The symbols (lexemes or tokens) that make up a language
- Common to lots of languages and tools
  - awk, sed, perl, grep, Java, OCaml, C libraries, etc.
- Based on some really elegant theory
  - Next lecture

Example Regular Expressions in Ruby

- /Ruby/
  - Matches exactly the string "Ruby"
  - Regular expressions can be delimited by /'s
  - Use \ to escape /'s in regular expressions
- /(Ruby|OCaml|Java)/
  - Matches either "Ruby", "OCaml", or "Java"
- /(Ruby|Regular)/ or /R(uby|egular)/
  - Matches either "Ruby" or "Regular"
  - Use ( )'s for grouping; use \ to escape ( )'s

Using Regular Expressions

- Regular expressions are instances of Regexp
  - we'll see use of a Regexp.new later
- Basic matching using =~ method of String

```
line = gets                    # read line from standard input
if line =~ /Ruby/ then         # returns nil if not found
  puts "Found Ruby"
end
```

- Can use regular expressions in index, search, etc.

```
offset = line.index(/(MAX|MIN)/)    # search starting from 0
line.sub(/(Perl|Python)/, "Ruby")   # replace
line.split(/(\t|\n| )/)             # split at tab, space,
                                    # newline
```

Using Regular Expressions (cont.)

- Invert matching using !~ method of String
  - Matches strings that don't contain an instance of the regular expression

Repetition in Regular Expressions

- /(Ruby)*/
  - {"", "Ruby", "RubyRuby", "RubyRubyRuby", ...}
  - * means *zero or more occurrences*
- /Ruby+/
  - {"Ruby", "Rubyy", "Rubyyy", ... }
  - + means *one or more occurrence*
  - so /e+/ is the same as /ee*/
- /(Ruby)?/
  - {"", "Ruby"}
  - ? means *optional*, i.e., zero or one occurrence

Repetition in Regular Expressions

- /(Ruby){3}/
  - {“RubyRubyRuby”}
  - {x} means repeat the search for exactly x occurrences
- /(Ruby){3,}/
  - {“RubyRubyRuby”, “RubyRubyRubyRuby”, …}
  - {x,} means repeat the search for at least x occurrences
- /(Ruby){3, 5}/
  - {“RubyRubyRuby”, “RubyRubyRubyRuby”, “RubyRubyRubyRubyRuby”}
  - {x, y} means repeat the search for at least x occurrences and at most y occurrences

Watch Out for Precedence

- /(Ruby)*/ means {"", "Ruby", "RubyRuby", ...}
  - But /Ruby*/ matches {"Rub", "Ruby", "Rubyy", ...}
- In general
  - * {n} and + bind most tightly
  - Then concatenation (adjacency of regular expressions)
  - Then |
- Best to use parentheses to disambiguate

Character Classes

- /[abcd]/
  - {"a", "b", "c", "d"} (Can you write this another way?)
- /[a-zA-Z0-9]/
  - Any upper or lower case letter or digit
- /[^0-9]/
  - Any character except 0-9 (the ^ is like not and must come first)
- /[\t\n]/
  - Tab, newline or space
- /[a-zA-Z_\$][a-zA-Z_\$0-9]*/
  - Java identifiers ($ escaped...see next slide)

Special Characters

.	any character
^	beginning of line
$	end of line
\$	just a $
\d	digit, [0-9]
\s	whitespace, [\t\r\n\f]
\w	word character, [A-Za-z0-9_]
\D	non-digit, [^0-9]
\S	non-space, [^\t\r\n\f]
\W	non-word, [^A-Za-z0-9_]

Potential Character Class Confusions

- ^
  - Inside character classes: not
  - Outside character classes: beginning of line
- [ ]
  - Inside regular expressions: character class
  - Outside regular expressions: array
    - Note: [a-z] does not make a valid array
- ( )
  - Inside character classes: literal characters ( )
    - Note /(0..2)/ does not mean 012
  - Outside character classes: used for grouping
- –
  - Inside character classes: range (e.g., a to z given by [a-z])
  - outside character classes: subtraction

