Ruby

CSC 372, Spring 2015
The University of Arizona
William H. Mitchell
whm@cs

The Big Picture

Topic Sequence:

- Functional programming with Haskell (Done!)
- Imperative and object-oriented programming using dynamic typing with Ruby
- Logic programming with Prolog
- Whatever else in the realm of programming languages that we find interesting and have time for.

Introduction

From: Ralph Griswold <ralph@CS.Arizona.EDU>
Date: Mon, 18 Sep 2006 16:14:46 -0700

whm wrote:
> I ran into John Cropper in the mailroom a few minutes ago. He said
> he was out at your place today and that you're doing well. I
> understand you've got a meeting coming up regarding math in your
> weaving book -- sounds like fun!?

Hi, William

I'm doing well in the sense of surviving longer than expected. But
I'm still a sick person without much energy and with a lot of pain.
>
> My first lecture on Ruby is tomorrow. Ruby was cooked up by a
> Japanese fellow. Judging by the number of different ways to do the
> same thing, I wonder if Japanese has a word like "no".

Interesting. I know nothing about Ruby, but I've noticed it's
getting a lot of press, so there must be something to it.

Ralph's obituary: http://cs.arizona.edu/news/articles/200610-griswold.html

What is Ruby?

"A dynamic, open source programming language with a focus on simplicity and productivity. It has an elegant syntax that is natural to read and easy to write." — `ruby-lang.org`

Ruby is commonly described as an "object-oriented scripting language". I hate the term "scripting language"!

I describe Ruby as a dynamically typed object-oriented language.

Ruby was invented by Yukihiro Matsumoto ("Matz"), a "Japanese amateur language designer", in his own words.

Ruby on Rails, a web application framework, has largely driven Ruby's popularity.

Matz says...

Here is a second-hand excerpt of a posting by Matz:

> "Well, Ruby was born on February 24, 1993. I was talking with my colleague about the possibility of an object-oriented scripting language. I knew Perl (Perl4, not Perl5), but I didn't like it really, because it had smell of toy language (it still has). The object-oriented scripting language seemed very promising."

Another quote from Matz:

> "I believe that the purpose of life is, at least in part, to be happy. Based on this belief, Ruby is designed to make programming not only easy but also fun. It allows you to concentrate on the creative side of programming, with less stress. If you don’t believe me, read this book [the "pickaxe" book] and try Ruby. I’m sure you’ll find out for yourself."

Resources

The Ruby Programming Language by David Flanagan and Matz

- Perhaps the best book on Safari that covers 1.9 (along with 1.8)
- I'll refer to it as "RPL".

Programming Ruby 1.9 & 2.0 (4th edition): *The Pragmatic Programmers' Guide* by Dave Thomas, with Chad Fowler and Andy Hunt

- Known as the "Pickaxe book"
- $28 for a DRM-free PDF at pragprog.com.
- I'll refer to it as "PA".
- First edition is here: http://ruby-doc.com/docs/ProgrammingRuby/

Safari has lots of pre-1.9 books, lots of books that teach just enough Ruby to get one into the water with Rails, and lots of "cookbooks".

Resources, continued

ruby-lang.org

– Ruby's home page

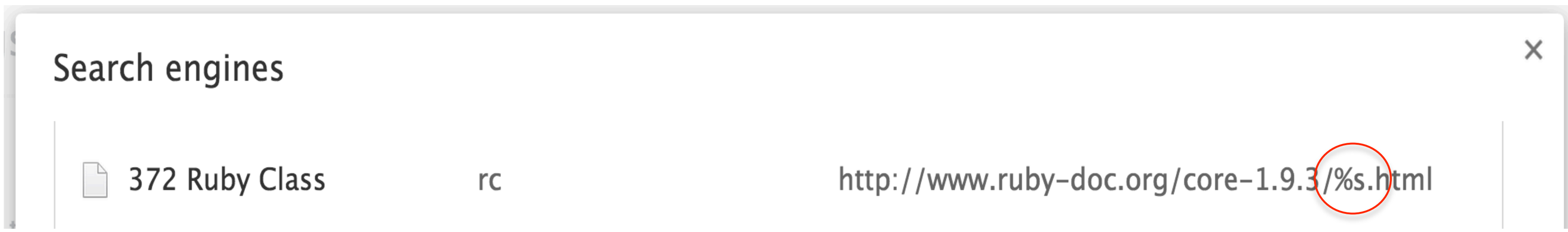
ruby-doc.org

– Documentation

– Here's a sample path, for the **String** class in 1.9.3:

http://www.ruby-doc.org/core-1.9.3/String.html

– I've got a Chrome "search engine" for Ruby classes:



Running Ruby

Experimenting with Ruby using **irb**

The **irb** command lets us evaluate Ruby expressions interactively.

irb can be run with no arguments but I usually start **irb** with a **bash** alias that specifies using a simple prompt and activates auto-completion:

```
alias irb="irb --prompt simple -r irb/completion"
```

On Windows you might use a batch file named **irbs.bat** to start with those options. Here's mine, in the directory where I'll be working with Ruby:

```
W:\372\ruby>type irbs.bat
irb --prompt simple -r irb/completion
```

I run it by typing **irbs** (not just **irb**).

Control-D terminates **irb** on all platforms.

irb, continued

irb evaluates expressions as they are typed.

```
>> 1+2
=> 3

>> "testing" + "123"
=> "testing123"
```

If you put in place the **~/.irbrc** file that I supply (see slide 20), you can use "**it**" to reference the last result:

```
>> it
=> "testing123"

>> it + it
=> "testing123testing123"
```

Note: To save space on the slides I'll typically not show the result line **(=> ...)** when it's uninteresting.

irb, continued

A couple more:

```
>> `ssh lec uptime`
=> " 18:00:58 up 10 days,  9:00, 99 users,  load average: 0.50, 0.32, 0.32\n"

>> it[-26,8]
=> "average:"
```

If an expression is definitely incomplete, irb displays an alternate prompt:

```
>> 1.23 +
?> 2e3
=> 2001.23
```

Extra Credit Assignment 2

For two assignment points of extra credit:

1. Run `irb` somewhere and try ten Ruby expressions with some degree of variety.

2. Capture the output and put it in a plain text file, eca2.txt, and turn it in via the eca2 D2L dropbox. (No need for your name, NetID, etc. in the file.)

Due: At the start of the next lecture after we hit this slide.

As of this slide we haven't seen much Ruby yet, so you'll need to look ahead in the slides to find some things to experiment with.

A section with some details on getting and running Ruby follows shortly.

Executing Ruby code in a file

The **ruby** command can be used to execute Ruby source code contained in a file.

By convention, Ruby files have the suffix **.rb**.

Here is "Hello" in Ruby:

```
% cat hello.rb
puts "Hello, world!"

% ruby hello.rb
Hello, world!
```

Windows, using a **.rb** file association:

```
W:\372\ruby>type hello.rb
puts "Hello, world!"

W:\372\ruby>hello.rb
Hello, world!
```

Note that the code does not need to be enclosed in a method—"top level" expressions are evaluated when encountered.

Executing Ruby code in a file, continued

Alternatively, code can be placed in a method that is invoked by an expression at the top level:

```
% cat hello2.rb
def say_hello
   puts "Hello, world!"
end

say_hello

% ruby hello2.rb
Hello, world!
```

The definition of **say_hello** must precede the call.

We'll see later that Ruby is somewhat sensitive to newlines.

A line-numbering program

Here's a program that reads lines from standard input and writes each, with a line number, to standard output:

```
line_num = 1        # numlines.rb

while line = gets
   printf("%3d: %s", line_num, line)
   line_num += 1    # Ruby does not have ++ and --
end
```

Execution:

```
% ruby numlines.rb < hello2.rb
  1: def say_hello
  2:     puts "Hello, world!"
  3: end
  4:
  5: say_hello
```

tac.rb

Problem: Write a program that reads lines from standard input and writes them in reverse order to standard output. Use only the Ruby you've seen.

For reference, here's the line-numbering program:

```
line_num = 1
while line = gets
   printf("%3d: %s", line_num, line)
   line_num += 1
end
```

Solution: (tac.rb)

```
reversed = ""
while line = gets
   reversed = line + reversed
end
puts reversed
```

Getting and running Ruby

Version issues

There is no written standard for Ruby. The language is effectively defined by MRI—Matz' Ruby Implementation.

The current stable version of MRI is 2.2.0.

If you take no special steps and run **`ruby`** on lectura, you'll get version 1.8.7. (See following re **`rvm`**.)

On Windows, version 1.9.3 is recommended.

OS X Mavericks and Yosemite have Ruby 2.0 installed. Mountain Lion has 1.8.7.

There are few significant differences between 1.9.3 and 2.X, especially wrt. the things we'll be doing.

<u>These slides use Ruby 1.9.3</u>.

Ruby on lectura

If you don't do anything special on lectura, you get an old version of Ruby!

```
$ irb
>> RUBY_VERSION
=> "1.8.7"
>> (control-D to exit)
```

To get 1.9.3, use **rvm** each time you login:

```
$ rvm 1.9   # (Ignore error iff irb and ruby versions (below) match!)
$ irb
>> RUBY_VERSION
=> "1.9.3"

$ ruby --version
ruby 1.9.3p484 (2013-11-22 revision 43786) [x86_64-linux]
```

To get the **.irbrc** file I recommend, do this:

```
$ cp /cs/www/classes/cs372/spring15/ruby/dotirbrc  ~/.irbrc
```

Ruby on Windows

Go to **http://rubyinstaller.org/downloads/** and get "**Ruby 1.9.3-p551**".

When installing, I recommend these selections:

Install Tcl/Tk support

Add Ruby executables to your PATH

Associate **.rb** and **.rbw** files with this Ruby installation

Here's a URL for the **.irbrc** file I recommend:

http://www.cs.arizona.edu/classes/cs372/spring15/ruby/dotirbrc

Copy the downloaded **dotirbrc** into your home directory on Windows, naming it **.irbrc**. On my (old) XP box, I'd do this:

```
c:>copy dotirbrc "c:\Documents and Settings\whm\.irbrc"
```

Ruby on OS X

Ruby 2.0 comes with Mavericks and Yosemite. For what we'll be doing I don't believe you'll see any compatibility issues between 2.0 on your Mac and 1.9.3 on lectura. (Let me know if any issues arise!)

I installed Ruby 1.9.3 on Yosemite using MacPorts.

https://www.ruby-lang.org/en/installation/ shows some other options.

To copy the recommended **.irbrc** file into place you might do this:

```
$ scp YOUR-NETID@lec:/cs/www/classes/cs372/spring15/ruby/dotirbrc ~/.irbrc
```

(assumes **lec** in **/etc/hosts** as described in *Unix Stuff for 372*)

Ruby basics

Every value is an object

In Ruby every value is an object.

Methods can be invoked using ***receiver*.*method*(*parameters...*)**

```
>> "testing".count("t")     # How many "t"s are there?
 => 2

>> "testing".slice(1,3)
=> "est"

>> "testing".length()
=> 7
```

Repeat: In Ruby every value is an object.

What are some values in Java that are not objects?

Everything is an object, continued

Parentheses can be omitted from an argument list:

```
>> "testing".count "aeiou"
=> 2

>> "testing".slice 1,3
=> "est"
```

If a method doesn't require any arguments, the argument list can be omitted.

```
>> "testing".length
=> 7
```

See Q6 in a4 FAQS for an example!

<u>There are some syntactic anomalies that are occasionally encountered when an argument list is present and parentheses are omitted.</u>

Some sources recommend always using parentheses with arguments, but I often omit them.

Everything is an object, continued

Of course, "everything" includes numbers:

```
>> 1.2.class
=> Float

>> (10-20).class
=> Fixnum

>> 17**25
=> 5770627412348402378939569991057

>> it.succ        # Remember: the custom .irbc is needed to use "it"
=> 5770627412348402378939569991058

>> it.class
=> Bignum
```

Everything is an object, continued

The TAB key can be used to show completions:

```
>> 100.<TAB><TAB>
Display all 107 possibilities? (y or n)
100.__id__                        100.display
100.__send__                      100.div
100.abs                           100.divmod
100.abs2                          100.downto
100.angle                         100.dup
100.arg                           100.enum_for
100.between?                      100.eql?
100.ceil                          100.equal?
100.chr                           100.even?
100.class                         100.extend
100.clone                         100.fdiv
100.coerce                        100.floor
100.conj                          100.freeze
100.conjugate                     100.frozen?
100.define_singleton_method       100.gcd
100.denominator                   100.gcdlcm
```

Methods from **Kernel**

Methods from the **Kernel** module are available in every method and in top-level expressions.

gets, **puts**, **printf** and many more reside in **Kernel**.

```
>> puts 2,"three"        # Instead of Kernel.puts 2, "three"
2
three
=> nil

>> printf "sum = %d, product = %d\n", 3+4, 3 * 4
sum = 7, product = 12
=> nil
```

See http://www.ruby-doc.org/core-1.9.3/Kernel.html
(With the **rc** "search engine" I type **cmd-L rc Kernel RET**)

Some basic types

The value **nil**

nil is Ruby's "no value" value. The name **nil** references the only instance of the class.

```
>> nil
=> nil

>> nil.class
=> NilClass

>> nil.object_id
=> 4
```

We'll see that Ruby uses **nil** in a variety of ways.

Speculate: Do uninitialized variables have the value **nil**?

```
>> x
NameError: undefined local variable or method `x' for main
```

Strings and string literals

Instances of Ruby's **String** class represent character strings.

A variety of "escapes" are recognized in double-quoted string literals:

```
>> puts "newline >\n< and tab >\t<"
newline >
< and tab >	<

>> "\n\t\\".length
=> 3

>> "Newlines: octal \012, hex \xa, control-j \cj"
=> "Newlines: octal \n, hex \n, control-j \n"
```

Section 3.2, page 49 in RPL has the full list of escapes.

String literals, continued

In <u>single-quoted</u> literals only `\'` and `\\` are recognized as escapes:

```
>> puts '\n\t'
\n\t
=> nil

>> '\n\t'.length     # Four chars: backslash, n, backslash, t
=> 4

>> puts '\'\\'
'\
=> nil

>> '\'\\'.length  # Two characters: apostrophe, backslash
=> 2
```

String has a lot of methods

The **public_methods** method shows the public methods that are available for an object. Here are some of the methods for **String**:

```
>> "abc".public_methods.sort
=> [:!, :!=, :!~, :%, :*, :+, :<, :<<, :<=, :<=>, :==, :===, :=~,
 :>, :>=, :[], :[]=, :__id__, :__send__, :ascii_only?,
 :between?, :bytes, :bytesize, :byteslice, :capitalize, :capitalize!
, :casecmp, :center, :chars, :chomp, :chomp!, :chop, :chop!, :chr
, :class, :clear, :clone, :codepoints, :concat, :count, :crypt, :defi
ne_singleton_method, :delete, :delete!, :display, :downcase, :d
owncase!, :dump, :dup, :each_byte, :each_char, :each_codepoi
nt, :each_line, :empty?, ...

>> "abc".public_methods.length
=> 164
```

Strings are mutable

Unlike Java, Haskell, and many other languages, strings in Ruby are mutable.

If two variables reference a string and the string is changed, the change is reflected by both variables:

```
>> x = "testing"

>> y = x   # x and y now reference the same instance of String

>> x.upcase!
=> "TESTING"

>> y
=> "TESTING"
```

Convention: If there are both applicative and imperative forms of a method, the name of the imperative form ends with an exclamation mark.

Strings are mutable, continued

The **dup** method produces a copy of a string.

```
>> x = "testing"
>> y = x.dup
=> "testing"

>> y.upcase!
>> y
=> "TESTING"

>> x
=> "testing"
```

Some objects that hold strings **dup** the string when the string is added to the object.

String comparisons

Strings can be compared with a typical set of operators:

```
>> s1 = "apple"

>> s2 = "testing"

>> s1 == s2
=> false

>> s1 != s2
=> true

>> s1 < s2
=> true
```

We'll talk about details of **true** and **false** later.

String comparisons, continued

There is also a *comparison operator*: **<=>**

It produces -1, 0, or 1 depending on whether the first operand is less than, equal to, or greater than the second operand.

```
>> "apple" <=> "testing"
=> -1

>> "testing" <=> "apple"
=> 1

>> "x" <=> "x"
=> 0
```

This operator is sometimes called "spaceship".

Substrings

Subscripting a string with a number produces a one-character string.

```
>> s="abcd"

>> s[0]          # Positions are zero-based
=> "a"

>> s[1]
=> "b"

>> s[-1]         # Negative positions are counted from the right
=> "d"

>> s[100]
=> nil           # IMPORTANT!!
```

Historical note: With Ruby versions prior to 1.9, `"abc"[0]` is 97.

Why doesn't Java provide `s[n]` instead of `s.charAt(n)`?

