Ruby
Introduction
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 don't like the term "scripting language"!
- I describe Ruby as a dynamically typed object-oriented language.

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

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

Let's see what we can learn about Ruby by trying some expressions using irb on lectura.

```
3+4
_.class
"x".<TAB><TAB>
("abc"*3_500).size.to_s
s = ('a..'z').to_a.join
s[5,9]
s[5..9]
s[-3..]
"a test here".split.map { |s| s.size}
10[3]
s=`date; hostname`
s.gsub!("2","TWO"); print s
```

```
"call 555-1212" =~ /[0-9]+/  
[$`, $&, $']  # just after above
"9-3".gsub(/\d/) { |d| d.to_i-1}
{1 => 2.0, :three => [!nil, 3] }
if 1 < 2 then 3 else 4 end
while line = gets; p line; end
print 3 unless 4 < 5
3.+(4)
Rational(1,3) * 300
1.i**2
```
**irb**—Interactive Ruby Shell

**irb** is the commonly used REPL for Ruby. Let's run it on lectura:

```
% irb
irb(main):001:0> 3 + 4
=> 7
```

**irb**'s prompt changes if an expression is incomplete:

```
irb(main):002:0> "abc" +
```

```
irb(main):003:0>
```

```
irb(main):004:0>
```

```
irb(main):005:0> "xyz"
=> "abcxyz"
```

The slides won't show the result line (=> ...) when it is uninteresting.

Control-D terminates **irb**.

Use Control-C if you're stuck with an incomplete expression.
The latest stable version of Ruby is 3.1.2.

Due to both compatibility concerns and my late switch to Ruby, the only available version of Ruby on lectura is 2.7, which is the last 2.X version.

Slides and assignments will be based on 2.7, released on 2019-12-25.

Ruby 3.0 improvements focus on performance, parallelism & concurrency, and static type analysis.

The last major upheaval in Ruby occurred between 1.8 and 1.9. (2007)

In general, there are few incompatibilities between 1.9.3 (2011) and 3.X
**ruby-lang.org** is Ruby's home page

**ruby-doc.org/core-2.7.0** is the root of the 2.7.0 documentation

A few places of interest:

Ruby syntax:

**ruby-doc.org/core-2.7.0/doc/syntax_rdoc.html**

But, for example, there's no link from **Literals** in the table! Instead, click **literals.rdoc** in the **Files** bar on the left!

Operator precedence:

**ruby-doc.org/core-2.7.0/doc/syntax/precedence_rdoc.html**

Class-level documentation, with **String** as an example:

**https://ruby-doc.org/core-2.7.0/String.html**

**discord.com/invite/EnSevaRfct** — "Official" Ruby Discord server
Resources, continued

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"
– $29.95 for a DRM-free PDF at pragprog.com.
– I'll call it "PA".
– Programming Ruby 3.2 (5th Edition) is in "beta"

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 call it "RPL"

learning.oreilly.com has:

• Many relatively new Ruby books
  One recommendation: The Ruby Way, 3rd edition by Hal Fulton
• Lots of books on Ruby on Rails
• A number of of pre-1.9 Ruby books
Go to rubyinstaller.org/downloads and under "WITHOUT DEVKIT" get Ruby 2.7.6-1 (x64).

When installing, I recommend the default selections:
• Add Ruby executables to your PATH
• Associate .rb and .rbw files with this Ruby installation
• Use UTF-8 as default external encoding

Then, on the "Completing the Ruby Setup Wizard Page", deselect Run 'ridk install' to set up MSYS2… before you click "Finish". (You'll need to revisit this choice if you later want to install Ruby gems that have C extensions.)
The version of Ruby on recent versions of macOS should be fine for our purposes, assuming `ruby --version` shows 2.4 or better.