Regular Expression Practice

- Make Ruby regular expressions representing
  - All lines beginning with a or b `/^(a|b)/`
  - All lines containing at least two (only alphabetic) words separated by white-space `/[a-zA-Z]+\s+[a-zA-Z]+/`
  - All lines where a and b alternate and appear at least once `/^((ab)+ a?)|((ba)+ b?)$/`
  - An expression which would match both of these lines (but not radically different ones)
    - CMSC330: Organization of Programming Languages: Fall 2007
    - CMSC351: Algorithms: Fall 2007

Regular Expression Coding Readability

```
> ls -l
drwx------    2 sorelle  sorelle  4096 Feb 18 18:05 bin
-rw-------    1 sorelle  sorelle  674 Jun  1 15:27 calendar
drwx------    3 sorelle  sorelle  4096 May 11 12:19 cmsc311
drwx------    2 sorelle  sorelle  4096 Jun  4 17:31 cmsc330
drwx------    1 sorelle  sorelle  4096 May 30 19:19 cmsc630
drwx------    1 sorelle  sorelle  4096 May 30 19:20 cmsc631
```

What if we want to specify the format of this line exactly?

```
/^(d|-)(r|-)(w|-)(x|-)(r|-)(w|-)(x|-)(r|-)(w|-)(x|-)
(\s+)(\d+)(\s+)(\w+)(\s+)(\w+)(\s+)(\d+)(\s+)(Jan|Feb
|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec)(\s+)(\d\d)
(\s+)(\d\d:\d\d)(\s+)(\S+)$/
```

This is unreadable!

Regular Expression Coding Readability

Instead, we can do each part of the expression separately and then combine them:

```
oneperm_re = '((r|-)(w|-)(x|-))'
permissions_re = '(d|-)' + oneperm_re + '{3}'
month_re = '(Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec)'
day_re = '\d{1,2}';   time_re = '(\d{2}:\d{2})'
date_re = month_re + '\s+' + day_re + '\s+' + time_re
total_re = '\d+';  user_re = '\w+';   group_re = '\w+'
space_re = '\d+';  filename_re = '\S+'

line_re = Regexp.new('^' + permissions_re + '\s+' + total_re
       + '\s+' + user_re + '\s+' + group_re + '\s+' +
       space_re + '\s+' + date_re + '\s+' + filename_re + '$')

if line =~ line_re
        puts "found it!"
end
```

Extracting Substrings based on R.E.'s Method 1: Back References

Two options to extract substrings based on R.E.'s:

- Use back references
  - Ruby remembers which strings matched the parenthesized parts of r.e.'s
  - These parts can be referred to using special variables called back references (named $1, $2,…)

Back Reference Example

- Extract information from a report

```
gets =~ /^Min: (\d+)  Max: (\d+)$/
min, max = $1, $2
```

```
sets min = $1
 and max = $2
```

- Warning
  - Despite their names, $1 etc are local variables

```
def m(s)
  s =~ /(Foo)/
  puts $1    # prints Foo
end
m("Foo")
puts $1      # prints nil
```

Another Back Reference Example

- Warning 2
  - If another search is performed, all back references are reset to nil

```
gets =~ /(h)e(ll)o/
puts $1
puts $2
gets =~ /h(e)llo/
puts $1
puts $2
gets =~ /hello/
puts $1
```

```
hello
h
ll
hello
e
nil
hello
nil
```

Method 2: String.scan

- Also extracts substrings based on regular expressions
- Can optionally use parentheses in regular expression to affect how the extraction is done
- Has two forms which differ in what Ruby does with the matched substrings
  - The first form returns an array
  - The second form uses a code block
    - We'll see this later

First Form of the Scan Method

- *str*.scan(*regexp*)
  - If regexp doesn't contain any parenthesized subparts, returns an array of matches
    - An array of all the substrings of *str* which matched

```
s = "CMSC 330 Fall 2007"
s.scan(/\S+ \S+/)
# returns array ["CMSC 330", "Fall 2007"]
```

    - Note: these string are chosen sequentially from as yet unmatched portions of the string, so while “330 Fall” *does* match the regular expression above, it is *not* returned since “330” has already been matched by a previous substring.