Substrings, continued

A subscripted string can be the target of an assignment. A string of any length can be assigned.

```
>> s = "abc"
=> "abc"

>> s[0] = 65.chr
=> "A"

>> s[1] = "tomi"

>> s
=> "Atomic"

>> s[-3] = ""

>> s
=> "Atoic"
```

Substrings, continued

A substring can be referenced with

s[*start*, *length*]

```
r e p l a c e
0 1 2 3 4 5 6
7 6 5 4 3 2 1 (negative)
```

```
>> s = "replace"

>> s[2,3]
=> "pla"

>> s[3,100]          # Note too-long behavior!
=> "lace"

>> s[-4,3]
=> "lac"

>> s[10,10]
=> nil
```

Substrings with ranges

Instances of Ruby's **Range** class represent a range of values. Ranges can be used to reference a substring.

```
>> r = 2..-2
=> 2..-2

>> r.class
=> Range

>> s = "replaced"

>> s[r]
=> "place"

>> s[r] = ""

>> s
=> "red"
```

It's more common to use literal ranges with strings:

```
>> s = "rebuilding"
>> s[2..-1]      # the common case
=> "building"

>> s[2..-4]
=> "build"

>> s[2...-3]     # three dots is "up to"
=> "build"
```

Changing substrings

A substring can be the target of an assignment:

```
>> s = "replace"

>> s[0,2] = ""
=> ""

>> s
=> "place"

>> s[3..-1] = "naria"
>> s
=> "planaria"

>> s["aria"] = "kton"  # If "aria" appears, replace it (error if not).

=> "kton"

>> s
=> "plankton"
```

r	e	p	l	a	c	e	
0	1	2	3	4	5	6	
7	6	5	4	3	2	1	*(negative)*

p	l	a	c	e	
0	1	2	3	4	
5	4	3	2	1	*(negative)*

Interpolation in string literals

In a string literal enclosed with double quotes, or specified with a "here document", the sequence #{*expr*} causes interpolation of *expr*, an arbitrary Ruby expression.

```
>> x = 10

>> y = "twenty"

>> s = "x = #{x}, y + y = #{y + y}"
=> "x = 10, y + y = twentytwenty"

>> puts "There are #{"".public_methods.length} string methods"
There are 164 string methods

>> "test #{"#{"abc".length*4}"}"     # Arbitrary nesting works
=> "test 12"
```

It's idiomatic to use interpolation rather than concatenation to build a string from multiple values.

Numbers

With 1.9.3 on lectura, integers in the range -2^{62} to $2^{62}-1$ are represented by instances of **Fixnum**. If an operation produces a number outside of that range, the value is represented with a **Bignum**.

```
>> x = 2**62-1
=> 4611686018427387903

>> x.class          => Fixnum

>> x += 1           => 4611686018427387904

>> x.class          => Bignum

>> x -= 1           => 4611686018427387903

>> x.class          => Fixnum
```

Is this automatic transitioning between **Fixnum** and **Bignum** a good idea? How do other languages handle this?

Numbers, continued

The **Float** class represents floating point numbers that can be represented by a double-precision floating point number on the host architecture.

```
>> x = 123.456
=> 123.456

>> x.class
=> Float

>> x ** 0.5
=> 11.111075555498667

>> x = x / 0.0
=> Infinity

>> (0.0/0.0).nan?
=> true
```

Numbers, continued

Fixnums and **Float**s can be mixed. The result is a **Float**.

```
>> 10 / 5.1
=> 1.9607843137254903

>> 10 % 4.5    => 1.0

>> it.class    => Float
```

Numbers, continued

Ruby has a **Complex** type.

```
>> Complex(2,3)
=> (2+3i)

>> Complex 'i'
=> (0+1i)

>> it * it
=> (-1+0i)
```

Numbers, continued

There's **Rational**, too.

```
>> Rational(1,3)
=> (1/3)

>> it * 300
=> (100/1)

>> Rational 0.5
=> (1/2)

>> Rational 0.6
=> (5404319552844595/9007199254740992)

>> Rational 0.015625
=> (1/64)
```

Conversions

Unlike some languages, Ruby does not automatically convert strings to numbers and numbers to strings as needed.

```
>> 10 + "20"
TypeError: String can't be coerced into Fixnum
```

The methods **to_i**, **to_f**, and **to_s** are used to convert values to **Fixnum**s, **Float**s and **String**s, respectively.

```
>> 10.to_s + "20"
=> "1020"

>> 10 + "20".to_f
=> 30.0

>> 10 + 20.9.to_i
=> 30
```

```
>> 33.to_<TAB><TAB>
33.to_c        33.to_int
33.to_enum     33.to_r
33.to_f        33.to_s
33.to_i
```

TAKE A BREAK?

Arrays

A sequence of values is typically represented in Ruby by an instance of `Array`.

An array can be created by enclosing a comma-separated sequence of values in square brackets:

```
>> a1 = [10, 20, 30]
=> [10, 20, 30]

>> a2 = ["ten", 20, 30.0, 2**40]
=> ["ten", 20, 30.0, 1099511627776]

>> a3 = [a1, a2, [[a1]]]
=> [[10, 20, 30], ["ten", 20, 30.0, 1099511627776], [[[10, 20, 30]]]]
```

What's a difference between Ruby arrays and Haskell lists?

Arrays, continued

Array elements and subarrays (sometimes called slices) are specified with a notation like that used for strings.

```
>> a = [1, "two", 3.0, %w{a b c d}]
=> [1, "two", 3.0, ["a", "b", "c", "d"]]

>> a[0]
=> 1

>> a[1,2]
=> ["two", 3.0]

>> a[-1]
=> ["a", "b", "c", "d"]

>> a[-1][-2]
=> "c"
```

Arrays, continued

Elements and subarrays can be assigned to. Ruby accommodates a variety of cases; here are some:

```
>> a = [10, 20, 30, 40, 50, 60]

>> a[1] = "twenty"; a
=> [10, "twenty", 30, 40, 50, 60]

>> a[2..4] = %w{a b c d e}; a
=> [10, "twenty", "a", "b", "c", "d", "e", 60]

>> a[1..-1] = []; a
=> [10]

>> a[0] = [1,2,3]; a                => [[1, 2, 3]]

>> a[4] = [5,6]; a                  => [[1, 2, 3], nil, nil, nil, [5, 6]]

>> a[0,2] = %w{a bb ccc}; a  => ["a", "bb", "ccc", nil, nil, [5, 6]]
```

Arrays, continued

A variety of operations are provided for arrays. Here's a sampling:

```
>> a = []

>> a << 1; a
=> [1]

>> a << [2,3,4]; a
=> [1, [2, 3, 4]]

>> a.reverse!; a
=> [[2, 3, 4], 1]
```

Arrays, continued

A few more:

```
>> a
=> [[2, 3, 4], 1]

>> a[0].shift
=> 2

>> a
=> [[3, 4], 1]

>> a.unshift "a","b","c"
=> ["a", "b", "c", [3, 4], 1]

>> a.shuffle.shuffle
=> ["a", [3, 4], "b", "c", 1]
```

Arrays, continued

Even more! ☺

```
>> a = [1,2,3,4]; b = [1,3,5]

>> a + b
=> [1, 2, 3, 4, 1, 3, 5]

>> a - b
=> [2, 4]

>> a & b
=> [1, 3]

>> a | b
=> [1, 2, 3, 4, 5]

>> ('a'..'zzz').to_a.size
=> 18278
```

Comparing arrays

We can compare arrays with `==` and `!=`. Elements are compared in turn, possibly recursively.

```
>> [1,2,3] == [1,2]
=> false

>> [1,2,[3,"bcd"]] == [1,2] + [[3, "abcde"]]
=> false

>> [1,2,[3,"bcd"]] == [1,2] + [[3, "abcde"[1..-2]]]
=> true
```

Comparing arrays

Comparison of arrays with <=> is lexicographic.

```
>> [1,2,3,4] <=> [1,2,10]
=> -1

>> [[10,20],[2,30], [5,"x"]].sort
=> [[2, 30], [5, "x"], [10, 20]]
```

Comparing arrays

Comparison with **<=>** produces **nil** if differing types are encountered.

```
>> [1,2,3,4] <=> [1,2,3,"four"]
=> nil
```

Tie!

```
>> [[10,20],[5,30], [5,"x"]].sort
ArgumentError: comparison of Array with Array failed
```

Here's a simpler failing case. Should it be allowed?

```
>> ["sixty",20,"two"].sort
ArgumentError: comparison of String with 20 failed
```

Contrast with Icon:

```
][ sort([3.0, 7, 2, "a", "A", ":", [2], [1], -1.0])
  r := [2, 7, -1.0, 3.0, ":", "A", "a", [2], [1]]  (list)
```

What does Icon do better? What does Icon do worse? How about Python?

Arrays can be cyclic

An array can hold a reference to itself:

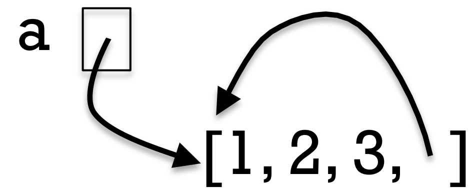
```
>> a = [1,2,3]

>> a.push a
=> [1, 2, 3, [...]]

>> a.size
=> 4

>> a[-1]
=> [1, 2, 3, [...]]

>> a[-1][-1][-1]
=> [1, 2, 3, [...]]
```




```
>> a << 10
=> [1, 2, 3, [...], 10]

>> a[-2][-1]
=> 10
```

Type Checking

Static typing

"The Java programming language is a statically typed language, which means that every variable and every expression has a type that is known at compile time."

-- *The Java Language Specification, Java SE 7 Edition*

Assume the following:

```
int i = ...;   String s = ...;   Object o = ...;   static int f(int n);
```

What are the types of the following expressions?

```
i + 5
i + s
s + o
o + o
o.hashCode()
f(i.hashCode())
i = i + s
```

Did we need to know any values or execute any code to determine those types?

Static typing, continued

Java does type checking based on the declared types of variables and the intrinsic types of literals.

Haskell supports type declarations but also provides type inferencing.

What are the inferred types for `x`, `y`, and `z` in the following expression?

```
(isLetter $ head $ [x] ++ y) && z
```

```
> let f x y z = (isLetter $ head $ [x] ++ y) && z
f :: Char -> [Char] -> Bool -> Bool
```

Did we need to know any values or execute any code to determine those types?

Haskell is a statically typed language—the type of every expression can be determined by analyzing the code.

Static typing, continued

With a statically typed language the type for all expressions is determined when a body of code is compiled. Any type inconsistencies that exist are discovered during compilation.

<u>Without having to run any code a statically typed language lets us guarantee that various types of errors don't exist</u>. Examples:

- Dividing a string by a float
- Taking the "head" of an integer
- Concatenating two numbers
- Putting an integer in a list of strings

Static typing, continued

How often did your Haskell code run correctly as soon as the type errors were fixed?

How does that compare with your experience with Java?
With C?
With Python?

"The best news is that Haskell's type system will tell you if your program is well-typed before you run it. This is a big advantage because most programming errors are manifested as typing errors."—Paul Hudak, Yale

Do you agree with Hudak?

Variables in Ruby have no type

In Java, variables are declared to have a type.

Variables in Ruby do not have a type. Instead, type is associated with values.

```
>> x = 10
>> x.class          What's the class of the object held in x?
=> Fixnum

>> x = "ten"
>> x.class
=> String

>> x = 2**100
>> x.class
=> Bignum
```

Dynamic typing

Ruby is a dynamically typed language. **There is no static analysis of the types involved in expressions**.

Consider this Ruby method:

```
def f x, y, z
  return x[y + z] * x.foo
end
```

For some combinations of types it will produce a value. For others it will produce a `TypeError`.

With dynamic typing such methods are allowed to exist.

Dynamic typing, continued

With dynamic typing, no type checking is done when code is compiled. Instead, types of values are checked during execution, as each operation is performed.

Consider this Ruby code:

```
while line = gets
  puts(f(line) + 3 + g(line)[-2])
end
```

What types must be checked each time through that loop?

Wrt. static typing, what are the implications of dynamic typing for...

- Compilation speed?
  - Probably faster!
- Execution speed?
  - Probably slower!
- Reliability?
  - It depends...

Can testing compensate?

A long-standing question in industry:

Can a good test suite find type errors in dynamically typed code as effectively as static type checking?

What's a "good" test suite?

Full code coverage? (every line executed by some test)

Full path coverage? (all combinations of paths exercised)

How about functions whose return type varies?

But wouldn't we want a good test suite no matter what language we're using?

"Why have to write tests for things a compiler can catch?"

--Brendan Jennings, SigFig

What ultimately matters?

What does the end-user of software care about?

Software that works
    Facebook game vs. radiation therapy system

Fast enough
    When does 10ms vs. 50ms matter?

Better sooner than later
    A demo that's a day late for a trade show isn't worth much.

Affordable
    How much more would you pay for a version of Destiny that has half as many bugs?

    I'd pay A LOT for a version of PowerPoint with more keyboard shortcuts!

Variety in type checking

Java is statically typed but casts introduce the possibility of a type error not being detected until execution.

C is statically typed but has casts that allow type errors during execution that are <u>never</u> detected.

Ruby, Python, and Icon have no static type checking whatsoever, but type errors during execution are <u>always</u> detected.

An example of a typing-related trade-off in execution time:

- C spends zero time during execution checking types.
- Java checks types during execution only in certain cases.
- Languages with dynamic typing check types on every operation, at least conceptually.

Is type inferencing applicable in a dynamically typed language?

Take a break then have a QUIZ!
(Hint: I first planned to give this quiz at the start of class today!)

"Why?" vs. "Why Not?"

"Why?" or "Why not?"

When designing a language some designers ask,
"Why should feature X be included?"

Some designers ask the opposite:
"Why should feature X <u>not</u> be included?"

Let's explore that question with Ruby.

More string literals!

A "here document" is a third way to literally specify a string.

```
>> s = <<XYZZY
    +------+
    | \\\  |
    | \*/  |
    | '''  |
    +------+
XYZZY
=> "    +------+\n    | \\  |\n    | */  |\n    | '''  |\n    +------+\n"
```

The string following **<<** specifies a delimiter that ends the literal. The ending occurrence must be at the start of a line.

"There's more than one way to do it!"—a Perl motto

And that's not all!

Here's another way to specify string literals. See if you can discern some rules from these examples:

```
>> %q{ just testin' this... }
=> " just testin' this... "

>> %Q|\n\t|
=> "\n\t"

>> %q(\u0041 is Unicode for A)
=> "\\u0041 is Unicode for A"

>> %q.test.
=> "test"
```

`%q` follows single-quote rules. `%Q` follows double quote rules. Symmetrical pairs like (), {}, and <> can be used.

How much is enough?

Partial summary of string literal syntax in Ruby:

```
>> x = 5; s = "x is #{x}"
=> "x is 5"

>> '\'\\\n\t'.length
=> 6

>> hd = <<X
just
testing
X
=> "just\ntesting\n"
```

How many ways does Haskell have to make a string literal?

How many ways should there be to make a string literal?

What's the minimum functionality needed?

Which would you remove?

```
>> %q{ \n \t } + %Q|\n \t | + %Q(\u0021 \u{23})
=> " \\n \\t \n \t ! #"
```

"Why" or "Why not?" as applied to operator overloading

Here are some examples of operator overloading:

```
>> [1,2,3] + [4,5,6] + [ ] + [7]
=> [1, 2, 3, 4, 5, 6, 7]

>> "abc" * 5
=> "abcabcabcabcabc"

>> [1, 3, 15, 1, 2, 1, 3, 7] - [3, 2, 1, 3]
=> [15, 7]

>> [10, 20, 30] * "..."
=> "10...20...30"

>> "decimal: %d, octal: %o, hex: %x" % [20, 20, 20]
=> "decimal: 20, octal: 24, hex: 14"
```

"Why" or "Why not?", continued

What are some ways in which inclusion of a feature impacts a language?

- Increases the "mental footprint" of the language.
- Maybe makes the language more expressive.
- Maybe makes the language useful for new applications.
- Probably increases size of implementation and documentation.
- Might impact performance.

Features come in all sizes!

Small: A new string literal escape sequence ("`\U{65}`" for "`A`")

Medium: Support for arbitrary precision integers

Large or small?

Support for object-oriented programming

Support for garbage collection

What would Ralph do?

At one of my first meetings with Ralph Griswold I put forth a number of ideas I had for new features for Icon.

He listened patiently. When I was done he said,
"Go ahead. Add all of those you want to."

As I was accelerating down the hall to my office he added,
"But for every feature you add, first find one to remove."

The art of language design

There's a lot of science in programming language design but there's art, too.

Excerpt from interview with Perl Guru Damian Conway:

Q: "What languages other than Perl do you enjoy programming in?"

A: "I'm very partial to Icon. It's so beautifully put together, so elegantly proportioned, almost like a Renaissance painting."

http://www.pair.com/pair/current/insider/1201/damianconway.html (404 now!)

"Icon: A general purpose language known for its elegance and grace. Designed by Ralph Griswold to be successor to SNOBOL4."
-- Digibarn "Mother Tongues" chart (see Intro slides)

Between SNOBOL4 and Icon there was there SL5 (SNOBOL Language 5).

I think of SL5 as an example of the "Second System Effect". It was never released.

Ralph once said, "I was laying in the hospital thinking about SL5. I felt there must be something simpler." That turned out to be Icon.

Design example: invocation in Icon

Procedure call in Icon:

```
][ reverse("programming")
  r := "gnimmargorp"  (string)

][ p := reverse
  r := function reverse  (procedure)

][ p("foo")
  r := "oof"  (string)
```

Doctoral student Steve Wampler added mutual goal directed evaluation (MGDE). A very trivial example:

```
][ 3("one", 2, "III")
  r := "III"  (string)

][ (?3)("one", 2, "III")
  r := "one"  (string)
```

Invocation in Icon, continued

After a CSC 550A lecture where Ralph introduced MGDE, I asked,

"How about 'string invocation', so that `"+"(3,4)` would be `7`?"

What do you suppose Ralph said?

"How would we distinguish between unary and binary operators?"