Alternatively, with Homebrew, you can do `brew install ruby@2.7`
Extra Credit Assignment 4

For two assignment points of extra credit:

1. Run `irb` and try ten Ruby expressions with some degree of variety. Explore the `String`, `Array`, `Integer`, `Float`, and `Rational` classes for methods to experiment with.

2. Capture the interaction (both expressions and results) and put it in a plain text file, `eca4.txt`. No need for your name, NetID, etc. in the file. No need to edit out errors.

3. On lectura, turn in `eca4.txt` with the following command:
   
   ```
   % turnin 372-eca4 eca4.txt
   ```

Due: At the start of the next lecture after we hit this slide.
Ruby basics
In Ruby, **every value is an object**.

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

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

>> "ostentatious".tr("aeiou","12345")
 => "4st2nt1t345s"

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

Repeat: In Ruby, **every value is an object**.

What are some values in Java that are not objects? How about Python?
Of course, "everything" includes numbers:

```ruby
>> 1.2.class()
=> Float

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

>> 17**37
=> 3362095853201812742282475234995233875224247377

>> _.pred()
=> 3362095853201812742282475234995233875224247376

>> _.class()
=> Integer
```
Everything is an object, continued

The TAB key shows completions in `irb`:

```
>> 7.<TAB><TAB>

7.bit_length                         7.to_s
7.digits                             7.to_i
7.lcm                                7.to_f
7.gcd                                7.to_r
7.rationalize                        7.div
7.next                               7.divmod
7.gcdlcm                             7.fdiv
7.denominator                        7.coerce
7.numerator                          7.modulo
7.upto                               7.remainder
7.chr                                7.abs
7.inspect                            7.magnitude
7.size                               7.integer?
7.succ                               7.floor
7.ord                                7.ceil
7.to_int                             (119 in all...)
```
Parentheses are often optional in method invocations:

```ruby
>> 1.2.class
=> Float

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

But, the following case fails. (Why?)

```ruby
>> "testing".count "aeiou".class
TypeError (no implicit conversion of Class into String)
```

Solution:

```ruby
>> "testing".count("aeiou").class
=> Integer
```

I usually omit parentheses in simple method invocations.
A post-Haskell hazard!

You might have a Haskell moment and leave out a comma between arguments:

```ruby
>> "testing".slice 2 3
SyntaxError ((irb):14: syntax error, unexpected integer literal, expecting end-of-input)
```

Commamas are required between arguments!

```ruby
>> "testing".slice 2,3
=> "sti"
```

I almost always use parentheses when there's more than one argument:

```ruby
>> "testing".slice(2,3)
=> "sti"
```

Note: **String#slice** makes for a good example but we'll soon see an operator-based slice.
Operators are methods, too

Ruby operators are methods with symbolic names.

In general,

```
expr1 op expr2
```

means

```
expr1.op(expr2)
```

Example:

```
>> 3 + 4
=> 7

>> 3.+(4)
=> 7

>> "abc".==(97.chr.+("bc"))   # Try it w/o parens, too!
=> true
```
The **Kernel module** has methods for I/O and more. Methods in **Kernel** can be invoked with only the method name.

```ruby
>> puts "hello"
hello
=> nil

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

>> puts gets.inspect
testing *(typed by user)*
"testing\n"
=> nil
```

What can say about value, type and side-effect for **puts** and **printf**?

See [ruby-doc.org/core-2.7.0/Kernel.html](http://ruby-doc.org/core-2.7.0/Kernel.html)
A LHtLaL suggestion:
Start accumulating a file of brief notes on Ruby. Example:

```bash
$ cat ~/notes/ruby.txt
#FAQ
https://ruby-doc.org/core-2.7.0/String.html

#irb (REPL)
reads ~/.irbrc
_ is last value, like `it` in Haskell

#misc
Every value is an object
Can often omit parens on methods:
    3.class, "testing".count "t"
Operators are methods: 3+4 is really 3.+(4)

#i/o
gets, puts, printf (in Kernel module)
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:

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

% 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.