First Form of the Scan Method (cont.)

- If regexp contains parenthesized subparts, returns an array of arrays
  - Each sub-array contains the parts of the string which matched one occurrence of the search

```
s = "CMSC 330 Fall 2007"
s.scan(/(\S+) (\S+)/)  # [["CMSC", "330"],
                       #  ["Fall", "2007"]]
```

  - Each sub-array has the same number of entries as the number of parenthesized subparts
  - All strings that matched the first part of the search (or $1 in back-reference terms) are located in the first position of each sub-array

Practice with Scan and Back-references

```
> ls -l
drwx------    2 sorelle  sorelle  4096 Feb 18 18:05 bin
-rw-------    1 sorelle  sorelle  674 Jun  1 15:27 calendar
drwx------    3 sorelle  sorelle  4096 May 11  2006 cmsc311
drwx------    2 sorelle  sorelle  4096 Jun  4 17:31 cmsc330
drwx------    1 sorelle  sorelle  4096 May 30 19:19 cmsc630
drwx------    1 sorelle  sorelle  4096 May 30 19:20 cmsc631
```

Extract just the file or directory name from a line using

- scan

```
name = line.scan(/\S+$/)  # ["bin"]
```

- back-references

```
if line =~ /(\S+$)/
      name = $1   # "bin"
end
```

Standard Library: Array

- Arrays of objects are instances of class Array
  - Arrays may be heterogeneous

    a = [1, "foo", 2.14]
  - C-like syntax for accessing elements, indexed from 0

    x = a[0]; a[1] = 37
- Arrays are *growable*
  - Increase in size automatically as you access elements

    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

```
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

```
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"
```

note: push, pop, shift, and unshift all permanently modify the array

Iterating Through Arrays

- It's easy to iterate over an array with while

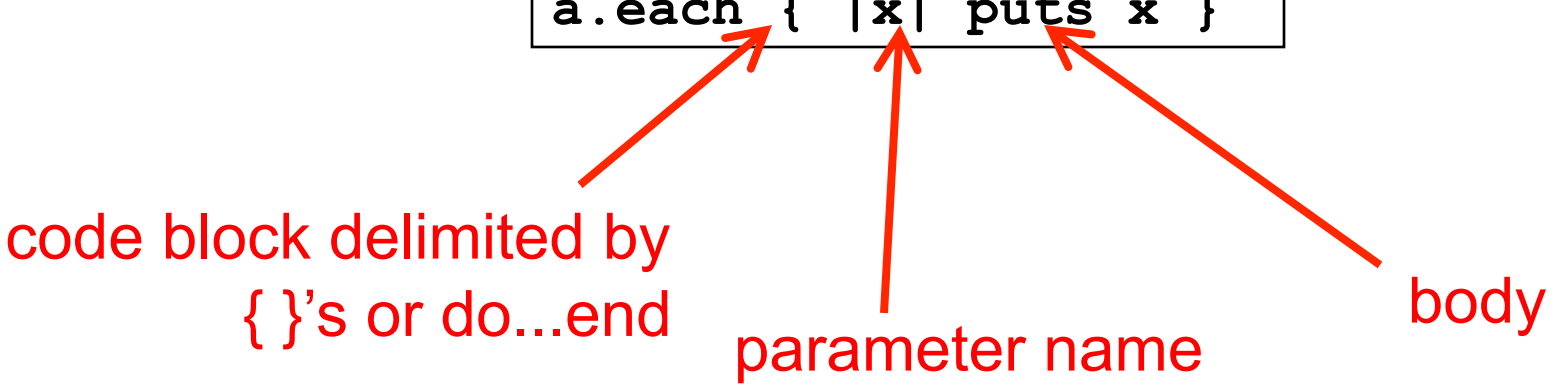
```
a = [1,2,3,4,5]
i = 0
while i < a.length
  puts a[i]
  i = i + 1
end
```

- 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

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



More Examples of Code Blocks

- Sum up the elements of an array