Solution: Discriminate based on the operand count!

```
][ "-"(5,3)
  r := 2  (integer)
][ "-"(5)
  r := -5  (integer)
][ (?"+-")(3,4)
  r := -1  (integer)
```

Within a day or two I added string invocation to Icon.

Why did Ralph choose to allow this feature?

He felt it would increase the research potential of Icon.

Control Structures

The **while** loop

Here is a loop to print the numbers from 1 through 10, one per line.

```
i=1
while i <= 10 do        # "do" is optional
   puts i
   i += 1
end
```

When **i <= 10** produces **false**, control branches to the code following **end**, if any.

The body of the **while** is always terminated with **end**, even if there's only one expression in the body.

while, continued

In Java control structures like **if**, **while**, and **for** are driven by the result of expressions that produce a value whose type is **boolean**.

C has a more flexible view: control structures consider an integer value that is non-zero to be "true".

PHP considers zeroes, the empty string, "0", empty arrays (and more) to be false.

Python, too, has a set of "falsey/falsy" values.

Here's the Ruby rule:

Any value that is not **false** or **nil** is considered to be "true".

Remember: Any value that is not **false** or **nil** is considered to be "true".

Consider this loop, which reads lines from standard input using **gets**.

```
while line = gets
    puts line
end
```

gets returns a string that is the next line of the input, or **nil**, on end of file.

The expression **line = gets** has two side effects but also produces a value.
Side effects: (1) a line is read from standard input and (2) is assigned to **line**.
Value: The string assigned to **line**.

If the first line of the file is "**one**", then the first time through the loop what's evaluated is **while "one"**.

The value "**one**" is not **false** or **nil**, so the body of the loop is executed, causing "**one**" to be printed on standard output.

At end of file, **gets** returns **nil**. **nil** is assigned to **line** and produced as the value of the assignment, terminating the loop in turn.

while, continued

String's **chomp** method removes a carriage return and/or newline from the end of a string, if present. (LHtLaL: Try **"test\n".chomp** in **irb**!)

Here's a program that is intended to flatten input lines to a single line:

```
result = ""
while line = gets.chomp
    result += line
end
puts result
```

It doesn't work. What's wrong with it?

Here's the error:

```
% ruby while4.rb < lines.txt
while4.rb:2:in `<main>': undefined method `chomp' for
nil:NilClass (NoMethodError)
```

Problem: Write a **while** loop that prints the characters in the string **s**, one per line. Don't use the **length** or **size** methods of **String**.

Extra credit: Don't use any variables other than **s**.

Solution: (**while5.rb**)

```
i = 0
while c = s[i]
   puts c
   i += 1
end
```

Solution with only **s**: (**while5a.rb**)

```
while s[0]
   puts s[0]
   s[0] = ""
end
```

Source code layout

Unlike Java, Ruby does pay some attention to the presence of newlines in source code.

For example, a while loop cannot be simply written on a single line.

```
while i <= 10 puts i i += 1 end          # Syntax error
```

If we add semicolons where newlines originally were, it works:

```
while i <= 10; puts i; i += 1; end       # OK
```

There is some middle ground, too:

```
while i <= 10 do puts i; i+=1 end        # OK.  Note added "do"
```

Unlike Haskell and Python, <u>indentation is never significant in Ruby</u>.

Source code layout, continued

Ruby considers a newline to terminate an expression, unless the expression is definitely incomplete.

For example, the following is ok because "**i <=**" is definitely incomplete.

```
while i <=
10 do puts i; i += 1 end
```

Is the following ok?

```
while i
<= 10 do puts i; i += 1 end
```

Nope...

```
syntax error, unexpected tLEQ
<= 10 do puts i; i += 1 end
^
```

Source code layout, continued

Can you think of any pitfalls that the incomplete expression rule could produce?

Example of a pitfall: Ruby considers

```
x = a + b
    + c
```

to be two expressions: `x = a + b` and `+ c`.

Rule of thumb: If breaking an expression across lines, end lines with an operator:

```
x = a + b +
    c
```

Alternative: Indicate continuation with a backslash at the end of the line.

Expression or statement?

Academic writing on programming languages commonly uses the term "statement" to denote a syntactic element that performs operation(s) but does not produce a value.

The term "expression" is consistently used to describe a construct that produces a value.

Ruby literature sometimes talks about the "while statement" even though **while** produces a value:

```
>> i = 1
>> while i <= 3 do i += 1 end
=> nil
```

Dilemma: Should we call it the "while statement" or the "while expression"?

We'll see later that the **break** construct can cause a while loop to produce a value other than **nil**.

Logical operators

Ruby has operators for conjunction, disjunction, and "not" with the same symbols as Java and C, but with somewhat different semantics.

Conjunction is **&&**, just like Java, but note the values produced:

```
>> true && false
=> false

>> 1 && 2
=> 2

>> true && "abc"
=> "abc"

>> nil && 1
=> nil
```

Remember:
Any value that is not **false** or **nil** is considered to be "true".

Challenge: Concisely describe the rule that Ruby uses to determine the value of a conjunction operation.

Logical operators, continued

Disjunction is ||, also like Java. As with conjunction, the values produced are interesting:

```
>> 1 || nil
=> 1

>> false || 2
=> 2

>> "abc" || "xyz"
=> "abc"

>> s = "abc"
>> s[0] || s[3]
=> "a"

>> s[4] || false
=> false
```

> Remember:
> Any value that is not **false** or **nil** is considered to be "true".

Logical operators, continued

An exclamation mark inverts a logical value. The resulting value is <u>always</u> **true** or **false**.

```
>> ! true
=> false

>> ! 1
=> false

>> ! nil
=> true

>> ! (1 || 2)
=> false

>> ! ("abc"[5] || [1,2,3][10])
=> true

>> ![nil]
=> false
```

Remember:
Any value that is not **false** or **nil** is considered to be "true".

Logical operators, continued

There are also **and**, **or**, and **not** operators, but with very low precedence.

Why?

They eliminate the need for parentheses in some cases.

We can write this,

```
x < 2 && y > 3 or x * y < 10 || z > 20
```

instead of this:

```
(x < 2 && y > 3) || (x * y < 10 || z > 20)
```

LHtLaL problem: Devise an example for **!** vs. **not**.

Sidebar: Parallel assignment

Ruby supports *parallel assignment*. Some simple examples:

```
>> a, b = 10, [20, 30]

>> a
=> 10

>> b
=> [20, 30]

>> c, d = b
>> c
=> 20

>> d
=> 30
```

```
>> x = 10
>> y = 20

>> x,y = y,x

>> x
=> 20

>> y
=> 10
```

Section 4.5.5 in RPL has full details on parallel assignment. It is both <u>more</u> complicated and <u>less</u> general than pattern matching in Haskell. (!)

The **if-then-else** construct

Here is Ruby's **if-then-else**:

```
>> if 1 < 2 then "three" else [4] end
=> "three"

>> if 10 < 2 then "three" else [4] end
=> [4]

>> if 0 then "three" else [4] end * 3
=> "threethreethree"
```

Observations?

Speculate: Is the following valid? If so, what will it produce?

```
if 1 > 2 then 3 end
```

if-then-else, continued

If a language's **if-then-else** returns a value, it creates an issue about the meaning of an **if-then** with no **else**.

In Ruby, if there's no **else** clause and the control expression is **false**, **nil** is produced:

```
>> if 1 > 2 then 3 end
=> nil
```

In the C family, **if-then-else** doesn't return a value.

Haskell and ML simply don't allow an **else**-less **if**.

In Icon, an expression like **if 2 > 3 then 4** is said to *fail*. No value is produced, and failure propagates to any enclosing expression, which in turn fails.

Ruby also provides **1 > 2 ? 3 : 4**, a ternary conditional operator, just like the C family. Is that a good thing or bad thing? (TMTOWTDI!)

if-then-else, continued

The most common Ruby coding style puts the **if**, the **else**, the **end**, and the expressions of the clauses on separate lines:

```
if lower <= x && x <= higher or inExRange(x, rangeList) then
    puts "x is in range"
    history.add x
else
    outliers.add x
end
```

Note the use of the low-precedence **or** instead of ||.

The trailing **then** above is optional.

then is not optional in this case:

```
if 1 then 2 else 3 end
```

The **elsif** clause

Ruby provides an **elsif** clause for "else-if" situations.

```
if average >= 90 then
   grade = "A"
elsif average >= 80 then
   grade = "B"
elsif average >= 70 then
   grade = "C"
else
   grade = "F"
end
```

Note that there is no "**end**" to terminate the **then** clauses. **elsif** both closes the current **then** and starts a new clause.

It is not required to have a final **else**.

Is **elsif** syntactic sugar?

elsif, continued

At hand:

```
if average >= 90 then
   grade = "A"
elsif average >= 80 then
   grade = "B"
elsif average >= 70 then
   grade = "C"
else
   grade = "F"
end
```

```
grade =
   if average >= 90 then "A"
   elsif average >= 80 then "B"
   elsif average >= 70 then "C"
   else "F"
   end
```

Can we shorten it by thinking less imperatively and more about values?

See 5.1.4 in RPL for Ruby's **case** (a.k.a. switch) expression.

if and **unless** as *modifiers*

if and **unless** can be used as *modifiers* to indicate conditional execution.

```
>> total, count = 123.4, 5        # Note: parallel assignment

>> printf("average = %g\n", total / count) if count != 0
average = 24.68
=> nil

>> total, count = 123.4, 0
>> printf("average = %g\n", total / count) unless count == 0
=> nil
```

The general forms are:

expr1 if *expr2*
expr1 unless *expr2*

What does '**x.f if x**' mean?

break and **next**

Ruby's **break** and **next** are similar to Java's **break** and **continue**.

Below is a loop that reads lines from standard input, terminating on end of file or when a line beginning with a period is read. Each line is printed unless the line begins with a pound sign.

```
while line = gets
    if line[0] == "." then
        break
    end
    if line[0] == "#" then
        next
    end
    puts line
end
```

```
while line = gets
    break if line[0] == "."
    next if line[0] == "#"
    puts line
end
```

Problem: Rewrite it to use **if** as a modifier.

break and **next**, continued

SKIP!

Remember that **while** is an expression that produces the value **nil** when the loop terminates.

If a while loop is exited with **break** *expr*, the value of **expr** is the value of the **while**.

Here's a contrived example to show the mechanics of it:

```
% cat break2.rb
s = "x"
puts (while true do
          break s if s.size > 30
          s += s
        end)

% ruby break2.rb
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

The **for** loop

Here are three examples of Ruby's **for** loop:

```
for i in 1..100 do          # as with while, the do is optional
   sum += i
end

for i in [10,20,30]
   sum += i
end

for msymbol in "x".methods
   puts msymbol if msymbol.to_s.include? "!"
end
```

The "in" expression must be an object that has an **each** method.

In the first case, the "in" expression is a **Range**. In the latter two it is an **Array**.

The **for** loop, continued

The **for** loop supports parallel assignment:

```
for s,n,sep in [["1",5,"-"], ["s",2,"o"], [" <-> ",10,""]]
   puts [s] * n * sep
end
```

Output:

```
1-1-1-1-1
sos
 <->  <->  <->  <->  <->  <->  <->  <->  <->  <-> 
```

Consider the feature of supporting parallel assignment in the **for**.

- How would we write the above without it?
- What's the mental footprint of this feature?
- What's the big deal since there's already parallel assignment?
- Is this creeping featurism?
- Might this be used to have array values used as method parameters?

TAKE A BREAK!
(No quiz...)

Method definition

Method definition

Here is a simple Ruby method:

```
def add x, y
  return x + y
end
```

The keyword **def** indicates that a method definition follows. Next is the method name. The parameter list follows, optionally enclosed in parentheses. <u>No types can be specified</u>.

If the end of a method is reached without encountering a **return**, the value of the last expression becomes the return value. Here is a more idiomatic definition for **add**:

```
def add x, y
  x + y
end
```

Method definition, continued

As we saw in an early example, if no arguments are required, the parameter list can be omitted:

```
def hello
    puts "Hello, world!"
end
```

What does **hello** return?

Loading code with **irb**

The **-r** option of **irb** specifies a file to load.

```
% irb -r simple    (assumes .rb suffix)
>> hello
Hello, world!

>> add 3,4
=> 7
```

On lectura use:
% irb –r ./simple

Speculate: What does the **rl** (RL) method do?

```
>> rl
=> true
```

See @110 on Piazza for more re **irb –r.**

```
% cat simple.rb
def add x, y
  x + y
end

def hello
  puts "Hello, world!"
end

def rl
    load __FILE__
end
```

The **rl** method approximates **:reload** in **ghci**!

Lesson: I like using "**it**" in **ghci**, so I added that to Ruby via my **.irbrc**. I like **:r(eload)** in **ghci** so I've created a simple-minded analog, **rl**.

Loading methods into **irb**, continued

Alternatively, we can type a definition directly into **irb**.

We'll use **irb** to bypass my **irb** alias and show the default **irb** prompt.

```
% \irb
irb(main):001:0> def add x, y
irb(main):002:1>   x + y
irb(main):003:1> end
=> nil
irb(main):004:0> add 3, 4
=> 7
```

Note that the default prompt includes a line counter and a nesting depth.

If **add** is a method, where's the class?

I claim to be defining a method named **add** but there's no class in sight!

<u>Methods can be added to a class at run-time in Ruby!</u>

A freestanding method defined in **irb** or found in a file is associated with an object referred to as "**main**", an instance of **Object**.

At the top level, the name **self** references that object.

```
>> [self.class, self.to_s]          => [Object, "main"]

>> methods_b4 = self.methods

>> def add x,y; x+y; end      # Note one-liner, w/semicolons

>> self.methods - methods_b4   => [:add]
```

We can see that <u>**self** has one more method after **add** is defined</u>.

Default values for arguments

Ruby allows default values to be specified for arguments:

```
def wrap s, wrapper = "()"               # wrap3.rb
   wrapper[0] + s + wrapper[-1]          # Why -1?
end

>> wrap "abc", "<>"
=> "<abc>"

>> wrap "abc"
=> "(abc)"

>> wrap it, "|"
=> "|(abc)|"
```

Lots of library methods use default arguments.

```
>> "a-b c-d".split           => ["a-b", "c-d"]
>> "a-b c-d".split "-"       => ["a", "b c", "d"]
```

Methods can't be overloaded!

Ruby does not allow the methods of a class to be overloaded. Here's a Java-like approach that does not work:

```
def wrap s
   wrap(s, "()")
end

def wrap s, wrapper
   wrapper[0] + s + wrapper[-1]
end
```

The imagined behavior is that if **wrap** is called with one argument it will call the two-argument **wrap** with **"()"** as a second argument. In fact, the second definition of **wrap** simply replaces the first. (Last **def** wins!)

```
>> wrap "x"
ArgumentError: wrong number of arguments (1 for 2)

>> wrap("testing", "[ ]")        => "[testing]"
```

Sidebar: A study in contrast

Different languages approach overloading and default arguments in various ways. Here's a sampling:

Java	Overloading; no default arguments
Ruby	No overloading; default arguments
C++	Overloading <u>and</u> default arguments
Icon	No overloading; no default arguments; use an idiom

How does the mental footprint of the four approaches vary? What's the impact on the language's written specification?

Here is **wrap** in Icon:

```
procedure wrap(s, wrapper)
   /wrapper := "()" # if wrapper is &null, assign "()" to wrapper
   return wrapper[1] || s || wrapper[2]
end
```

Arbitrary number of arguments

Java's **String.format** and C's **printf** can accept any number of arguments.

This Ruby method accepts any number of arguments and prints them:

```
def showargs(*args)
   puts "#{args.size} arguments"
   for i in 0...args.size do        # Recall a...b is a to b-1
      puts "##{i}: #{args[i]}"
   end
end
```

The rule: <u>If a parameter is prefixed with an asterisk, an array is made of all following arguments.</u>

```
>> showargs(1, "two", 3.0)
3 arguments:
#0: 1
#1: two
#2: 3.0
```

Arbitrary number of arguments, continued

Problem: Write a method **format** that interpolates argument values into a string where percent signs are found.

```
>> format("x = %, y = %, z = %\n", 7, "ten", "zoo")
=> "x = 7, y = ten, z = zoo\n"

>> format "testing\n"
=> "testing\n"
```

Use **to_s** for conversion to **String.**

A common term for this sort of facility is "varargs"--variable number of arguments.

```
def format(fmt, *args)
    result = ""
    for i in 0...fmt.size do
        if fmt[i] == "%" then
            result += args.shift.to_s
        else
            result += fmt[i]
        end
    end
    result
end
```

Source File Layout

Here's an example of source file layout for a program with several methods:

```
def main
   puts "in main"; f; g
end

def f; puts "in f" end
def g; puts "in g" end

main  # This runs the program
```

Execution:

```
% ruby main1.rb
in main
in f
in g
```

A rule: the definition for a method must be seen before it is executed.

The definitions for **f** and **g** can follow the definition of **main** because they aren't executed until **main** is executed.

Could the line "**main**" appear before the definition of **f**?

Try shuffling the three definitions and "**main**" to see what works and what doesn't.

Global variables

Ordinary variables are local to the method in which they're created.

Example:

```
def f                      # global0.rb
  puts "f: x = #{x}"       # undefined local variable or method `x'
end

def g
  x = 100                  # This x is visible only in g
end

x = 10                     # This x is visible only at the top-level
f                          # in this file.
g

puts "top-level: x = #{x}"
```

Global variables, continued

<u>Variables prefixed with a $ are global</u>, and can be referenced in any method in any file, including top-level code.