There is no evident compilation step or artifact produced. It just runs!
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:

```ruby
# hello2.rb

def say_hello
  puts "Hello, world!"
end

say_hello
```

```bash
% 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

The program below reads lines from standard input and writes each, with a line number, to standard output:

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

Execution:

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

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

Solution:

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

Usage:

```
% cal | ruby tac.rb

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
</tr>
<tr>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>
Su Mo Tu We Th Fr Sa
November 2022
```

ADD SLIDE on pitfall with gets vs. STDIN.gets; show ENOENT error!
Some basic types
The value `nil`

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

```ruby
>> nil
=> nil

>> nil.class
=> NilClass

>> nil.object_id
=> 4
```

We'll see that Ruby uses `nil` in a variety of ways.
Instances of Ruby's `String` class represent character strings.

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

```ruby
>> puts 'newline >\n< and tab >\t<'
newline >
< and tab > <

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

All escapes:

```
docs.ruby-lang.org/en/2.7.0/syntax/literals_rdoc.html#label-Strings
```

Unicode:

```ruby
>> "\u{1F355} \u{2b} \u{1F41F} = \u{1f610}"
=> "を集め + 海 = 😊"
```
String literals, continued

In single-quoted string literals only \' and \" are recognized as escapes:

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

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

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

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

```ruby
>> "".public_methods.sort
```

```ruby
>> "".public_methods.length
=> 183
```
Unlike Python, 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:

```ruby
>> x = "testing"

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

>> y << " this"    # the << operator appends a string

>> y
=> "testing this"

>> x
=> "testing this"
```

Is it a good idea to have mutable strings?
Strings are mutable, continued

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

```ruby
>> x = "testing"
>> x2 = x.dup
=> "testing"

>> x2 << "...more"

>> x2
=> "testing...more"

>> x
=> "testing"
```

Some objects that hold strings `dup` strings they are given.
Sidebar: Applicative vs. imperative methods

Some methods have both an *applicative* and an *imperative* form.

**String's upcase method** is *applicative*—it produces a new **String** but doesn't change its *receiver*, the instance of **String** on which it's called:

```ruby
>> s = "testing"
=> "testing"

>> s.upcase
=> "TESTING"

>> s
=> "testing"
```
In contrast, an *imperative* method potentially changes its receiver.

String's `upcase!` method is the imperative counterpart to `upcase`:

```
>> s.upcase!
=> "TESTING"
```

```
>> s
=> "TESTING"
```

A Ruby **convention**: When methods have both an applicative and an imperative form, the imperative form ends with an exclamation mark.
Strings can be compared with a typical set of operators:

```ruby
>> "apples" == "oranges"
=> false

>> "apples" != "oranges"
=> true

>> "apples" < "Oranges"
=> false
```

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

These (common) operators work with most other types, too!
There is also a *comparison operator*: `<=>`

**Behavior:**

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

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

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

Speculate: How is the operator `<=>` read aloud by some programmers? "spaceship"

What are the Java and C analogs for `<=>` when applied to strings? `String.compareTo` and `strcmp`, respectively.
Substrings

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

```ruby
>> s = "abcd"

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

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

>> s[100]  # An out-of-bounds reference produces nil
=> nil

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

```ruby
>> s = "tack"
>> s[0] = 65.chr
>> s[1] = "tomi"
>> s
=> "Atomick"
>> s[-1] = ""
>> s
=> "Atomic"
```
A substring can be referenced with
\( s[start, length] \)

```ruby
>> s = "replace"
```

```plaintext
replace
0 1 2 3 4 5 6
7 6 5 4 3 2 1 (negative)
```

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

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

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