```
a = [1,2,3,4,5]
sum = 0
a.each { |x| sum = sum + x }
printf("sum is %d\n", sum)
```

- Print out each segment of the string as divided up by commas (commas are printed trailing each segment)
  - Can use any delimiter

```
s = "Student,Sally,099112233,A"
s.each(',') { |x| puts x }
```

("delimiter" = symbol used to denote boundaries)

Yet More Examples of Code Blocks

```
3.times { puts "hello"; puts "goodbye" }
5.upto(10) { |x| puts(x + 1) }
[1,2,3,4,5].find { |y| y % 2 == 0 }
[5,4,3].collect { |x| -x }
```

- n.times runs code block n times
- n.upto(m) runs code block for integers n..m
- a.find returns first element x of array such that the block returns true for x
- a.collect applies block to each element of array and returns new array (a.collect! modifies the original)

Still Another Example of Code Blocks

```
File.open("test.txt", "r") do |f|
  f.readlines.each { |line| puts line }
end
```

- open method takes code block with file argument
  - File automatically closed after block executed
- readlines reads all lines from a file and returns an array of the lines read
  - Use each to iterate

Using Yield To Call Code Blocks

- Any method can be called with a code block
  - Inside the method, the block is called with yield
- After the code block completes
  - Control returns to the caller after the yield instruction

```
def countx(x)
  for i in (1..x)
     puts i
     yield
  end
end

countx(4) { puts "foo" }
```

```
1
foo
2
foo
3
foo
4
foo
```

So What Are Code Blocks?

- A code block is just a special kind of method
  - { |y| x = y + 1; puts x } is almost the same as
  - def m(y) x = y + 1; puts x end
- The each method takes a code block as an argument
  - This is called higher-order programming
    - In other words, methods take other methods as arguments
    - We'll see a lot more of this in OCaml
- We'll see other library classes with each methods
  - And other methods that take code blocks as arguments
  - As we saw, your methods can use code blocks too!

Second Form of the Scan Method

- Remember the scan method?
  - Executing returns an array of matches
  - Can also take a code block as an argument

- str.scan(regexp) { |match| block }
  - Applies the code block to each match
  - Short for str.scan(regexp).each { |match| block }
  - The regular expression can also contain parenthesized subparts

Example of Second Form of Scan

```
12  34  23
20  77  87
13  98  3
2   45  0
```

input file:
will be read line by line, but
column summation is desired

```
sum_a = sum_b = sum_c = 0
while (line = gets)
  line.scan(/(\d+)\s+(\d+)\s+(\d+)/) { |a,b,c|
    sum_a += a.to_i
    sum_b += b.to_i
    sum_c += c.to_i
  }
end
printf("Total: %d %d %d\n", sum_a, sum_b, sum_c)
```

converts the string
to an integer

Sums up three columns of numbers

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

```
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

```
italy.keys.each {
    |key| puts("key: #{key}, value: #{italy[key]}")
}
```

Hash (cont.)

Convenient syntax for creating literal hashes

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

```
credits = {
  "cmsc131" => 4,
  "cmsc330" => 3,
}

x = credits["cmsc330"]  # x now 3
credits["cmsc311"] = 3
```

Standard Library: File

- Lots of convenient methods for IO

```
File.new("file.txt", "rw")      # open for rw access
f.readline                      # reads the next line from a file
f.readlines                     # returns an array of all file lines
f.eof                           # return true if at end of file
f.close                         # close file
f << object                     # convert object to string and write to f
$stdin, $stdout, $stderr        # global variables for standard UNIX IO
```