```
def f
   puts "f: $x = #{$x}"
end

def g
   $x = 100
end

$x = 10
f
g

puts "top-level: $x = #{$x}"
```

The code at left...

1. Sets **$x** at the top-level.
2. Accesses **$x** in **f**.
3. Changes **$x** in **g**.
4. Prints the final value of **$x** at the top-level.

Output:

```
f: $x = 10
top-level: $x = 100
```

Constants

A rule in Ruby is that if an identifier begins with a capital letter, it represents a *constant*.

The first assignment to a constant is considered initialization.

```
>> MAX_ITEMS = 100
```

Assigning to an already initialized constant is permitted but a warning is generated.

```
>> MAX_ITEMS = 200
(irb):4: warning: already initialized constant MAX_ITEMS
=> 200
```

Modifying an object referenced by a constant does not produce a warning:

```
>> L = [10,20]
=> [10, 20]

>> L.push 30
=> [10, 20, 30]
```

Constants, continued

If a method is given a name that begins with a capital letter, it compiles ok but it can't be run!

```
>> def Hello; puts "hello!" end

>> Hello
NameError: uninitialized constant Hello
```

Constants, continued

There are a number of predefined constants. Here are a few:

`RUBY_VERSION`
The version of Ruby that's running.

`ARGV`
An array holding the command line arguments, like the argument to `main` in a Java program.

`ENV`
An object holding the "environment variables" (shown with `env` on UNIX machines and `set` on Windows machines.)

`STDIN`, `STDOUT`
Instances of the `IO` class representing standard input and standard output (the keyboard and screen, by default).

Duck Typing

Duck typing

Recall these examples of the **for** loop:

```
for i in 1..100 do
   sum += i
end

for i in [10,20,30] do
   sum += i
end
```

It is only required that the "**in**" value be an object that has an **each** method. (It doesn't need to be a subclass of **Enumerable**, for example.)

This is an example of *duck typing*, so named based on the "duck test":

If it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck.

For the case at hand, the value produced by the "**in**" expression qualifies as a "duck" if it has an **each** method.

Duck typing, continued

The key characteristic of duck typing is that we only care about whether an object supports the operation(s) we require.

With Ruby's **for** loop, it is only required that the **in** value have an **each** method.

Consider this method:

```
def double x
   return x * 2
end
```

What operation(s) must **x** support?

Important: `x * 2` actually means `x.*(2)` — invoke the method `*` on the object `x` and pass it the value `2` as a parameter.

Duck typing, continued

```
>> double 10
=> 20

>> double "abc"
=> "abcabc"

>> double [1,2,3]
=> [1, 2, 3, 1, 2, 3]

>> double Rational(3)
=> (6/1)

>> double 1..10
NoMethodError: undefined method `*' for 1..10:Range
```

```
def double x
  return x * 2
end
```

Is it good or bad that **double** operates on so many different types?

Is **double** polymorphic? What's the type of **double**?

Should we limit **double** to certain types, like numbers, strings and lists?

Duck typing, continued

Recall: The key characteristic of duck typing is that we only care about whether an object supports the operation(s) we require.

Should we have **double** check for known types?

```
def double x
   if [Fixnum, Float, String, Array].include? x.class
      return x * 2
   else raise "Can't double a #{x.class}!" end
end
```

```
>> double "abc"
=> "abcabc"

>> double 1..2
RuntimeError: Can't double a Range!

>> double Rational(3)
RuntimeError: Can't double a Rational!
```

Previously...

```
>> double 1..10
NoMethodError: undefined
method `*' for 1..10:Range
```

Duck typing, continued

Recall: <u>The key characteristic of duck typing is that we only care about whether an object supports the operation(s) we require.</u>

Does this Java method exemplify duck typing?

```
static double sumOfAreas(Shape shapes[]) {
    double area = 0.0;
    for (Shape s: shapes)
        area += s.getArea();
    return area;
    }
```

No! sumOfAreas requires an array of Shape instances.

Could we change Shape to Object above? Would that be duck typing?

Does duck typing require a language to be dynamically typed?

Iterators and blocks

Iterators and blocks

Some methods are *iterators*. An iterator that is implemented by the **Array** class is **each**.

each iterates over the elements of the array. Example:

```
>> x = [10,20,30]

>> x.each { puts "element" }
element
element
element
=> [10, 20, 30]   # (each returns its receiver but it's often not used)
```

An iterator is a method that can invoke a block.

The construct **{ puts "element" }** is a *block*.

Array#each invokes the block once for each element of the array.

Because there are three values in **x**, the block is invoked three times, printing "**element**" each time.

Iterators and blocks, continued

Recall: An iterator is a method that can invoke a block.

Iterators can pass one or more values to a block as arguments.

A block can access arguments by naming them with a parameter list, a comma-separated sequence of identifiers enclosed in vertical bars.

```
>> [10, "twenty", [30,40]].each { |e| puts "element: #{e}" }
element: 10
element: twenty
element: [30, 40]
=> [10, "twenty", [30, 40]]
```

The behavior of the iterator **Array#each** is to invoke the block with each array element in turn.

Iterators and blocks, continued

For reference:

```
[10, "twenty", [30,40]].each { |e| puts "element: #{e}" }
```

Problem: Using a block, compute the sum of the numbers in an array containing values of any type. (Use **e.is_a? Numeric** to decide whether **e** is a number of some sort.)

```
>> sum = 0
>> [10, "twenty", 30].each {  ??? }

>> sum            => 40
```

Note: **sum = ...** inside block changes it outside the block. (Rules coming soon!)

```
>> sum = 0
>> (1..100).to_a.each { |e| sum += e if e.is_a? Numeric  }
>> sum            => 5050
```

Sidebar: Iterate with **each** or use a **for** loop?

Recall that the **for** loop requires the value of the "**in**" expression to have an **each** method.

That leads to a choice between a **for** loop,

```
for name in "x".methods do
    puts name if name.to_s.include? "!"
end
```

and iteration with **each**,

```
"x".methods.each {|name| puts name if name.to_s.include? "!" }
```

Which is better?

Iterators and blocks, continued

Array#each is typically used to create side effects of interest, like printing values or changing variables.

In contrast, with some iterators it is the value returned by an iterator that is of principle interest.

See if you can describe what the following iterators are doing.

```
>> [10, "twenty", 30].collect { |v| v * 2 }
=> [20, "twentytwenty", 60]

>> [[1,2], "a", [3], "four"].select { |v| v.size == 1 }
=> ["a", [3]]
```

What do those remind you of?

Iterators and blocks, continued

The block for **Array#sort** takes two arguments.

```
>> [30, 20, 10, 40].sort { |a,b| a <=> b}
=> [10, 20, 30, 40]
```

Speculate: what are the arguments being passed to **sort**'s block? How could we find out?

```
>> [30, 20, 10, 40].sort { |a,b| puts "call: #{a} #{b}"; a <=> b}
call: 30 10
call: 10 40
call: 30 40
call: 20 30
call: 10 20
=> [10, 20, 30, 40]
```

How could we reverse the order of the **sort**?

Iterators and blocks, continued

Problem: sort the words in a sentence by descending length.

```
>> "a longer try first".split.sort { |a,b| b.size <=> a.size  }
=> ["longer", "first", "try", "a"]
```

What do the following examples remind you of?

```
>> [10, 20, 30].inject(0) { |sum, i| sum + i }
=> 60

>> [10,20,30].inject([]) {
        |memo, element| memo << element << "---" }
=> [10, "---", 20, "---", 30, "---"]
```

Iterators in Enumerable

We can query the "ancestors" of a class like this:

```
>> Array.ancestors
=> [Array, Enumerable, Object, Kernel, BasicObject]
```

For now we'll simply say that an object can call methods in its ancestors.

Enumerable has a number of iterators. Here are some:

```
>> [2,4,5].any? { |n| n.odd? }
=> true

>> [2,4,5].all? { |n| n.odd? }
=> false

>> [1,10,17,25].find { |n| n % 5 == 0 }
=> 10
```

Iterators in **Enumerable**

At hand:

A object can call methods in its **ancestors**. An ancestor of **Array** is **Enumerable**.

Another **Enumerable** method is **max**:

```
>> ["apple", "banana", "grape"].max {
                    |a,b| v = "aeiou"
                    a.count(v) <=> b.count(v)
                    }
=> "banana"
```

The methods in **Enumerable** use duck typing. They require only an **each** method except for **min**, **max**, and **sort**, which also require **<=>**.

See **http://ruby-doc.org/core-1.9.3/Enumerable.html**

Iterators abound!

Recall: <u>An iterator is a method that can invoke a block.</u>

Many classes have one or more iterators. One way to find them is to search their **ruby-doc.org** page for "block".

www.ruby-doc.org/core-1.9.3/Integer.html

times {|i| block } → self

times → an_enumerator

Iterates block *int* times, passing in values from zero to *int* - 1.

If no block is given, an enumerator is returned instead.

What will **3.times { |n| puts n }** do?

```
>> 3.times { |n| puts n }
0
1
2
=> 3
```

A few more iterators

Three more examples:

```
>> "abc".each { |c| puts c }
NoMethodError: undefined method `each' for "abc":String

>> "abc".each_char { |c| puts c }
a
b
c
=> "abc"

>> i = 0
>> "Mississippi".gsub("i") { (i += 1).to_s }
=> "M1ss2ss3pp4"
```

The "do" syntax for blocks

An alternative to enclosing a block in braces is to use **do**/**end**:

```
a.each do
    |element|
    print "element: #{element}\n"
end
```

Common style is to use brackets for one-line blocks, like previous examples, and **do**...**end** for multi-line blocks.

The opening brace or **do** for a block must be on the same line as the iterator invocation. Here's an error:

```
a.each
    do    # syntax error, unexpected keyword_do_block,
          #   expecting $end
    |element|
    print "element: #{element}\n"
end
```

Nested blocks

sumnums.rb reads lines from standard input, assumes the lines consist of integers separated by spaces, and prints their total, count, and average.

```
total = n = 0
STDIN.readlines().each do
   |line|
   line.split(" ").each do
      |word|
      total += word.to_i
      n += 1
   end
end
printf("total = %d, n = %d, average = %g\n",
     total, n, total / n.to_f) if n != 0
```

```
% cat nums.dat
5 10 0 50

 200
1 2 3 4 5 6 7 8 9 10
% ruby sumnums.rb < nums.dat
total = 320, n = 15, average = 21.3333
```

STDIN represents standard input. It is an instance of the **IO** class.

STDIN.readlines reads/returns all of standard input as an array of lines.

The **printf** format specifier **%g** indicates to format a floating point number and select the better of fixed point or exponential form based on the value.

Scoping issues with blocks

<u>Blocks raise issues with the scope of variables.</u>

If a variable exists outside of a block, references to that variable in a block refer to that existing variable. Example:

```
>> sum = 0
```

Note: sum will accumulate across two iterator calls

```
>> [10,20,30].each {|x| sum += x}

>> sum
=> 60

>> [10,20,30].each {|x| sum += x}

>> sum
=> 120
```

Scoping issues with blocks, continued

If a variable is created in a block, the scope of the variable is limited to the block.

In the example below we confirm that **x** exists only in the block, and that the block's parameter, **e**, is local to the block.

```
>> e = "eee"
>> x
NameError: undefined local variable or method `x' ...

>> [10,20,30].each {|e| x = e * 2; puts x}
20
...
>> x
NameError: undefined local variable or method `x' ...
>> e
=> "eee"        # e's value was not changed by the block
```

LHtLaL

Scoping issues with blocks, continued

Pitfall: If we write a block that references a currently unused variable but later add a use for that variable outside the block, we might get a surprise.

Version 1:

```
a.each do |x|
    result = ... # first use of result in this method
    ...
end
```

Version 2:

```
result = ...       # new first use of result in this method
...
a.each do |x|
    ...
    result = ... # references/clobbers result in outer scope
end
...
...use result...   # uses value of result set in block.  Surprise!
```

Scoping issues with blocks, continued

We can make variable(s) local to a block by adding them at the end of the block's parameter list, preceded by a semicolon.

```
result = ...
...
a.each do
    |x; result, tmp|
      result = ... # result is local to block
      ...
end

...
...use result...   # uses result created outside of block
```

Writing iterators

A simple iterator

Key Slide!

Recall: An iterator is a method that can invoke a block.

The **yield** expression invokes the block associated with the current method invocation. Arguments of **yield** become parameters of the block.

Here is a simple iterator that yields two values, a 3 and a 7:

```
def simple
    puts "simple: Starting..."
    yield 3
    puts "simple: Continuing..."
    yield 7
    puts "simple: Done..."
    "simple result"
end
```

Usage:

```
>> simple  {|x|puts "\tx = #{x}" }
simple: Starting...
        x = 3
simple: Continuing...
        x = 7
simple: Done...
=> "simple result"
```

The **puts** in **simple** are used to show when **simple** is active. Note the interleaving of execution between the iterator and the block.

A simple iterator, continued

At hand:

```
def simple
    puts "simple: Starting..."
    yield 3
    puts "simple: Continuing..."
    yield 7
    puts "simple: Done..."
    "simple result"
end
```

Usage:

```
>> simple { |x| puts "\tx = #{x}" }
simple: Starting...
        x = 3
simple: Continuing...
        x = 7
simple: Done...
=> "simple result"
```

<u>There's no formal parameter that corresponds to a block.</u> The block, if any, is implicitly referenced by **yield**.

The parameter of **yield** becomes the named parameter for the block.

Calling **simple** without a block produces an error on the first **yield**:

```
>> simple
simple: Starting...
LocalJumpError: no block given (yield)
```

Write **from_to**

Problem: Write an iterator **from_to(f, t, by)** that yields the integers from **f** through **t** in steps of **by**, which defaults to 1. Assume **f <= n**.

```
>> from_to(1,3) { |i| puts i }
1
2
3
=> 3

>> from_to(0,99,25) { |i| puts i }
0
25
50
75
=> 4
```

Parameters are passed to the iterator (the method) just like any other method.

from_to, continued

Solution:

```
def from_to(from, to, by = 1)
    n = from
    results = 0
    while n <= to do
        yield n
        n += by
        results += 1
    end
    results
end
```

Desired:

```
>> from_to(1,10,2) { |i| puts i }
1
3
5
7
9
=> 5
```

Another test:

```
>> from_to(-5,5,1) { |i| print i, " " }
-5 -4 -3 -2 -1 0 1 2 3 4 5 => 11
```

yield, continued

To pass multiple arguments for a block, specify multiple arguments for **yield**.

Imagine an iterator that produces overlapping pairs from an array:

```
>> elem_pairs([3,1,5,9]) { |x,y| print "x = #{x}, y = #{y}\n" }
x = 3, y = 1
x = 1, y = 5
x = 5, y = 9
```

Implementation:

```
def elem_pairs(a)
   for i in 0...(a.length-1)
      yield a[i], a[i+1]              # yield(a[i], a[i+1]) is ok, too
   end
end
```

Speculate: What will be the result with **yield [a[i], a[i+1]]**? (Extra **[...]**)

A round-trip with **yield**

<u>When **yield** passes a value to a block the result of the block becomes the value of the **yield** expression.</u>

Here is a trivial iterator to show the mechanics:

```
def round_trip x
   r = yield x
   "yielded #{x} and got back #{r}"
end
```

Usage:

```
>> round_trip(3) {|x| x * 5 }       # parens around 3 are required!
=> "yielded 3 and got back 15"

>> round_trip("testing") {|x| x.size }
=> "yielded testing and got back 7"
```

A round-trip with **yield**, continued

At hand:

```
def round_trip x
    r = yield x
    "yielded #{x} and got back #{r}"
end
```

Key Slide!

```
>> round_trip(3) {|x| x * 5 }
=> "yielded 3 and got back 15"
```

1. Iterator yields `3` to block. **`x`** becomes `3`.

```
r = yield 3                {|x| x * 5 }
```

2. Block returns `15`, which becomes value of **yield** `3`.

3. Value of **yield** 3 is assigned to **`r`**.

Round trips with **yield**

Consider this iterator:

```
>> select([[1,2], "a", [3], "four"]) { |v| v.size == 1 }
=> ["a", [3]]

>> select("testing this here".split) { |w| w.include? "e" }
=> ["testing", "here"]
```

What does it appear to be doing?

Producing the elements in its argument, an array, for which the block produces true.

Problem: Write it!

Round trips with **yield**, continued

At hand:

```
>> select([[1,2], "a", [3], "four"]) { |v| v.size == 1 }
=> ["a", [3]]
```

Solution:

```
def select array
  result = [ ]

  for element in array
    if yield element then
      result << element
    end
  end

  result
end
```

What does the iterator/block interaction look like?

Iterator	Block
if yield [1,2] then ~~result << [1,2]~~	# [1,2].size == 1
if yield "a" then result << "a"	# "a".size == 1
if yield [3] then result << [3]	# [3] .size == 1
if yield "four" then ~~result << "four"~~	# "four".size == 1

Round trips with **yield**, continued

Is **select** limited to arrays?

```
>> select(1..10) {|n| n.odd? && n > 5 }
=> [7, 9]
```

Why does that work?

Because **for var in x** works for any **x** that has an **each** method. (Duck typing!)

What's a better name than **array** for **select**'s parameter?

Problem: Rewrite select to use the iterator **each** instead of a for loop. Also use an **if** modifier with the **yield**.