```ruby
>> s[10,10]
=> nil
```
Instances of Ruby's `Range` class represent a range of values. A `Range` can be used to reference a substring.

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

>> r.class
=> Range

>> s = "replaced"

>> s[r]
=> "place"

>> s[r] = ""

>> s
=> "red"
```
It's more common to use range literals with strings:

```ruby
>> s = "rebuilding"
>> s[2..-4]
=> "build"
```

```ruby
>> s[2...-4]
=> "buil"    # three dots is "up to" a
```

```ruby
>> s[-8..-4]
=> "build"
```

```ruby
>> s[-4..-8]
=> ""
```

```ruby
>> s[2..]
=> "building"    # 2.. is an endless Range
```
A substring can be the target of an assignment:

```ruby
>> s = "replace"
```

```plaintext
replace
0 1 2 3 4 5 6
```

```ruby
>> s[0..2] = ""
```

```plaintext
place
0 1 2 3 4 5 6
```

```ruby
>> s
=> "place"
```

```ruby
>> s[3..-1] = "naria"
```

```plaintext
planaria
0 1 2 3 4 5 6
```

```ruby
>> s
=> "planaria"
```

```ruby
>> s["aria"] = "kton"
```

```plaintext
kton
0 1 2 3 4 5 6
```

```ruby
=> "kton"  # If "aria" appears, replace it (error if not).
```

```ruby
>> s
=> "plankton"
```
In a string literal enclosed with double quotes the sequence `#{expr}` causes interpolation of the value of `expr`, an arbitrary Ruby expression.

```ruby
>> 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 183 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.
Ruby's **Integer** class represents integers of any size.

```ruby
>> 5.class
=> Integer

>> 11 ** 111
=> 3931769528717253549053417338688275670476160766413
5852855034678556753487133293648186980649622260361388
994869790176611

>> (3 ** 33_000).to_s.size
=> 15746

>> (3 ** 33_000_000).bit_length
(irb):115: warning: in a**b, b may be too big
NoMethodError (undefined method `bit_length' for Infinity:Float)
```
The **Float** class represents floating point numbers that can be represented by a double-precision floating point number on the host architecture.

```ruby
>> x = 123.456
=> 123.456

>> x.class
=> Float

>> x ** 0.5
=> 11.111075555498667

>> x = x / 0.0
=> Infinity

>> (0.0/0.0).nan?
=> true
```
Arithmetic on two **Integers** produces an **Integer**.

```ruby
>> 2/3
=> 0
```

```ruby
>> _.class
=> Integer
```

**Integers** and **Floats** can be mixed. The result is a **Float**.

```ruby
>> 10 / 5.1
=> 1.9607843137254903
```

```ruby
>> 10 % 4.5
=> 1.0
```

```ruby
>> _.class
=> Float
```
Ruby has a `Complex` type.

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

>> x * 2 + 7
=> (11+6i)

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

> >> _ * _
=> (-1+0i)
```
There's **Rational**, too.

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

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

>> Rational 0.5
=> (1/2)

>> Rational 0.6
=> (5404319552844595/9007199254740992)

>> Rational 0.015625
=> (1/64)
```
Sidebar: **Fixnum and Bignum—RIP**

With 2.2.4 on lectura, integers in the range \(-2^{62}\) to \(2^{62}-1\) were represented by instances of **Fixnum**. If an operation produces a number outside of that range, the value is represented with a **Bignum**.

\[
\begin{align*}
>> & x = 2**62-1 \\
=> & 4611686018427387903
\end{align*}
\]

\[
\begin{align*}
>> & x.class \\
=& Fixnum
\end{align*}
\]

\[
\begin{align*}
>> & x += 1 \\
=& 4611686018427387904
\end{align*}
\]

\[
\begin{align*}
>> & x.class \\
=& Bignum
\end{align*}
\]

\[
\begin{align*}
>> & x -= 1 \\
=& 4611686018427387903
\end{align*}
\]

\[
\begin{align*}
>> & x.class \\
=& Fixnum
\end{align*}
\]
Unlike some languages, Ruby does not automatically convert strings to numbers and numbers to strings as needed.