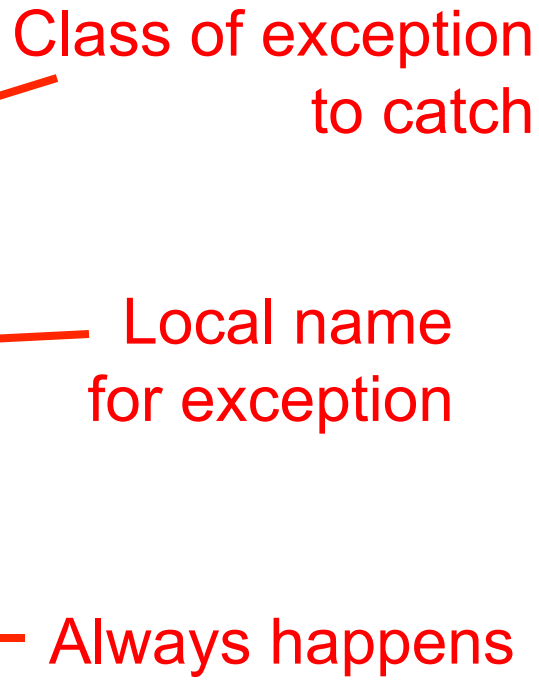
  By default stdin reads from keyboard, and stdout and stderr both write to terminal

- File inherits some of these methods from IO

Exceptions

- Use begin...rescue...ensure...end
  - Like try...catch...finally in Java

```
begin
  f = File.open("test.txt", "r")
  while !f.eof
    line = f.readline
    puts line
  end
rescue Exception => e
  puts "Exception:" + e.to_s +
   " (class " + e.class.to_s + ")"
ensure
  f.close
end
```



Command Line Arguments

- Stored in predefined array variable $*
  - Can refer to as predefined global constant ARGV

- Example
  - If
    - Invoke test.rb as “ruby test.rb a b c”
  - Then
    - ARGV[0] = “a”
    - ARGV[1] = “b”
    - ARGV[2] = “c”