```
def select array
  result = [ ]
  for element in array
    if yield element then
      result << element
    end
  end

  result
end
```

Round trips with **yield**, continued

Solution:

```
def select eachable
    result = []
    eachable.each do
        |element|
        result << element if yield element
    end
    result
end
```

```
def select array
    result = [ ]
    for element in array
        if yield element then
            result << element
        end
    end

    result
end
```

What's the difference between our **select**,

```
select([[1,2], "a", [3], "four"]) { |v| v.size == 1 }
```

And Ruby's **Array#select**?

```
[[1,2], "a", [3], "four"].select { |v| v.size == 1 }
```

Ruby's **Array#select** is a method of **Array**. Our **select** is added to the object "**main**". (See slide 112.)

Sidebar: Ruby vs. Haskell

```
def select array
  result = [ ]
  for element in array
    if yield element then
      result << element
    end
  end

  result
end
```

```
select _ [] = []
select f (x:xs)
    | f x = x : select f xs
    | otherwise = select f xs


> select (\x -> length x == 4) ["just","a", "test"]
["just","test"]
```

```
>> select(["just","a", "test"]) { |x| x.size == 4 }
=> ["just", "test"]
```

Which is better?

Various types of iteration side-by-side

```
>> [10, "twenty", [30,40]].each { |e| puts "element: #{e}" }
>> sum = 0; [1,2,3].each { |x| sum += x }
```

Invokes block with each element in turn for side-effect(s). Result of **each** uninteresting.

```
>> [10,20,30].map { |x| x * 2 }   => [20, 40, 60]
```

Invokes block with each element in turn and returns array of block results.

```
>> [2,4,5].all? { |n| n.odd? }      => false
```

Invokes block with each element in turn; each block result contributes to final result of **true** or **false**, possibly short-circuiting.

```
>> [[1,2], "a", [3], "four"].select { |v| v.size == 1 } => ["a", [3]]
```

Invokes block to determine membership in final result.

```
>> "try this first".split.sort {|a,b| b.size <=> a.size }  => [...]
```

Invokes block an arbitrary number of times; each block result guides further computation towards final result.

The **Hash** class

Symbols

An identifier preceded by a colon creates a **Symbol** instance.

A symbol is much like a string but a given identifier always produces the same **Symbol** object.

```
>> s1 = :length              => :length
>> s1.object_id              => 7848
>> :length.object_id         => 7848    # same as s1.object_id!
```

In contrast, two identical string literals produce two different String objects:

```
>> "length".object_id        => 2164862320
>> "length".object_id        => 2164856820
```

Symbols can be quickly compared and are immutable. They're commonly used as keys in instances of **Hash**.

Symbols, continued

A string can be turned into a symbol with **.to_sym** (an analog to Java's **String.intern**).

```
>> s = "length".to_sym          => :length
>> s.object_id                  => 7848
```

methods returns an array of symbols:

```
>> "x".methods.sort[10,30]
=> [:==, :===, :=~, :>, :>=, :[], :
[]=, :__id__, :__send__, :ascii_only?, :between?, :bytes, :bytesiz
e, :byteslice, :capitalize, :capitalize!, :casecmp, :center, :chars,
:chomp, :chomp!, :chop, :chop!, :chr, :class, :clear, :clone, :cod
epoints, :concat, :count]
```

<u>For our purposes it's sufficient to simply know that *:identifier* is a *symbol*.</u>

Symbols are sometimes used with hashes.

The **Hash** class

Ruby's **Hash** class is similar to the **Map** family in Java and dictionaries in Python. It's like an array that can be subscripted with values of <u>any</u> type.

The expression **{ }** (empty curly braces) creates a **Hash**:

```
>> numbers = {}        => {}

>> numbers.class       => Hash
```

Subscripting with a *key* and assigning a value stores that key/value pair.

```
>> numbers["one"] = 1

>> numbers["two"] = 2

>> numbers
=> {"one"=>1, "two"=>2}

>> numbers.size
=> 2
```

Hash, continued

At hand:

```
>> numbers
=> {"one"=>1, "two"=>2}
```

Subscripting with a key fetches the associated value. If the key is not found, **nil** is produced.

```
>> numbers["two"]
=> 2

>> numbers["three"]
=> nil
```

Hash, continued

At hand:

```
>> numbers   => {"one"=>1, "two"=>2}
```

The **Hash** class has many methods. Here's a sampling:

```
>> numbers.keys
=> ["one", "two"]

>> numbers.values
=> [1, 2]

>> numbers.invert
=> {1=>"one", 2=>"two"}

>> numbers.to_a
=> [["one", 1], ["two", 2]]
```

Some of the many **Hash** iterators: **delete_if**, **each_pair**, **select**

Hash, continued

At hand:

```
>> numbers
=> {"one"=>1, "two"=>2}
```

The value associated with a key can be changed via assignment.

```
>> numbers["two"] = "1 + 1"
```

A key/value pair can be removed with **Hash#delete**.

```
>> numbers.delete("one")
=> 1  # Returns associated value

>> numbers
=> {"two"=>"1 + 1"}

>> numbers["one"]
=> nil
```

Hash, continued

The rules for <u>keys</u> and <u>values</u>:

Key values must have a **hash** method that produces a **Fixnum**. (Duck typing!)

Any value can be the value in a key/value pair.

```
>> h = {};  a = [1,2,3]

>> h[a] = "-"

>> h[String] = ["a","b","c"]

>> h["x".class] * h[(1..3).to_a]
=> "a-b-c"

>> h[h] = h

>> h
=> {[1, 2, 3]=>"-", String=>["a", "b", "c"], {...}=>{...}}
```

Hash, continued

Inconsistencies can arise when using mutable values as keys.

```
>> h = {}; a = []

>> h[a] = "x"

>> h
=> {[]=>"x"}

>> a << 1

>> h
=> {[1]=>"x"}

>> h[a]
=> nil
```

Ruby treats string-valued keys as a special case and makes a copy of them.

Hash, continued

Here's a sequence that shows some of the flexibility of hashes.

```
>> h = {}

>> h[1000] = [1,2]

>> h[true]  = {}

>> h[[1,2,3]] = [4]

>> h
=> {1000=>[1, 2], true=>{}, [1, 2, 3]=>[4]}

>> h[h[1000] + [3]] << 40

>> h[!h[10]]["x"] = "ten"

>> h
=> {1000=>[1, 2], true=>{"x"=>"ten"}, [1, 2, 3]=>[4, 40]}
```

Default values

An earlier simplification: If a key is not found, **nil** is returned.
Full detail: If a key is not found, the *default value* of the hash is returned.

The default value of a hash defaults to **nil** but an arbitrary default value can be specified when creating a hash with **new**:

```
>> h = Hash.new("Go Fish!")      # Example from ruby-doc.org

>> h.default
=> "Go Fish!"

>> h["x"] = [1,2]

>> h["x"]
=> [1, 2]

>> h["y"]
=> "Go Fish!"
```

There is also a form of **Hash#new** that uses a block to produce default values. The block accepts the hash and the key as arguments.

Problem: write **tally.rb**, to tally occurrences of blank-separated "words" on standard input.

```
% ruby tally.rb
to be or
not to be
^D
{"to"=>2, "be"=>2, "or"=>1, "not"=>1}
```

How can we approach it?

tally.rb

Solution:

```
# Use default of zero so += 1 works
counts = Hash.new(0)

readlines.each do
    |line|
    line.split(" ").each do
        |word|
        counts[word] += 1
    end
end

# Like puts counts.inspect
p counts
```

```
% ruby tally.rb
to be or
not to be
^D
{"to"=>2, "be"=>2,
"or"=>1, "not"=>1}
```

```
counts = Hash.new(0)
while line = gets do
    for word in line.split(" ") do
        counts[word] += 1
    end
end
p counts
```

tally.rb, continued

The output of **tally.rb** is not customer-ready!

```
{"to"=>2, "be"=>2, "or"=>1, "not"=>1}
```

Hash#sort produces an array of key/value arrays ordered by the keys, in ascending order:

```
>> counts.sort
=> [["be", 2], ["not", 1], ["or", 1], ["to", 2]]
```

Problem: Produce nicely labeled output, like this:

```
Word     Count
be           2
not          1
or           1
to           2
```

tally.rb, continued

```
Word     Count
be           2
not          1
or           1
to           2
```

At hand:

```
>> counts.sort
[["be", 2], ["not", 1], ["or", 1], ["to", 2]]
```

Solution:

```
([["Word","Count"]] + counts.sort).each do
   |k,v| printf("%-7s %5s\n", k, v)
end
```

Notes:

- The minus in the format **%-7s** <u>left</u>-justifies, in a field of width seven.
- As a shortcut for easy alignment, the column headers are put at the start of the array, <u>as a fake key/value pair</u>.
- We use **%5s** instead of **%5d** to format the counts and accommodate "**Count**", too. This works because **%s** causes **to_s** to be invoked on the value being formatted.)
- A next step might be to size columns based on content.

More on **Hash** sorting

Hash#sort's default behavior of ordering by keys can be overridden by supplying a block. The block is repeatedly invoked with two key/value pairs, like **["be", 2]** and **["or", 1]**.

Here's a block that sorts by descending count: (the second element of the two-element arrays)

```
>> counts.sort { |a,b| b[1] <=> a[1] }
=> [["to", 2], ["be", 2], ["or", 1], ["not", 1]]
```

How we could resolve ties on counts by alphabetic ordering of the words?

```
counts.sort do
    |a,b|
    r = b[1] <=> a[1]
    if r != 0 then r else a[0] <=> b[0] end
end
=> [["be", 2], ["to", 2], ["not", 1], ["or", 1]]
```

Hash initialization

Imagine a hash that maps strings like "**up**" and "**right**" to x and y deltas on a Cartesian plane:

```
moves = {}
moves["up"] =      [0,1]
moves["down"] =  [0,-1]
moves["left"] =     [-1,0]
moves["right"] =   [1,0]
```

<u>Instead of a series of assignments we can use an initialization syntax:</u>

```
moves = {
    "up"     => [0,1],
    "down" => [0,-1],
    "left"    => [-1,0],
    "right"  => [1,0]
    }
```

Hash initialization, continued

Symbols are commonly used instead of strings as hash keys because they're more efficient. Here's the previous hash with symbols, albeit on one line.

```
>> moves =
  { :up => [0,1], :down => [0,-1], :left => [-1,0], :right => [1,0] }
=> {:up=>[0, 1], :down=>[0, -1], :left=>[-1, 0], :right=>[1, 0]}

>> moves[:up] => [0, 1]
```

With symbols as keys, there's an even shorter initializing form, where the colon separates the symbol from the value:

```
>> moves = { up: [0,1], down: [0,-1], left: [-1,0], right: [1,0] }
=> {:up=>[0, 1], :down=>[0, -1], :left=>[-1, 0], :right=>[1, 0]}
```

If symbols are used as keys, be sure to convert strings before lookup.

```
>> s = "up"; moves[s]          => nil   # Key is :up, not "up"

>> moves[s.to_sym]        => [0, 1]
```

Regular Expressions

A little theory

In computer science theory, a *language* is a set of strings. The set may be infinite.

The Chomsky hierarchy of languages looks like this:

Unrestricted languages	("Type 0")
Context-sensitive languages	("Type 1")
Context-free languages	("Type 2")
Regular languages	("Type 3")

Roughly speaking, natural languages are unrestricted languages that can only specified by unrestricted grammars.

Programming languages are usually context-free languages—they can be specified with context-free grammars, which have restrictive rules.

Every Java program is a string in the context-free language that is specified by the Java grammar.

A regular language is a very limited kind of context free language that can be described by a regular grammar. <u>A regular language can also be described by a regular expression</u>.

A little theory, continued

A regular expression is simply a string that may contain *metacharacters*—characters with special meaning.

Here is a simple regular expression:

    a+

It specifies the regular language that consists of the strings {a, aa, aaa, ...}.

Here is another regular expression:

    (ab)+c*

It describes the set of strings that start with ab repeated one or more times and followed by zero or more c's.

Some strings in the language are ab, ababc, and ababababcccccc.

The regular expression (north|south)(east|west) describes a language with four strings: {northeast, northwest, southeast, southwest}.

Good news and bad news

Regular expressions have a sound theoretical basis and are also very practical.

UNIX tools such as the **ed** editor and **grep/fgrep/egrep** introduced regular expressions to a wide audience.

Many languages provide a library for working with regular expressions. Java provides the **java.util.regex** package. The command **man regex** produces some documentation for the C library's regular expression routines.

Some languages, Ruby included, have a regular expression <u>type</u>.

Good news and bad news, continued

Regular expressions as covered in a theory class are relatively simple.

Regular expressions as available in many languages and libraries have been extended far beyond their theoretical basis.

In languages like Ruby, regular expressions are truly a language within a language.

An edition of the "Pickaxe" book devoted four pages to its summary of regular expressions. Four more pages sufficed to cover integers, floating point numbers, strings, ranges, arrays, and hashes.

Entire books have been written on the subject of regular expressions. A number of tools have been developed to help programmers create and maintain complex regular expressions.

Good news and bad news, continued

Here is a regular expression written by Mark Cranness and posted at RegExLib.com:

```
^((?>[a-zA-Z\d!#$%&'*+\-/=?^_`{|}~]+\x20*|"((?=[\x01-\x7f])
[^"\\]|\\[\x01-\x7f])*"\x20*)*(? <angle><))?((?!\.)(?>\.?[a-zA-
Z\d!#$%&'*+\-/=?^_`{|}~]+)+|"((?=[\x01-\x7f])[^"\\]|\\[\x01-\
x7f])*")@(((?!-)[a-zA-Z\d\-]+(?<!-)\.)+[a-zA-Z]{2,}|\[(((?(?<!\[)
\.)(25[0-5]|2[0-4]\d|[01]?\d? \d)){4}|[a-zA-Z\d\-]*[a-zA-Z\d]:
((?=[\x01-\x7f])[^\\\[\]]|\\[\x01-\x7f])+)\])(?(angle)>)$
```

It describes RFC 2822 email addresses.

My opinion: regular expressions are good for simple tasks but grammar-based parsers should be favored as complexity rises, especially when an underlying specification includes a grammar.

We'll cover a subset of Ruby's regular expression capabilities.

A simple regular expression in Ruby

One way to create a regular expression (RE) in Ruby is to use the *`/pattern/`* syntax, for regular expression literals.

```
>> re = /a.b.c/          => /a.b.c/

>> re.class              => Regexp
```

In a RE, a dot is a metacharacter (a character with special meaning) that will match any (one) character.

Letters, numbers, and some special characters simply match themselves.

The RE `/a.b.c/` matches strings that contain the five-character sequence a<*anychar*>b<*anychar*>c, like "albacore", "barbecue", "drawback", and "iambic".

The match operator

The binary operator **=~** is called "match".

One operand must be a string and the other must be a regular expression. If the string contains a match for the RE, the position of the match is returned. **nil** is returned if there is no match.

```
>> "albacore" =~ /a.b.c/      => 0

>> "drawback" =~ /a.b.c/      => 2

>> "abc" =~ /a.b.c/           => nil

>> "abcdef" =~ /..f/
=> 3

>> "abcdef" =~ /.f./
=> nil

>> "abc" =~ /..../
=> nil
```

Regular expressions are "in deep" in Ruby

What's the implication of the following?

```
>> /x/.class    => Regexp
```

Ruby has syntactic support for regular expressions. We can say that regular expressions are *first-class values* in Ruby.

In general there are two levels of support for a type:

- Syntactic support
  - Many languages have syntactic support for strings with "...".
  - Scala and ActionScript have syntactic support for XML.
  - In Icon, '`aeiou`' is a *character set*, not a string.
- Library support
  - Java and Python have classes for working with REs.
  - C and Icon have function libraries for working with REs.

What are the tradeoffs between the two levels?

Example from Icon: '`aeiou`' vs. `cset("aeiou")`

Sidebar: rgrep.rb

The UNIX **grep** command reads standard input or files named as arguments and prints lines that contain a specified regular expression:

```
$ grep g.h.i < /usr/share/dict/words
lengthwise

$ grep l.m.n < /usr/share/dict/words | wc -l
   252    252   2825

$ grep  ....................... < /usr/share/dict/words
electroencephalograph's
```

Problem: Write a simple **grep** in Ruby that will handle the cases above.

Hint: **#{...}** interpolation works in **/.../** (regular expression) literals.

rgrep.rb sidebar, continued

UNIX **grep**:

```
$ grep g.h.i < /usr/share/dict/words
```

Solution:

```
while line = STDIN.gets    # STDIN so "g.h.i" isn't opened for input
   puts line if line =~ /#{ARGV[0]}/
end
```

Usage:

```
$ ruby rgrep.rb g.h.i < /usr/share/dict/words
lengthwise

$ ruby rgrep.rb  ......................  < /usr/share/dict/words
electroencephalograph's
```

The match operator, continued

After a successful match we can use some cryptically named predefined global variables to access parts of the string:

$\$`$ Is the portion of the string that precedes the match. (That's a backquote—ASCII code 96.)

$& Is the portion of the string that was matched by the regular expression.

$' Is the portion of the string following the match.

Example:

```
>> "limit=300" =~ /=/      => 5
>> $`                      => "limit"  (left of the match)
>> $&                      => "="      (the match itself)
>> $'                      => "300"    (right of the match)
```

The match operator, continued

Here's a handy utility routine from the Pickaxe book:

```
def show_match(s, re)
    if s =~ re then
        "#{$`}<<#{$&}>>#{$'}"
    else
        "no match"
    end
end
```

Usage:

```
>> show_match("limit is 300",/is/)
=> "limit <<is>> 300"

>> %w{albacore drawback iambic}.
            each { |w| puts show_match(w, /a.b.c/) }
<<albac>>ore
dr<<awbac>>k
i<<ambic>>
```

LHtLaL

Great idea: Put it in your **.irbrc**! Call it "**sm**", to save some typing!

Character classes

[*characters*] is a *character class*—a RE that matches any one of the characters enclosed by the square brackets.

/[aeiou]/ matches a single lower-case vowel

```
>> show_match("testing", /[aeiou]/)
=> "t<<e>>sting"
```

A dash between two characters in a class specification creates a range based on the collating sequence. [0-9] matches a single digit.