```ruby
>> 10 + "20"
TypeError (String can't be coerced into Integer)
```

The methods `to_i`, `to_f`, and `to_s` are used to convert values to `Integers`, `Floats` and `Strings`, respectively.

```ruby
>> 10.to_s + "20" => "1020"
>> 10 + "20".to_f => 30.0
>> 10 + 20.9.to_i => 30
```
A sequence of values is typically by an instance of the **Array** class.

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

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

```ruby
>> a = [1, 'two', 3.0, %w[a b c d]]
=> [1, 'two', 3.0, ['a', 'b', 'c', 'd']]

>> a[0]
=> 1

>> a[1,2]  # a[start, length]
=> ['two', 3.0]

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

>> a[-1][-2]
=> 'c'
```
Elements and subarrays can be assigned to. Ruby accommodates a variety of cases; here are some:

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

Semicolon separates expressions. We make a change and show new value.
What's important to retain from the examples above?
• Elements of arrays and subarrays can be assigned to.
• Lots of rules; some complex.
A variety of other operations are provided for arrays. Here's a sampling:

```ruby
>> a = []

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

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

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

>> a.reverse!; a
=> [[2, 3, 4], 1]
```
A few more:

```ruby
>> 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]
```
Even more!

```ruby
>> 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
```

TODO: show intercalation!
Comparing arrays

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

```ruby
>> [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.

\[
\begin{align*}
&\text{\texttt{>> [1,2,3,4] <=> [1,2,10]}}
\Rightarrow -1 \\
&\text{\texttt{>> [[10,20], [2,30], [5, "x"]].sort}}
\Rightarrow [[2, 30], [5, "x"], [10, 20]]
\end{align*}
\]

Speculate: Will the following work?

\[
\begin{align*}
&\text{\texttt{>> [1,2] < [3,4]}}
\text{NoMethodError (undefined method `<' for [1, 2]:Array)}
\text{Did you mean?  \texttt{<=>}}
\end{align*}
\]
Comparing arrays

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

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

```
>> [[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 Integer with String failed)
```
At hand:

```ruby
>> ['sixty', 20, 'two'].sort
ArgumentError (comparison of Integer with String failed)
```

Contrast with Icon:

```ruby
] [ sort(['sixty', 20, 'two'])
  r := [20, 'sixty', 'two'] (list)
] [ 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?

Here's Python 2:

```python
>>> sorted([3.0, 7, 2, 'a', 'A', ':', [2], [1], -1.0])
[-1.0, 2, 3.0, 7, [1], [2], ':', 'A', 'a']
```
An array can hold a reference to itself:

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

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

>> a.size
=> 4

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

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

Arrays can be cyclic:

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

>> a[-2][-1]
=> 10
```
Control Structures
The **while** loop

Here's a loop to print the integers from 1 through 10, one per line.

```ruby
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.
Java control structures such as **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 a scalar value that is non-zero to be "true".

From my Python notes:

```python
>>> list(map(bool, [False, None, [], {}, '', set(), 0, 0.0]))
[False, False, False, False, False, False, False, False]
```

PHP and JavaScript, too, have sets of "truthy" and "falsy/falsey" 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".

Let's analyze this loop, which reads lines from standard input using `gets`.
```ruby
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 from standard input 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, in turn terminating the loop.
LHtLaL sidebar: Partial vs. full understanding

From the previous slide:

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

Partial understanding:
The loop reads and prints every line from standard input.

Full understanding:
What we worked through on the previous slide.

I think there's merit in full understanding.

More examples of full understanding:
- Knowing exactly how `*p++ = *q++` works in C.
- Knowing the rules for field initialization in Java.
- Knowing exactly when you need to quote shell metacharacters.
- Knowing the full set of truthy/falsy rules for a language.
String's `chomp` method removes a carriage return and/or newline from the end of a string, if present.

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

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

Will it work?

