CMSC 330: Organization of Programming Languages

Introduction to Ruby

Last Lecture

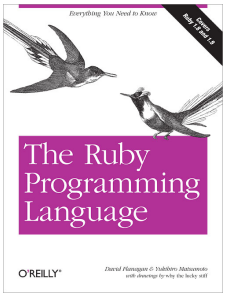
- Many types of programming languages
  - Imperative, functional, logical, OO, scripting
- Many programming language attributes
  - Clear, orthogonal, natural…
- Programming language implementation
  - Compiled, interpreted

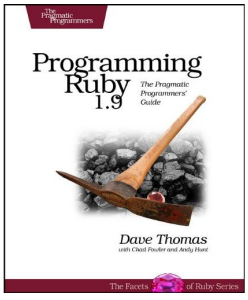
Introduction

- 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

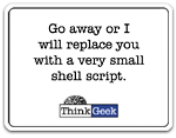




- Earlier version of Thomas book available on web
  - See course web page

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  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,
    0,0,572059,175306,343213,2006,14762,383,21728,14661,572059,527044,15861
    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
...
```

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")
```

no special main() function or method

line break separates expressions (can also use ";" to be safe)

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
  - Can type in Ruby programs one line at a time, and watch as each line is executed

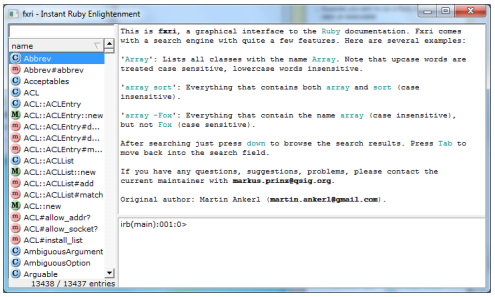
    irb(main):001:0> 3+4

    ⇒ 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'

- 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.)

```
#!/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 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

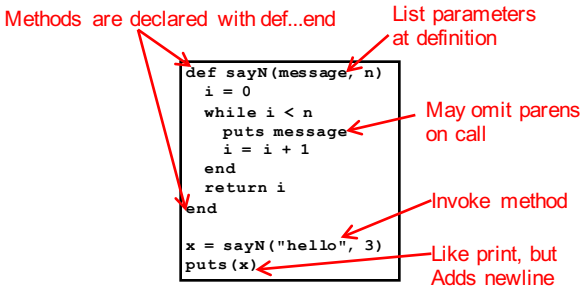
Explicit vs. Implicit Declarations

- 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?

Explicit Declarations	Implicit Declarations
More text to type	Less text to type
Helps prevent typos	Easy to mistype variable name
Forces programmer to document types	Variable not held to a fixed type (could imagine variable declarations without types)

Methods in Ruby



Methods should begin with lowercase letter and be defined before they are called
Variable names that begin with uppercase letter are *constants* (only assigned once)

Method Return Values

- Value of the return is the value of the last executed statement in the method
  - These are the same:

```
def add_three(x)
  return x+3
end
```

```
def add_three(x)
  x+3
end
```

- Methods can return multiple results (as a list)

```
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
...
```

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
```

- 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? 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!

Object	Class (aka *type*)
10	Fixnum
-3.30	Float
"CMSC 330"	String
String.new	String
['a', 'b', 'c']	Array
Fixnum	Class

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

Two Cool Things to Do with 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

```
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
```


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 when coding	Less work when coding
Helps prevent some subtle errors	Can use objects incorrectly and not discover until run time
Fewer programs type check	More programs type check

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
    ```
    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

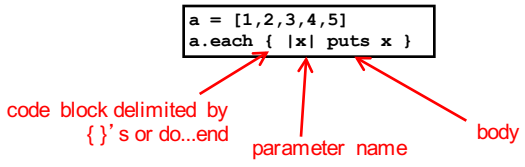
- It's easy to iterate over an array with while

```
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



- 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)
    - (1.3…2.7).include? 2.0 # ⇒ true
- 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

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

generates a string; cf. to_i

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

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

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

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

code block

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

does not need break

More Data-driven Control Statements

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

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

```
def even(x)
  x.times { |i|
    if i % 2 == 0
      puts i
    end
  }
end
```

```
def even(x)
  0.upto(x-1) { |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

```
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 { |k|
    print "key: ", k, " value: ", italy[k]
}

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


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
```


Defining Your Own Classes

```
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)
p.add_x(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

No Access To Internal State

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

A typical getter

```
def x
  @x
end
```

A typical setter

```
def x= (value)
  @x = value
end
```

- Very common, so Ruby provides a shortcut

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

Says to generate the x= and x and y= and y methods

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 run-time 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))
```

extend superclass

invoke add method of parent

```
b.is_a? A
true
b.instance_of? A
false
```

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

```
class OneDPoint
  attr_accessor "x"
  include Comparable
  def <=>(other)# used by Comparable
    if @x < other.x then return -1
    elsif @x > other.x then return 1
    else return 0
    end
  end
end
```

```
p = OneDPoint.new
p.x = 1
q = OneDPoint.new
q.x = 2
x < y # true
puts [y,x].sort
# prints x, then y
```


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
  - 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.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\nr2\n\nr4".each_line { |rec| puts rec }
    - Apply code block to each newline-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

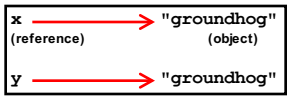
Regular Expressions!

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
```

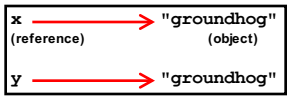
  - Ruby and Java would both do a reference copy
- But for

```
x = "groundhog"
y = String.new(x)
```

  - Ruby would cause an object copy
  - Unnecessary in Java since Strings are immutable

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**

Summary

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