```
>> show_match("Take 5!", /[0-9]/)
=> "Take <<5>>!"

>> show_match("Take five!", /[0-9]/)
=> "no match"
```

Character classes

[^*characters*] is a RE that matches any character not in the class. (It matches the complement of the class.)

/[^0-9]/ matches a single character that is not a digit.

```
>> show_match("1,000", /[^0-9]/)
=> "1<<,>>000"
```

For any RE we can ask,

What is the shortest string the RE can match? What is the longest?

What is the shortest string that [A-Za-z345] can match? The longest?

One for both! [*anything*] <u>always has a one-character match!</u>

Character classes, continued

Describe what's matched by this regular expression:

```
/.[a-z][0-9][a-z]./
```

A five character string whose middle three characters are, in order, a lowercase letter, a digit, and a lowercase letter.

In the following, which portion of the string is matched, if any?

```
>> show_match("A1b33s4ax1", /.[a-z][0-9][a-z]./)
=> "A1b3<<3s4ax>>1"
```

Character classes, continued

String#gsub does global substitution with both plain old strings and regular expressions

```
>> "520-621-6613".gsub("-", "<DASH>")
=> "520<DASH>621<DASH>6613"

>> "520-621-6613".gsub(/[02468]/, "(e#)")
=> "5(e#)(e#)-(e#)(e#)1-(e#)(e#)13"
```

What will result from the following?

```
>> "5-3^2*2.0".gsub(/[-6^.]/, "_")
=> "5_3_2*2_0"
```

The preceding example shows that metacharacters sometimes aren't special when used out of context.

Character classes, continued

Some frequently used character classes can be specified with *C*

`\d`	Stands for [0-9]
`\w`	Stands for [A-Za-z0-9_]
`\s`	Whitespace—blank, tab, carriage return, newline, formfeed

The abbreviations \D, \W, and \S produce a complemented class.

Examples:

```
>> show_match("Call me at 555-1212", /\d\d\d-\d\d\d\d/)
=> "Call me at <<555-1212>>"

>> "fun double(n) = n * 2".gsub(/\w/,".")
=> "... ......(.) = . * ."

>> "CHVEZ 111, 9:30-10:45 TR".gsub(/\D/, "")
=> "1119301045"

>> "buzz93@tv2000.com".gsub(/[\w\d]/,".")
=> "......@........."
```

Alternatives

Alternatives can be specified with a vertical bar:

```
>> show_match("a green box", /red|green|blue/)
=> "a <<green>> box"

>> %w{you ate a pie}.select { |s| s =~ /ea|ou|ie/ }
=> ["you", "pie"]
```

Alternatives and grouping

Parentheses can be used for grouping. Consider this regular expression:

```
/(two|three) (apple|biscuit)s/
```

It corresponds to a regular language that is a set of four strings:

{two apples, three apples, two biscuits, three biscuits}

Usage:

```
>> "I ate two apples." =~ /(two|three) (apple|biscuit)s/
=> 6

>> "She ate three mice." =~ /(two|three) (apple|biscuit)s/
=> nil
```

Another:

```
>> %w{you ate a mouse}.select { |s| s =~ /.(ea|ou|ie)./ }
=> ["mouse"]
```

Simple app: looking for letter patterns

Imagine a program to look through a word list for a pattern of consonants and vowels specified on the command line, showing matches in bars.

```
% ruby convow.rb cvcvcvcvcvcvcvcvc < web2
c|hemicomineralogic|al
|hepatoperitonitis|
o|verimaginativenes|s
```

A capital letter means to match exactly that letter. `e` matches either consonant or vowel.

```
% ruby convow.rb vvvDvvv < web2
Chromat|ioideae|
Rhodobacter|ioideae|

% ruby convow.rb vvvCvvv < web2  | wc -l
24

% ruby convow.rb vvvevvv < web2  | wc -l
43
```

convow.rb

Here's a solution. We loop through the command line argument and build up a regular expression of character classes and literal characters, and then look for lines with a match.

```
re = ""
ARGV[0].each_char do |char|
   re += case char                   # An example of Ruby's case
         when "v" then "[aeiou]"
         when "c" then "[^aeiou]"
         when "e" then "[a-z]"
         else char.downcase
       end
end
puts re
re = /#{re}/         # Transform re from String to Regexp
STDIN.each do
   |line|
   puts [$`, $&, $'] * "|" if line.chomp =~ re
end
```

```
$ ruby convow.rb cvc
[^aeiou][aeiou][^aeiou]

$ ruby convow.rb cEEcc
[^aeiou]ee[^aeiou][^aeiou]
```

There are regular expression operators

A rule we've been using but haven't formally stated is this:

If $\mathtt{R}_1$ and $\mathtt{R}_2$ are regular expressions then $\mathtt{R}_1\mathtt{R}_2$ is a regular expression.
In other words, juxtaposition is the concatenation operation for REs.

There are also postfix operators on regular expressions.

If **R** is a regular expression, then...

R* matches zero or more occurrences of **R**

R+ matches one or more occurrences of **R**

R? matches zero or one occurrences of **R**

All have higher precedence than juxtaposition.

*****, **+**, and **?** are commonly called *quantifiers* but PA doesn't use that term.

The *, +, and ? quantifiers

At hand:

R* matches <u>zero or more</u> occurrences of **R**
R+ matches <u>one or more</u> occurrences of **R**
R? matches <u>zero or one</u> occurrences of **R**

What does the RE **ab*c+d** describe?

An '**a**' that is followed by zero or more '**b**'s that are followed by one or more '**c**'s and then a '**d**'.

```
>> show_match("acd", /ab*c+d/)
=> "<<acd>>"

>> show_match("abcccc", /ab*c+d/)
=> "no match"

>> show_match("abcabccccddd", /ab*c+d/)
=> "abc<<abccccd>>dd"
```

The *, +, and ? quantifiers, continued

At hand:

R* matches zero or more occurrences of R
R+ matches one or more occurrences of R
R? matches zero or one occurrences of R

What does the RE -?\d+ describe?

Integers with any number of digits

```
>> show_match("y is -27 initially", /-?\d+/)
=> "y is <<-27>> initially"

>> show_match("maybe --123.4e-10 works", /-?\d+/)
=> "maybe -<<-123>>.4e-10 works"

>> show_match("maybe --123.4e-10 works", /-?\d*/)  # *, not +
=> "<<>>maybe --123.4e-10 works"
```

The *, +, and ? quantifiers, continued

What does `a(12|21|3)*b` describe?

Matches strings like `ab`, `a3b`, `a312b`, and `a3123213123333b`.

Write an RE to match numbers with commas, like these:

58   4,297   1,000,000   446,744   73,709,551,616

```
(\d\d\d|\d\d|\d)(,\d\d\d)*      # Why is \d\d\d first?
```

Write an RE to match floating point literals, like these:

1.2   .3333e10   -4.567e-30   .0001

```
>> %w{1.2 .3333e10 -4.567e-30 .0001}.
    each {|s| puts show_match(s, /-?\d*\.\d+(e-?\d+)?/) }
<<1.2>>
<<.3333e10>>
<<-4.567e-30>>
<<.0001>>
```

Note the \. to match only a period.

`*`, `+`, and `?` are greedy!

The operators `*`, `+`, and `?` are "greedy"—each tries to match the longest string possible, and cuts back only to make the full expression succeed.

Example:

Given `a.*b` and the input '`abbb`', the first attempt is:

- `a` matches `a`
- `.*` matches `bbb`
- `b` fails—no characters left!

The matching algorithm then *backtracks* and does this:

- `a` matches `a`
- `.*` matches `bb`
- `b` matches `b`

*, +, and ? are greedy, continued

More examples of greedy behavior:

```
>> show_match("xabbbbc", /a.*b/)
=> "x<<abbbb>>c"

>> show_match("xabbbbc", /ab?b?/)
=> "x<<abb>>bbc"

>> show_match("xabbbbcxyzc", /ab?b?.*c/)
=> "x<<abbbbcxyzc>>"
```

Why are *, +, and ? greedy?

Lazy/reluctant quantifiers

In the following we'd like to match just '**abc**' but the greedy asterisk goes too far:

```
show_match("x + 'abc' + 'def' + y", /'.*'/)
=> "x + <<'abc' + 'def'>> + y"
```

We can make ***** lazy by putting **?** after it, causing it to match only as much as needed to make the full expression match. Example:

```
>> show_match("x + 'abc' + 'def' + y", /'.*?'/)
=> "x + <<'abc'>> + 'def' + y"
```

?? and **+?** are supported, too. The three are also called *reluctant quantifiers*.

Once upon a time, before ***?** was supported, one would do this:

```
>> show_match("x + 'abc' + 'def' + y", /'[^']+'/)
=> "x + <<'abc'>> + 'def' + y"
```

Specific numbers of repetitions

We can use curly braces to require a specific number of repetitions:

```
>> show_match("Call me at 555-1212!", /\d{3}-\d{4}/)
=> "Call me at <<555-1212>>!"
```

There are also forms with **{min,max}** and **{min,}**

```
>> show_match("3/17/2013", /\d{1,2}\/\d{1,2}\/(\d{4}|\d{2})/)
=> "<<3/17/2013>>"
```

Note that the RE above has escaped slashes to match the literal slashes.

split and **scan** with regular expressions

We can split a string using a regular expression:

```
>> " one, two,three / four".split(/[\s,\/]+/) # w.s., commas, slashes
=> ["", "one", "two", "three", "four"]
```

Note that leading delimiters produce an empty string in the result.

If we can describe the <u>strings of interest</u> instead of <u>what separates them</u>, **scan** is a better choice:

```
>> " one, two,three / four".scan(/\w+/)
=> ["one", "two", "three", "four"]

>> "10.0/-1.3...5.700+[1.0,2.3]".scan(/-?\d+\.\d+/)
=> ["10.0", "-1.3", "5.700", "1.0", "2.3"]
```

Here's a way to keep all the pieces:

```
>> " one, two,three / four".scan(/\w+|\W+/)
=> [" ", "one", ", ", "two", ",", "three", " / ", "four"]
```

Anchors

Reminder: **`s =~ /x/`** succeeds if "**`x`**" appears anywhere in **`s`**.

The metacharacter **`^`** is an *anchor* when used <u>at the start</u> of a RE. (At the start of a character class it means to complement.)

`^` doesn't match any characters but it constrains the following regular expression to appear at the beginning of the string being matched against.

```
>> show_match("this is x", /x/)          => "this is <<x>>"

>> show_match("this is x", /^x/)         => "no match"

>> show_match("this is x", /^this/)      => "<<this>> is x"
```

What will **`/^x|y/`** match? Hint: it's not the same as **`/^(x|y)/`**

How about **`/^[^0-9]/`** ?

Anchors, continued

Another anchor is **$**. It constrains the preceding regular expression to appear at the end of the string.

```
>> show_match("ending", /end$/)
=> "no match"
>> show_match("the end", /end$/)
=> "the <<end>>"
```

Write a RE to match lines with only a curly brace and (maybe) whitespace.

```
>> show_match("  }   ", /^\s*[{}]\s*$/)
=> "<<  }   >>"
>> show_match("}", /^\s*[{}]\s*$/)
=> "<<}>>"
>> show_match("{ e }", /^\s*[{}]\s*$/)
=> "no match"
```

Write a RE to match lines that are exactly three characters long.

```
>> show_match("123", /^...$/)
=> "<<123>>"
```

convow.rb with anchors

Recall that **convow.rb** on slide 201 simply does **char.downcase** on any characters it doesn't recognize. **downcase** doesn't change **^** or **$**.

The command

```
% ruby convow.rb ^cvc$
```

builds this this RE

```
/^[^aeiou][aeiou][^aeiou]$/
```

Let's explore with it:

```
% ruby convow.rb ^cvc$ < web2 | wc -l
858
% ruby convow.rb ^vcv$ < web2 | wc -l
92
% ruby convow.rb ^vccccv$ < web2 | wc -l
15
% ruby convow.rb ^vcccccv$ < web2
|oxyphyte|
```

Anchors, continued

What does `/\w+\d+/` specify?

One or more "word" characters followed by one or more digits.

How do the following matches differ from each other?

```
line =~ /\w+\d+/

line =~ /^\w+\d+/

line =~ /\w+\d+$/

line =~ /^\w+\d+$/

line =~ /^.\w+\d+.$/

line =~ /^.*\w+\d+$/
```

Sidebar: Dealing with too much input

Imagine a program that's reading dozens of large data files whose lines start with first names, like "**Mary,**". We're getting drowned by the data.

```
for fname in files
  f = open(fname)
  while line = f.gets
    ...lots of processing to build a data structure, bdata...
  end
  p bdata        # WAY TOO MUCH to easily analyze!!
```

Should we edit a copy of the data files to include only a handful of names?

A better solution, using an RE:

```
for fname in files
  f = open(fname)
  while line = f.gets
    next unless line =~ /^(John|Dana|Mary),/
    ...processing...        # toomuch.rb
```

NOTE: COMMA ADDED!

Groups and references

A side effect of enclosing a regular expression in parentheses is that when a match is found, a "group" is created that contains the matched text. That "group" can be referenced later in the same regular expression.

Here's a regular expression that matches five-character palindromes, like "civic" and "kayak":

```
/(.)(.).\2\1/
```

Piece by piece:

- `(.)` Match a character and save that one-character string as group 1
- `(.)` Match a character and save it as group 2
- `.` Match any character
- `\2` Match the text held by group 2
- `\1` Match the text held by group 1

Groups and references, continued

A RE that matches five-character palindromes: **/(.)(.).\2\1/**

smg, in **www/ruby/dotirbrc** is a variant of **show_match** that shows groups, too:

```
>> smg("Please refer to the kayak radar",/(.)(.).\2\1/)
"Please <<refer>> to the kayak radar"
$1 = <<r>>, $2 = <<e>>
```

<u>To find the number of a group, count left parentheses.</u> 1 2

Why does the following fail?

```
>> smg("Please refer to the kayak radar",/((.)(.).\2\1)/)
no match
```

The added parentheses make the group numbers wrong! 1 2 3

Groups and references, continued

At hand:

```
>> smg("Please refer to the kayak radar",/((.)(.).\2\1)/)
no match
```

1 2 3

Groups are numbered by counting left parentheses. The parentheses we added to wrap the entire RE make the previously correct group numbers wrong!

Solution—just renumber:

```
>> smg("Please refer to the kayak radar",/((.)(.).\3\2)/)
"Please <<refer>> to the kayak radar"
$1 = <<refer>>, $2 = <<r>>, $3 = <<e>>
```

scan makes arrays of arrays containing the groups for each match found.

```
>> "Please refer to the kayak radar".scan(/((.)(.).\3\2)/)
=> [["refer", "r", "e"], ["kayak", "k", "a"], ["radar", "r", "a"]]
```

Groups and references, continued

<u>After a successful match, **$N** is set to the Nth group.</u>

```
while line = gets
   if line =~ /((\d+)\+(\d+))/ then
      left = $2.to_i
      right=$3.to_i

      puts "#{$1} = #{left + right}"
   else puts "?" end
end  # group1.rb
```

Usage:

```
$ ruby group1.rb          (keyboard input is underlined)
3+4
3+4 = 7
How about 12+345?    (note match in middle of line)
12+345 = 357
What's 1000 + 2000?
?                    (why no match?!)
```

Groups and references, continued

What does the regular expression `(\w\w\w+).*\1.*\1` describe?

Strings with a substring of 3+ "word" characters that appears three or more times.

What does the following program do? (Groups can be ref'd with `$N`.)

```
while line = gets
   if line =~ /(\w\w\w+).*\1.*\1/ then
      puts line
      puts line.gsub($1, "^" * $1.size).gsub(/[^^]/, " ")
   end
end
```

Usage:

```
% cat ~/372/work/*.rb | ruby ~/372/ruby/triplematch.rb
                strPrint = strPrint + (getString str,j) +" "
                ^^^        ^^^                    ^^^
    addlinetostring s1,line  # equivalent to s1 += line
       ^^^^            ^^^^                            ^^^^
```

Iteration with **gsub**

Recall **String#gsub**:

```
>> "load = max * 2".gsub(/\w/,".")    => ".... = ... * ."
```

gsub has a one argument form that is an iterator. Each match is passed to the block in turn. The result of the block is substituted for the match.

Imagine augmenting a string with a running sum of the numbers it holds:

```
>> running_sum("1 pencil, 3 erasers, 2 pens")
=> "1(1) pencil, 3(4) erasers, 2(6) pens"

def running_sum(s)
   sum = 0
   s.gsub(/\d+/) do |match|
      sum += match.to_i
      match + "(%d)" % sum   # string formatting operator
   end
end # running_sum.rb
```

Application: Time totaling

Consider an application that reads elapsed times on standard input and prints their total:

```
% ruby ttl.rb
3h
15m
4:30
^D
7:45
```

Multiple times can be specified per line, separated by spaces and commas.