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

What's the problem?
At hand:

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

At end of file, `gets` returns `nil`, producing an error on `gets.chomp`.

Which of the two alternatives below is better?

| result = ""
while line = gets
  line.chomp!
  result += line
end
puts result |
|---|
| result = ""
while line = gets
  result += line.chomp
end
puts result |
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)

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

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

```ruby
while s[0]
  puts s[0]
  s[0] = ""
end
```
Unlike Java, Ruby does pay some attention to the presence of newlines in source code.

For example, a while loop cannot be trivially squashed onto a single line.

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

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

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

There is some middle ground, too:

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

Unlike Haskell and Python, indentation is never significant in Ruby.
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.

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

Is the following ok?

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

Nope...

```
    test.rb:2: syntax error, unexpected <=
    <= 10 do puts i; i += 1 end
```
The incomplete expression rule does have some pitfalls.

Example: Ruby considers

\[
\begin{align*}
x &= a + b \\
    &+ c
\end{align*}
\]

to be two expressions: \( x = a + b \) and \( + c \).

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

\[
\begin{align*}
x &= a + b + \\
    &c
\end{align*}
\]

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:

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

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

```ruby
>> 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:

```ruby
>> 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".
An exclamation mark inverts a logical value. The resulting value is always true or false.

```ruby
>> ! true
=> false
```

```ruby
>> ! 1
=> false
```

```ruby
>> ! nil
=> true
```

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

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

```ruby
>> ![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,

\[ x < 2 \land y > 3 \lor x \times y < 10 \lor z > 20 \]

instead of this:

\[ (x < 2 \land y > 3) \lor (x \times y < 10 \lor z > 20) \]

LHtLaL problem: Devise an example for **!** vs. **not**.
Here is Ruby's if-then-else:

```ruby
>> 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?

```ruby
if 1 > 2 then 3 end
```
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 the C family, `if-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.

Speculate: How does Ruby handle it?
```ruby
>> if 1 > 2 then 3 end
=> nil
```
If there's no `else` clause and the control expression is `false`, `nil` is produced.

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

```ruby
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 "then" at the end of the first line above is optional.

then is not optional in this one-line expression:

```ruby
if 1 then 2 else 3 end
```
The **elsif** clause

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

```ruby
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?
At hand:

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

```ruby
>> 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?
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.

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

Problem: Rewrite the above loop to use **if** as a modifier.

```ruby
while line = gets
  break if line[0] == "."  
  next if line[0] == "#"
  puts line
end
```
Remember that `while` is an expression that by default 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:

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

% ruby break2.rb
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```
Here are three examples of Ruby's `for` loop:

```ruby
for i in 1..100 do
  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 produce a value 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 supports parallel assignment:

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

What's the output?
- 1-1-1-1-1
- sos
- <-> <-> <-> <-> <-> <-> <-> <-> <-> <->

Consider the design choice 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?
Methods and more
Here is a simple Ruby method:

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

The keyword `def` indicates that this is a method definition.

Following `def` is the method name.

The parameter list follows, optionally enclosed in parentheses. **No types can be specified.**

Zero or more expressions follow

`end` terminates the definition.
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`:

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

Do you prefer the above or the below?

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

Which is more like Haskell?
As we saw in an early example, if no arguments are required, the parameter list can be omitted:

```ruby
def hello
  puts "Hello!"
end
```

What does `hello` return?

What does the last expression in `hello` return?

Execution:

```
>> hello
Hello!
=> nil
```

Contrast with Python:

```
>>> hello()
Hello!
>>> ```
Ruby methods in a file can be tested by hand using `irb`.

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

% irb -r ./simple.rb # .rb is not required
>> add 3,4
=> 7
```

If the environment variable `RUBYLIB` is set to ".", the "./" can be omitted:

```bash
% echo $RUBYLIB
.

% irb -r simple
...
```

(Set it with "export RUBYLIB=." in your ~/.bashrc.)
Testing methods with `irb`, continued

A source file can be loaded with `Kernel.load`:

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

```
>> load "simple.rb"
=> true
>> add 3, 4
=> 7
```

How does `load` in Ruby differ from `:load` in `ghci`?