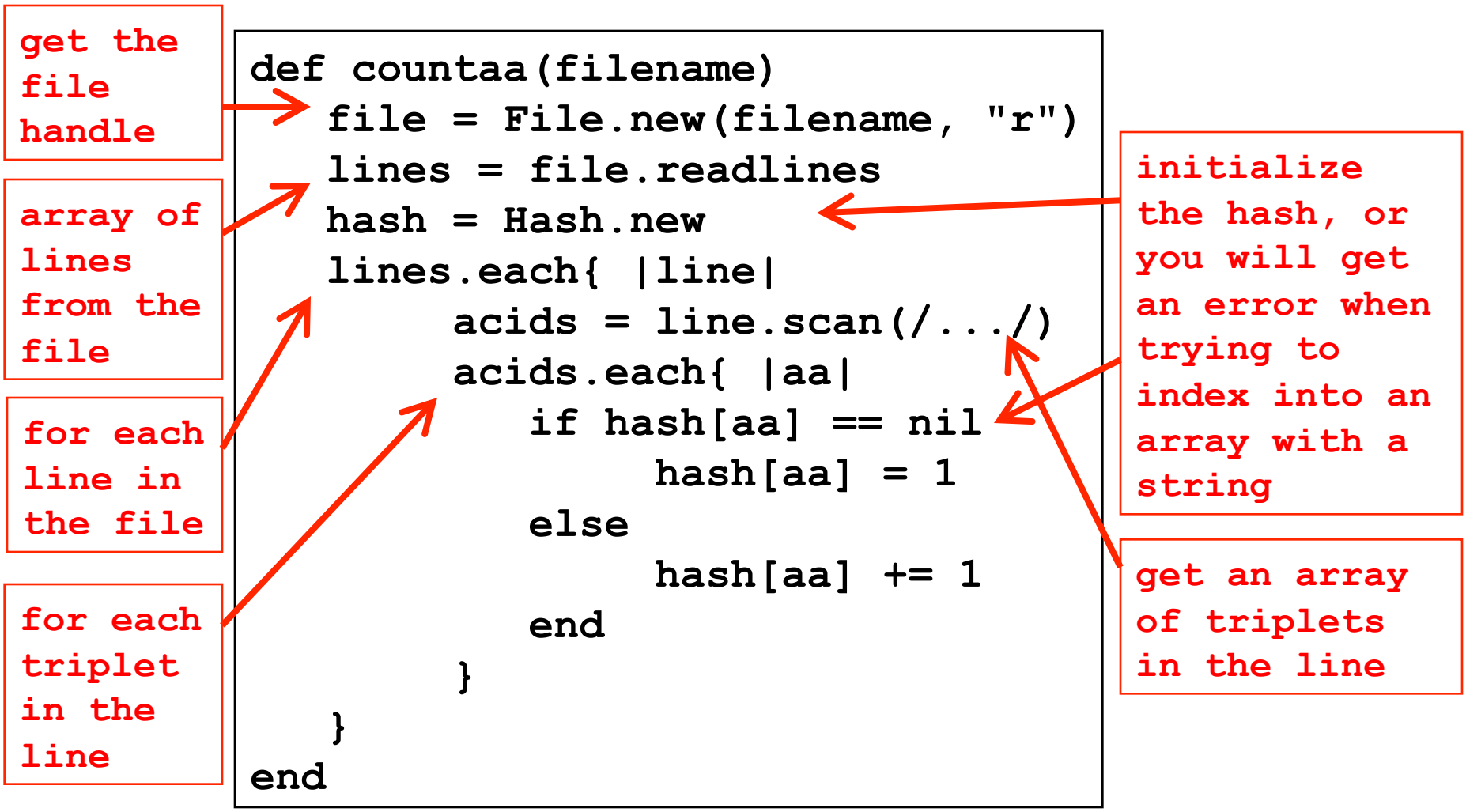
Practice: Amino Acid counting in DNA

Write a function that will take a filename and read through that file counting the number of times each group of three letters appears so these numbers can be accessed from a hash.

(assume: the number of chars per line is a multiple of 3)

```
gcggcattcagcacccgtatactgttaagcaatccagattttgtgtataacataccggc
catactgaagcattcattgaggctagcgctgataacagtagcgctaacaatgggggaatg
tggcaatacggtgcgattactaagagccgggaccacacaccccgtaaggatggagcgtgg
taacataataatccgttcaagcagtgggcgaaggtggagatgttccagtaagaatagtgg
gggcctactacccatggtacataattaagagatcgtcaatcttgagacggtcaatggtac
cgagactatatcactcaactccggacgtatgcgcttactggtcacctcgttactgacgga
```

Practice: Amino Acid counting in DNA



Ruby Summary

- Interpreted
- Implicit declarations
- Dynamically typed
- Built-in regular expressions
- Easy string manipulation
- Object-oriented
  - Everything (!) is an object
- Code blocks
  - Easy higher-order programming!
  - Get ready for a lot more of this...

Makes it quick to write small programs

Hallmark of scripting languages

Other Scripting Languages

- Perl and Python are also popular scripting languages
  - Also are interpreted, use implicit declarations and dynamic typing, have easy string manipulation
  - Both include optional “compilation” for speed of loading/execution
- Will look fairly familiar to you after Ruby
  - Lots of the same core ideas
  - All three have their proponents and detractors
  - Use whichever language you personally prefer

Example Perl Program

```
#!/usr/bin/perl
foreach (split//, $ARGV[0])) {
  if ($G{$_}) {
    $RE .= "\\" . $G{$_};
  } else {
    $RE .= $N ? "(?!\\" .
  join("|\\",values(%G)) . ')(\w)' : '(\w)';
    $G{$_} = ++$N;
}  }
```

Example Python Program

```
#!/usr/bin/python
import re
list = ("deep", "deer", "duck")
x = re.compile("^\S{3,5}.[aeiou]")
for i in list:
  if re.match(x, i):
    print I
  else:
    print
```