```
% ruby ttl.rb
10m, 3:30
20m 2:15 1:01 3h
^D
10:16
```

How can we approach it?

Time totaling, continued

```
def main
  mins = 0
  while line = gets do
    line.scan(/[^\s,]+/).each {|time| mins += parse_time(time) }
  end
  printf("%d:%02d\n", mins / 60, mins % 60)
end

def parse_time(s)
  if s =~ /^(\d+):([0-5]\d)$/
    $1.to_i * 60 + $2.to_i
  elsif s =~ /^(\d+)([hm])$/
    if $2 == "h" then $1.to_i * 60
    else $1.to_i end
  else
    0 # return 0 for things that don't look like times
  end
end

main unless $0 == "irb"     # Note: Was $PROGRAM_NAME
```

Application: **ftypes.hs** checker

Assignment 2's **ftypes.hs** asked you to create a function **fd** that had a type <u>equivalent</u> to this:

```
(a, Int) -> (Int, t) -> (t, [a])
```

Problem: write **equiv_to_fd(type)** that returns true or false depending on whether **type** is equivalent to that expected for **fd**.

```
>> equiv_to_fd("(t1, Int) -> (Int, t) -> (t, [t1])")   => true
>> equiv_to_fd("(b, Int) -> (Int, a) -> (a, [b])")   => true
>> equiv_to_fd("(a, Int) -> (Int, a) -> (a, [b])")   => false

def equiv_to_fd(s)
   !! (s =~ /^\((.+?), Int\) -> \(Int, (.+?)\) -> \(\2, \[\1\]\)$/)
end                                               # equiv_to_fd.rb
```

Note:

Use of reluctant/non-greedy **+?** operator.

The numerous backslash escapes for literal **()**s and **[]**s.

ftypes.hs checker, continued

At hand:

```
>> equiv_to_fd("(t1, Int) -> (Int, t) -> (t, [t1])")   => true
>> equiv_to_fd("(a, Int) -> (Int, a) -> (a, [b])")   => false
```

Current solution:

```
def equiv_to_fd(s)
   !! (s =~ /^\((.+?), Int\) -> \(Int, (.+?)\) -> \(\2, \[\1\]\)$/)
end
```

The above uses non-greedy +. Below we use [^,]+ and [^)]+

```
def equiv_to_fd0(s)
   !! (s =~ /^\(([^,]+), Int\) -> \(Int, ([^)]+)\) -> \(\2, \[\1\]\)$/)
end
```

How about a generalized version, **equiv_type(exp_type, act_type)**?

Lots more with regular expressions

Our venture into regular expressions ends here but there's lots more, like...

- Nested regular expressions
- Named matches
- Nested and conditional groups
- Conditional subpatterns
- Zero-width positive lookahead

Proverb:

A programmer decided to use regular expressions to solve a problem.
Then he had two problems.

Regular expressions are great, up to point.

SNOBOL4 patterns, Icon's string scanning facility, and Prolog grammars can all recognize unrestricted languages and are far less complex than the regular expression facility in most languages.

Sidebar: A question that was headed for the final

In Ruby an out of bounds array or string reference like `x[n]` produces `nil`.

In Java an out of bounds reference with `s.charAt(n)` or `array[n]` produces an exception.

Imagine a proposed change to Java wherein an out of bounds reference `s.charAt(n)` or `array[n]` would return `null`. Present an argument that is in either in favor of this change or against it.

Answer:

It won't work!

(1) `null` isn't a valid `char` value.

(2) `null` for `array[n]` would only work with arrays of references.

To what extent does dynamic typing enable Ruby's behavior?

Defining classes

A tally counter

Imagine a class named **Counter** that models a tally counter.

Here's how we might create and interact with an instance of Counter:


```
c1 = Counter.new
c1.click
c1.click

puts c1    # Output: Counter's count is 2
c1.reset

c2 = Counter.new "c2"
c2.click

puts c2    # Output: c2's count is 1

c2.click
puts "c2 = #{c2.count}"    # Output: c2 = 2
```

Here is a partial implementation of **Counter**:

```
class Counter
   def initialize(label = "Counter")
      ...
   end
   ...
end  # Counter.rb
```

Class definitions are bracketed with **class** and **end**. Class names must start with a capital letter. Unlike Java there are no filename requirements.

The **initialize** method is the constructor, called when **new** is invoked.

```
c1 = Counter.new
c2 = Counter.new "c2"
```

If no argument is supplied to **new**, the default value of "**Counter**" is used.

Here is the body of **initialize**:

```
class Counter
  def initialize(label = "Counter")
    @count = 0
    @label = label
  end
end
```

Instance variables are identified by prefixing them with **@**.

<u>An instance variable comes into existence when it is assigned to</u>. The code above creates **@count** and **@label**. (There are no instance variable declarations.)

Just like Java, each object has its own copy of instance variables.

Counter, continued

Let's add **click** and **reset** methods, which are straightforward:

```
class Counter
    def initialize(label = "Counter")
        @count = 0
        @label = label
    end

    def click
        @count += 1
    end

    def reset
        @count = 0
    end
end
```

Counter, continued

In Ruby the instance variables of an object cannot by accessed by any other object.

The only way way to make the value of **@count** available to other objects is via methods.

Here's a simple "getter" for the counter's count.

```
def count
   @count
end
```

Let's override **Object#to_s** with a **to_s** that produces a detailed description:

```
def to_s
   return "#{@label}'s count is #{@count}"
end
```

In Ruby, there is simply no such thing as a public instance variable. All access must be through methods.

Counter, continued

Full source for **Counter** thus far:

```
class Counter
  def initialize(label = "Counter")
    @count = 0; @label = label
  end

  def click
    @count += 1
  end

  def reset
    @count = 0
  end

  def count # Note the convention: count, not get_count
    @count
  end

  def to_s
    return "#{@label}'s count is #{@count}"
  end
end # Counter.rb
```

Common error: omitting the @ on a reference to an instance variable.

An interesting thing about instance variables

Consider this class: (**instvar.rb**)

```
class X
   def initialize(n)
      case n
         when 1 then @x = 1
         when 2 then @y = 1
         when 3 then @x = @y = 1
 end; end; end
```

What's interesting about the following?

```
>> X.new 1
=> #<X:0x00000101176838 @x=1>

>> X.new 2
=> #<X:0x00000101174970 @y=1>

>> X.new 3
=> #<X:0x0000010117aaa0 @x=1, @y=1>
```

Addition of methods

If **class X ... end** has been seen and another **class X ... end** is encountered, the second definition adds and/or replaces methods.

Let's confirm **Counter** has no **label** method.

```
>> c = Counter.new "ctr 1"

>> c.label
NoMethodError: undefined method `label' ...
```

Now we add a **label** method: (we're typing lines into **irb** but could **load**)

```
>> class Counter
>>   def label; @label; end
>> end

>> c.label       => "ctr 1"
```

What's an implication of this capability?

We can add methods to built-in classes!

Addition of methods, continued

In Icon, the unary **?** operator can be used to generate a random number or select a random value from an aggregate.

```
Icon Evaluator, Version 1.1
][ ?10
  r1 := 3  (integer)

][ ?"abcd"
  r2 := "b"  (string)
```

I miss that. Let's add something similar to Ruby!

If we call **Kernel#rand** with a **Fixnum n** it will return a random **Fixnum** greater than or equal to zero and less than **n**.

There's no unary **?** to overload in Ruby so let's just add a **rand** method to **Fixnum** and **String**.

Addition of methods, continued

Here is **random.rb**:

```
class Fixnum
    def rand
      Kernel.rand(self)+1
    end
end

class String
    def rand
      self[self.size.rand-1]  # Uses Fixnum.rand
    end
end
```


```
>> load "random.rb"        => true

>> 12.times { print 6.rand, " " }     # Output: 2 1 2 4 2 1 4 3 4 4 6 3

>> 8.times { print "HT".rand, " " }  # Output: H H T H T T H H
```

An interesting thing about class definitions

Observe the following. What does it suggest to you?

```
>> class X
>> end
=> nil

>> p (class Y; end)
nil
=> nil

>> class Z; puts "here"; end
here
=> nil
```

Class definitions are executable code!

Class definitions are executable code

At hand: A class definition is executable code. The following class definition uses a **case** statement to selectively execute **def**s for methods.

```
class X
    print "What methods would you like? "
    gets.split.each do |m|
        case m
            when "f" then def f; "from f" end
            when "g" then def g; "from g" end
            when "h" then def h; "from h" end
            end
    end
end
```

Use:

```
>> load "dynmethods1.rb"
What methods would you like? f g
>> x = X.new        => #<X:0x007fc45c0b0f40>
>> x.f              => "from f"
>> x.g              => "from g"
>> x.h
NoMethodError: undefined method `h' for #<X:...>
```

Sidebar: Fun with eval

Kernel#eval parses a string containing Ruby source code and executes it.

```
>> s = "abc"

>> n = 3

>> eval "x = s * n"              => "abcabcabc"

>> x                             => "abcabcabc"

>> eval "x[2..-2].length"        => 6

>> eval gets
s.reverse
=> "cba"
```

Note that **eval** uses variables from the current scope and that an assignment to **x** is reflected in the current scope. (Note: There are details about scoping!)

Bottom line: A Ruby program can generate code for itself.

Sidebar, continued

mk_methods.rb prompts for a method name, parameters, and method body. It then creates that method and adds it to class **X**.

```
>> load "mk_methods.rb"
What method would you like? add
Parameters? a, b
What shall it do? a + b
Method add(a, b) added to class X

What method would you like? last
Parameters? x
What shall it do? x[-1]
Method last(x) added to class X
```

What method would you like? `^D => true`

```
>> x = X.new        => #<X:0x0000010185d930>
>> x.add(3,4)       => 7
>> x.last "abcd"    => "d"
```

Sidebar, continued

Here is **mk_methods.rb**. Note that the body of the class is a **while** loop.

```
class X
  while (print "What method would you like? "; name = gets)
    name.chomp!

    print "Parameters? "
    params = gets.chomp

    print "What shall it do? "
    body = gets.chomp

    code = "def #{name} #{params}; #{body}; end"

    eval(code)
    print("Method #{name}(#{params}) added to class #{self}\n\n");
  end
end
```

Is this a useful capability or simply fun to play with?

Sidebar: Risks with eval

Does eval pose any risks?

```
while (print("? "); line = gets)
   eval(line)
end # eval1.rb
```

Interaction: (input is underlined)

```
% ruby eval1.rb
? puts 3*5
15
? puts "abcdef".size
6
? system("date")
Mon Mar 23 19:09:35 MST 2015
? system("rm –rf ...")
...
? system("chmod 777 ...")
...
```

Sidebar, continued

At hand:

```
% ruby eval1.rb
? system("rm –rf ...")
...
? system("chmod 777 ...")
...
```

```
while (print("? "); line = gets)
    eval(line)
end # eval1.rb
```

But, we can do those things without using Ruby!

eval gets risky when we can't trust the source of the data. Examples:

A collaborator on a project sends us a data file.

A Ruby on Rails web app calls **eval** with user-supplied data. (!)

It's very easy to fall victim to a variety of code-injection attacks when using **eval**.

The **define_method** (et. al) machinery is often preferred over **eval** but risks still abound!

Related topic: Ruby supports the notion of *tainted* data.

Class variables and methods

Like Java, Ruby provides a way to associate data and methods with a class itself rather than each instance of a class.

Java uses the **static** keyword to denote a class variable.

<u>In Ruby a variable prefixed with two at-signs is a class variable.</u>

Here is **Counter** augmented with a class variable that keeps track of how many counters have been created.

```
class Counter
  @@created = 0 # Must precede any use of @@created
  def initialize(label = "Counter")
    @count = 0; @label = label
    @@created += 1
  end
end
```

Note: Unaffected methods are not shown.

Class variables and methods, continued

To define a class method, simply prefix the method name with the name of the class:

```
class Counter
   @@created = 0
   ...
   def Counter.created
      return @@created
   end
end
```

Usage:

```
>> Counter.created                 => 0
>> c = Counter.new
>> Counter.created                 => 1
>> 5.times { Counter.new }
>> Counter.created                 => 6
```

A little bit on access control

By default, methods are public. If **private** appears on a line by itself, subsequent methods in the class are private. Ditto for **public**.

```
class X
    def f; puts "in f"; g end     # Note: calls g
  private
    def g; puts "in g" end
end
```

Usage:

```
>> x = X.new
>> x.f
in f
in g
>> x.g
NoMethodError: private method `g' ...
```

Speculate: What are **private** and **public**? Keywords?

Methods in **Module**! (**Module** is an ancestor of **Class**.)

Getters and setters

If **Counter** were in Java, we might provide methods like **void setCount(int n)** and **int getCount()**.

Our **Counter** already has a **count** method as a "getter".

For a "setter" we implement **count=**, with a trailing equals sign.

```
def count= n
    print("count=(#{n}) called\n")   # Just for observation
    @count = n unless n < 0
end
```

Usage:

```
>> c = Counter.new
>> c.count = 10
count=(10) called
=> 10
>> c                      => Counter's count is 10
```

Getters and setters, continued

Here's a class to represent points on a Cartesian plane:

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

Usage:

```
>> p1 = Point.new(3,4)  => #<Point:0x00193320 @x=3, @y=4>
>> [p1.x, p1.y]         => [3, 4]
```

It can be tedious and error prone to write a number of simple getter methods like **Point#x** and **Point#y**.

Getters and setters, continued

The method **attr_reader** <u>creates</u> getter methods.

Here's an equivalent definition of **Point**:

```
class Point
   def initialize(x, y)
      @x = x
      @y = y
   end
   attr_reader :x, :y     # Could use "x" & "y" instead
end
```

Usage:

```
>> p = Point.new(3,4)
>> p.x                          => 3
>> p.x = 10
NoMethodError: undefined method `x=' for #<Point: ...>
```

Why does **p.x = 10** fail?

Getters and setters, continued

If you want both getters and setters, use **attr_accessor**.

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

Usage:

```
>> p = Point.new(3,4)
>> p.x
=> 3
>> p.y = 10
```

<u>It's important to appreciate that</u> **attr_reader** <u>and</u> **attr_accessor** <u>are methods that create methods</u>. (What if Ruby didn't provide them?)

Operator overloading

Operator overloading

In most languages at least a few operators are "overloaded"—an operator stands for more than one operation.

C: `+` is used to express addition of integers, floating point numbers, and pointer/integer pairs.

Java: `+` is used to express addition and string concatenation.

Icon: `*x` produces the number of...
- characters in a string
- values in a list
- key/value pairs in a table
- results a "co-expression" has produced and more...

Icon: `+` means only addition; `s1 || s2` is string concatenation

What are examples of overloading in Ruby? In Haskell?

Operators as methods

Most Ruby operators can be expressed as method calls.

```
>> 3.+(4)                  => 7        # 3 + 4

>> "testing".[](2,3)       => "sti"    # "testing"[2,3]

>> 10.==20                 => false    # 10 == 20
```

In general, **expr1 op expr2** can be written as **expr1.op expr2**

Unary operators are indicated by adding @ after the operator:

```
>> 5.-@()        => -5                 # -5

>> "abc".!@()    => false              # !"abc"
```

What are some binary operations that might be problematic to express as a method call?

Operator overloading, continued

We'll use a dimensions-only rectangle class to study overloading in Ruby:

```
class Rectangle
    def initialize(w,h)
        @width, @height = w, h       # parallel assignment
    end
    def area; @width * @height; end
    attr_reader :width, :height
    def inspect
        "#{width} x #{height} Rectangle"  # silly use of % fixed!
    end
end
```

Usage:

```
>> r = Rectangle.new(3,4)    => 3 x 4 Rectangle
>> r.area                    => 12
>> r.width                   => 3
```

Operator overloading, continued

Let's imagine that we can compute the "sum" of two rectangles:

```
>> a = Rectangle.new(3,4)     => 3 x 4 Rectangle

>> b = Rectangle.new(5,6)     => 5 x 6 Rectangle

>> a + b                      => 8 x 10 Rectangle

>> c = a + b + b              => 13 x 16 Rectangle

>> (a + b + c).area           => 546
```

As shown above, what does **Rectangle + Rectangle** mean?

Operator overloading, continued

Our vision:

```
>> a = Rectangle.new(3,4); b = Rectangle.new(5,6)
>> a + b => 8 x 10 Rectangle
```

Here's how to make it so:

```
class Rectangle
  def + rhs
     Rectangle.new(self.width + rhs.width, self.height + rhs.height)
  end
end
```

Remember that `a + b` is equivalent to `a.+(b)`. We are invoking the method "+" on `a` and passing it `b` as a parameter.

The parameter name, `rhs`, stands for "right-hand side".

Do we need `self` in `self.width` or would just `width` work? How about `@width`?

Even if somebody else had provided `Rectangle`, we could still overload `+` on it—the lines above are additive, assuming `Rectangle.freeze` hasn't been done.

Operator overloading, continued

For reference:

```
def + rhs
  Rectangle.new(self.width + rhs.width, self.height + rhs.height)
end
```

Here is a **faulty implementation** of our +, and usage of it:

```
def + rhs
  @width += rhs.width; @height += rhs.height
end

>> a = Rectangle.new(3,4)
>> b = Rectangle.new(5,6)

>> c = a + b                    => 10

>> a                            => 8 x 10 Rectangle
```

What's the problem?

We're changing the attributes of the left operand instead of creating and returning a new instance of Rectangle.

Operator overloading, continued

Just like with regular methods, we have complete freedom to define what's meant by an expression using an overloaded operator.

Here is a method for Rectangle that defines unary minus to be an imperative "rotation" (a clear violation of the Principle of Least Astonishment!)

```
def -@      # Note: @ suffix to indicate unary form of -
    # Use parallel assignment to swap
    @width, @height = @height, @width
    self
end

>> a = Rectangle.new(2,5)   => 2 x 5 Rectangle
>> -a                       => 5 x 2 Rectangle
>> a + -a                   => 4 x 10 Rectangle
>> a                        => 2 x 5 Rectangle
```

Goofy, yes?

Operator overloading, continued

At hand:

```
def -@
   # Use parallel assignment to swap
   @width, @height = @height, @width
   self
end
```

What's a (slightly) more sensible implementation of unary -?

```
def -@
    Rectangle.new(height, width)
end

>> a = Rectangle.new(5,2)   => 5 x 2 Rectangle
>> -a                       => 2 x 5 Rectangle
>> a                        => 5 x 2 Rectangle
>> a += -a; a               => 7 x 7 Rectangle
```

Operator overloading, continued

Consider "scaling" a rectangle by some factor. Example:

```
>> a = Rectangle.new(3,4)    => 3 x 4 Rectangle
>> b = a * 5                 => 15 x 20 Rectangle
>> c = b * 0.77              => 11.55 x 15.4 Rectangle
```

Implementation:

```
def * rhs
  Rectangle.new(self.width * rhs, self.height * rhs)
end
```

A problem:

```
>> a          => 3 x 4 Rectangle
>> 3 * a
TypeError: Rectangle can't be coerced into Fixnum
```

What's wrong?

We've implemented only Rectangle * Fixnum

What should a / 3 do?

Operator overloading, continued

Imagine a case where it's useful to reference width and height uniformly, via subscripts:

```
>> a = Rectangle.new(3,4)   => 3 x 4 Rectangle
>> a[0]                     => 3
>> a[1]                     => 4
>> a[2]                     RuntimeError: out of bounds
```

Note that `a[n]` is `a.[ ](n)`

Implementation:

```
def [] n
   case n
      when 0 then width
      when 1 then height
      else raise "out of bounds"
   end
end
```

Is Ruby extensible?

A language is considered to be <u>extensible</u> if we can create new types that can be used as easily as built-in types.

Does our simple **Rectangle** class and its overloaded operators demonstrate that Ruby is extensible?

What would **a = b + c * 2** with **Rectangle**s look like in Java?

Maybe: **Rectangle a = b.plus().times(2);**

How about in C?

Would **Rectangle a = rectPlus(b, rectTimes(c, 2));** be workable?

Haskell goes further with extensibility, allowing new operators to be defined.

Ruby is mutable

Ruby is not only extensible; it is also <u>mutable</u>—we can change the meaning of expressions.

If we wanted to be sure that a program never used integer addition, we could start with this:

```
class Fixnum
  def + x
    raise "boom!"
  end
end
```

What else would we need to do?

Contrast: C++ is extensible, but not mutable. For example, in C++ you can define the meaning of **Rectangle * int** but you can't change the meaning of integer addition, as we do above.

Inheritance

A **Shape** hierarchy in Ruby

Here's the classic **Shape/Rectangle/Circle** inheritance example in Ruby:

```
class Shape
    def initialize(label)
        @label = label
    end

    attr_reader :label
end
```

```
class Rectangle < Shape
    def initialize(label, width, height)
        super(label)
        @width, @height = width, height
    end

    def area
        return width * height
    end

    def inspect
        "Rectangle #{label} (#{width} x
#{height})"
    end

    attr_reader :width, :height
end
```

Rectangle < Shape specifies inheritance.

Note that **Rectangle** methods use the generated **width** and **height** methods rather than **@width** and **@height**.

Shape, continued

```
class Circle < Shape
   def initialize(label, radius)
      super(label)
      @radius = radius
   end

   def area
      return Math::PI * radius * radius
   end

   def perimeter
      return Math::PI * radius * 2
   end

   def inspect
      "Circle #{label} (r = #{radius})"
   end

   attr_reader :radius
end
```

Math::PI references the constant **PI** in the **Math** class.

There's no **abstract**

The **abstract** reserved word is used in Java to indicate that a class, method, or interface is abstract.

Ruby does not have any language mechanism to mark a class or method as abstract.

Some programmers put "abstract" in class names, like **AbstractWindow**.

A method-level practice is to have abstract methods raise an error if called:

```
class Shape
   def area
      raise "Shape#area is abstract"
   end
end
```

There is also an **abstract_method** "gem" (a package of code and more):

```
class Shape
   abstract_method :area
   ...
```

Inheritance is important in Java

A common use of inheritance in Java is to let us write code in terms of a superclass type and then use that code to operate on subclass instances.

With a **Shape** hierarchy in Java we might write a routine **sumOfAreas**:

```
static double sumOfAreas(Shape shapes[]) {
    double area = 0.0;
    for (Shape s: shapes)
        area += s.getArea();
    return area;
    }
```

We can make **Shape.getArea()** abstract to force concrete subclasses to implement **getArea()**.

sumOfAreas is written in terms of **Shape** but works with instances of any subclass of **Shape**.

Inheritance is less important in Ruby

Here is **sumOfAreas** in Ruby:

```
def sumOfAreas(shapes)
   area = 0
   for shape in shapes do
      area += shape.area
   end
   area
end
```

Does it make any use of inheritance?

Even simpler:

```
sum = shapes.inject (0.0) {|memo,shape| memo + shape.area }
```

Dynamic typing in Ruby makes it unnecessary to require common superclasses or interfaces to write polymorphic methods that operate on a variety of underlying types.

If you look closely, you'll find that many common design patterns are simply patterns of working with inheritance hierarchies in statically typed languages.

Example: ~~XString~~ VString

Imagine an abstract class VString with two concrete subclasses: **ReplString** and **MirrorString**.

A **ReplString** is created with a string and a replication count. It supports **size**, substrings with **[pos]** and **[pos,len]**, and **to_s** operations.

```
>> r1 = ReplString.new("abc", 2)    => ReplString(6)

>> r1.size       => 6

>> r1[0]         => "a"

>> r1[10]        => nil

>> r1[2,3]       => "cab"

>> r1.to_s       => "abcabc"
```

XString has been changed to VString on all the following slides!

VString, continued

A **MirrorString** represents a string concatenated with a reversed copy of itself.

```
>> m1 = MirrorString.new("abcdef")
=> MirrorString("abcdef")

>> m1.to_s                    => "abcdeffedcba"

>> m1.size                    => 12

>> m1[3,6]                    => "deffed"
```

What's a trivial way to implement the **VString**/**ReplString**/**MirrorString** hierarchy?

A trivial **VString** implementation

```
class VString
  def initialize(s)
    @s = s
  end

  def [](start, len = 1)
    @s[start, len]
  end

  def size
    @s.size
  end

  def to_s
    @s.dup
  end

end
```

```
class ReplString < VString
  def initialize(s, n)
    super(s * n)
  end

  def inspect
    "ReplString(#{size})"
  end
end

class MirrorString < VString
  def initialize(s)
    super(s + s.reverse)
  end

  def inspect
    "MirrorString(#{size})"
  end
end
```

VString, continued

New requirements:

A VString can be created using either an VString or a String.

A ReplString can have a very large replication count.

Will VStrings in constructors work with the implemetation as-is?

```
>> m2 = MirrorString.new(ReplString.new("abc",3))
NoMethodError: undefined method `reverse' for ReplString

>> r2 = ReplString.new(MirrorString.new("abc"),5)
NoMethodError: undefined method `*' for MirrorString
```

What's the problem?

*The ReplString and MirrorString constructors use * n and .reverse*

What will ReplString("abc", 1_000_000_000_000) do?

VString, continued

Here's behavior with a working version:

```
>> s1 = ReplString.new("abc", 2_000_000_000_000)
=> ReplString("abc",2000000000000)

>> s1[0]                    => "a"

>> s1[-1]                   => "c"

>> s1[1_000_000_000]        => "b"

>> s2 = MirrorString.new(s1)
=> MirrorString(ReplString("abc",2000000000000))

>> s2.size                  => 12000000000000

>> s2[-1]                   => "a"

>> s2[s2.size/2 - 3, 6]     => "abccba"
```

VString, continued

Let's review requirements:

- Both **ReplString** and **MirrorString** are subclasses of **VString**.
- A **VString** can be created using either a **String** or a **VString**.
- The **ReplString** replication count can be a **Bignum**.
- If **xs** is a **VString**, **xs[pos]** and **xs[pos,len]** produce **String**s.
- **VString#size** works, possibly producing a **Bignum**.
- **VString#to_s** "works" but is problematic with long strings.

How can we make this work?

VString, continued

Let's play computer!

```
>> s = MirrorString.new(ReplString.new("abc",1_000_000))
=> MirrorString(ReplString("abc",1000000))

>> s.size
=> 6000000

>> s[-1]
=> "a"

>> s[3_000_000]
=> "c"

>> s[3_000_000,6]
=> "cbacba"
```

VString stands for "virtual string"—the hierarchy creates the illusion of very long strings by using very little data.

To be continued, on assignment 6!

What data did you need to perform those computations?

Modules and "mixins"

Modules

A Ruby *module* can be used to group related methods for organizational purposes.

Imagine some methods to comfort a homesick Haskell programmer at Camp Ruby:

```
module Haskell
   def Haskell.head(a)
      a[0]
   end

   def Haskell.tail(a)
      a[1..-1]
   end
   ...more...
end

>> a = [10, "twenty", 30, 40.0]
>> Haskell.head(a)
=> 10
>> Haskell.tail(a)
=> ["twenty", 30, 40.0]
```

Modules as "mixins"

In addition to providing a way to group related methods, a module can be "included" in a class. When a module is used in this way it is called a "mixin" because it mixes additional functionality into a class.

Here is a revised version of the Haskell module. The class methods are now written as instance methods; they use **self** and have no parameter:

```
module Haskell
   def head
      self[0]
   end

   def tail
      self[1..-1]
   end
end
```

Previous version:

```
module Haskell
   def Haskell.head(a)
      a[0]
   end

   def Haskell.tail(a)
      a[1..-1]
   end
end
```

Mixins, continued

We can mix our Haskell methods into the **Array** class like this:

```
% cat mixin1.rb
require './Haskell'     # loads ./Haskell.rb if not already loaded
class Array
   include Haskell
end
```

We can load **mixin1.rb** and then use **.head** and **.tail** on arrays:

```
>> load "mixin1.rb"
>> ints = (1..10).to_a => [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

>> ints.head
=> 1

>> ints.tail
=> [2, 3, 4, 5, 6, 7, 8, 9, 10]

>> ints.tail.tail.head
=> 3
```

Mixins, continued

We can add those same capabilities to **String**, too:

```
class String
   include Haskell
end
```

Usage:

```
>> s = "testing"

>> s.head                 => "t"
>> s.tail                 => "esting"
>> s.tail.tail.head       => "s"
```

Does Java have any sort of mixin capability? What would be required to produce a comparable effect?

In addition to the include mechanism, what other aspect of Ruby facilitates mixins?

Modules and superclasses

The Ruby core classes and standard library make extensive use of mixins.

The class method **ancestors** can be used to see the superclasses and modules that contribute methods to a class:

```
>> Array.ancestors
=> [Array, Enumerable, Object, Kernel, BasicObject]

>> Fixnum.ancestors
=> [Fixnum, Integer, Numeric, Comparable, Object, Kernel,
BasicObject]

>> load "mixin1.rb"

>> Array.ancestors
=> [Array, Haskell, Enumerable, Object, Kernel, BasicObject]
```

Modules and superclasses, continued

The method **included_modules** shows the modules that a class **include**s.

```
>> Array.included_modules     => [Haskell, Enumerable, Kernel]

>> Fixnum.included_modules  => [Comparable, Kernel]
```

instance_methods can be used to see what methods are in a module:

```
>> Enumerable.instance_methods.sort =>
[:all?, :any?, :chunk, :collect, :collect_concat, :count, :cycle, :de
tect, :drop, :drop_while, :each_cons, :each_entry, ...more...

>> Comparable.instance_methods.sort
=> [:<, :<=, :==, :>, :>=, :between?]

>> Haskell.instance_methods
=> [:head, :tail]
```

Modules and superclasses, continued

All classes except BasicObject include the module Kernel.

If no superclass is specified, a class subclasses Object.

Example:

```
>> class X; end

>> X.ancestors              => [X, Object, Kernel, BasicObject]

>> X.included_modules       => [Kernel]

>> X.superclass             => Object
```

Note the inheritance structure: (And that Class and Module are classes!)

```
>> Class.superclass         => Module
>> Module.superclass        => Object
```

Expressed in Ruby: Class < Module < Object

Modules and superclasses, continued

BasicObject is the superclass of Object.

BasicObject was introduced to provide a (nearly) blank slate for some uses with metaprogramming.

BasicObject includes no modules.

```
>> Object.instance_methods.size      => 57

>> BasicObject.instance_methods.size => 8

>> BasicObject.included_modules      => []
```

The **Enumerable** module

When talking about iterators we encountered **Enumerable**. It's a module:

```
>> Enumerable.class      => Module
>> Enumerable.instance_methods.sort =>
[:all?, :any?, :chunk, :collect, :collect_concat, :count, :cycle, :de
tect, :drop, :drop_while, :each_cons, :each_entry, :each_slice, :
each_with_index, :each_with_object, :entries, :find, :find_all, :f
ind_index, :first, :flat_map, :grep, :group_by, :include?, :inject,
:map, :max, :max_by, :member?, :min, :min_by, :minmax, :min
max_by, :none?, :one?, :partition, :reduce, :reject, :reverse_eac
h, :select, :slice_before, :sort, :sort_by, :take, :take_while, :to_a,
:zip]
```

The methods in **Enumerable** use duck typing, requiring only an **each** method. **min**, **max**, and **sort**, also require **<=>** for values operated on.

<u>If class implements **each** and includes **Enumerable** then all those methods become available to instances of the class.</u>

The **Enumerable** module, continued

Here's a class whose instances simply hold three values:

```
class Trio
   include Enumerable
   def initialize(a,b,c); @values = [a,b,c]; end

   def each
      @values.each {|v| yield v }
   end
end
```

Because **Trio** implements **each** and includes **Enumerable**, we can do a lot with it:

```
>> t = Trio.new(10, "twenty", 30)
>> t.member?(30) => true
>> t.map{|e| e * 2} => [20, "twentytwenty", 60]
>> t.partition {|e| e.is_a? Numeric } => [[10, 30], ["twenty"]]
```

What would the Java equivalent be for the above?

The **Comparable** module

Another common mixin is **Comparable**:

```
>> Comparable.instance_methods
=> [:==, :>, :>=, :<, :<=, :between?]
```

Comparable's methods are implemented in terms of **<=>**.

Let's compare rectangles on the basis of areas:

```
class Rectangle
  include Comparable
  def <=> rhs
    (self.area - rhs.area) <=> 0
  end
end
```

Comparable, continued

Usage:

```
>> r1 = Rectangle.new(3,4)  => 3 x 4 Rectangle
>> r2 = Rectangle.new(5,2)  => 5 x 2 Rectangle
>> r3 = Rectangle.new(2,2)  => 2 x 2 Rectangle

>> r1 < r2                     => false

>> r1 > r2                     => true

>> r1 == Rectangle.new(6,2)    => true

>> r2.between?(r3,r1)          => true
```

Is Comparable making the following work?

```
>> [r1,r2,r3].sort
=> [2 x 2 Rectangle, 5 x 2 Rectangle, 3 x 4 Rectangle]

>> [r1,r2,r3].min
=> 2 x 2 Rectangle
```

In conclusion...

What do you like (or not?) about Ruby?

- Everything is an object?
- Substring/subarray access with **x[...]** notation?
- Negative indexing to access from right end of strings and arrays?
- Modifiers? (**puts x if x > y**)
- Iterators and blocks?
- Ruby's support for regular expressions?
- ~~Monkey patching?~~ Adding methods to built-in classes?
- Programmer-defined operator overloading?
- Dynamic typing?

Is programming more fun with Ruby?

My first practical Ruby program

September 3, 2006:

```
n=1
d = Date.new(2006, 8, 22)
incs = [2,5]
pos = 0
while d < Date.new(2006, 12, 6)
    if d != Date.new(2006, 11, 23)
        printf("%s %s, #%2d\n",
            if d.cwday() == 2: "T"; else "H";end,
                d.strftime("%m/%d/%y"), n)
        n += 1
    end
    d += incs[pos % 2]
    pos += 1
end
```

Output:

```
T 08/22/06, # 1
H 08/24/06, # 2
T 08/29/06, # 3
...
```

More with Ruby...

If we had more time, we'd...

- Learn about **lambda**s, blocks as explicit parameters, and **call**.
- Play with **ObjectSpace.** (Try **ObjectSpace.count_objects**)
- Do some metaprogramming with *hooks* like **method_missing**, **included**, and **inherited**.
- Experiment with internal Domain Specific Languages (DSL).
- Look at how Ruby on Rails puts Ruby features to good use.
- Write a Swing app with JRuby, a Ruby implementation for the JVM.
- Take a peek at BDD (Behavior-Driven Development) with Cucumber and RSpec.