`load "simple.rb"` is simply a Ruby expression that's evaluated by `irb`. Its side-effect is that the specified file is loaded.

How can we take advantage of `load` being a Ruby method?
Let's include in `simple.rb` an "r" method that loads `simple.rb`:

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

def hello # new for example
  puts "Hello!"
end

def r
  load "simple.rb"
end
```

After loading `simple.rb` once with `irb`, we can edit in a different window and then reload `simple.rb` by just typing `r`.

```
$ irb -r ./simple.rb
>> add 3,4
=> 7
>> hello
Hello!
[...edit simple.rb...]
>> r
=> true
>> add 3,4
add called...
=> 7
>> hello
Hello! (v2)
```
I claim to be defining methods `add` and `hello` but there's no class in sight!

A freestanding method 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.

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

```ruby
>> methods_b4 = self.private_methods
>> load "simple.rb"

>> self.private_methods - methods_b4
=> [:r, :add, :hello]
```

`sself` has three more private methods after loading `simple.rb`.

Methods can be added to a class at run-time in Ruby!
Where's the class, continued?

We'll later see how to define classes but our initial "mode" on the Ruby assignments will be writing programs in terms of top-level methods.

This is essentially procedural programming with an object-oriented library.
Default values for arguments

Imagine a **wrap** method that can be used in three ways:

```ruby
>> wrap "abc"
=> "(abc)"

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

>> wrap _, "|"
=> "|<abc>|"
```

Ruby allows default values to be specified for a method's arguments:

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

Lots of library methods use default arguments.

```ruby
>> "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:

```ruby
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!)

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

>> wrap("testing", "[ ]") => "[testing]"
```
Different languages approach overloading and default arguments in various ways. Here's a sampling:

- **Java**: Overloading; no default arguments
- **Python**: No overloading; default arguments
- **C++**: Overloading and 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:

```icon
procedure wrap(s, wrapper)
    /wrapper := "()" # if wrapper is &null, assign "()" to wrapper
    return wrapper[1] || s || wrapper[-1]
end
```
Java's `String.format` and Python's `print` can accept any number of arguments.

Imagine a Ruby method that can be called with any number of arguments:

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

Implementation:
```ruby
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: If a parameter is prefixed with an asterisk, an array is made of all following arguments.
Problem: Write a method `format` that interpolates argument values into a string where percent signs are found.

```ruby
>> 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.

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

```ruby
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
```

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 it: Shuffle around the three definitions and "main" to see what works and what doesn't.
I'd like to load the following file and then test `showline`, but loading it in `irb` seems to hang. Why?

```
% cat main3.rb
def showline s
  puts "Line: #{s.inspect} (#{s.size} chars)"
end
def main
  while line = gets; showline line; end
end
main
```

```
% irb
>> load "main3.rb"
...no output or >> prompt after the load...
```

It's waiting for input! After the `defs` for `showline` and `main`, `main` is called. `main` does a `gets`, and that `gets` is waiting for input.

TODO: `irb -r ./main3.rb`
(next slide, too)
Testing methods when there's a "main", cont.

Here's a technique that lets a program run normally with **ruby** but **not** run **main** when loaded with **irb**:

```ruby
% cat main3a.rb
def showline s
  puts "Line: #{s.inspect} (#{s.size} chars)"
end
def main
  while line = gets; showline line; end
end
main unless $0 == "irb"
```

Call **main** unless the name of the program being run is **"irb"**.

Now I can test methods by hand in **irb** but still do **ruby main3.rb** ...

% **irb**
>> load "main3a.rb"
>> showline "testing"
Line: "testing" (7 chars)
>> main
(waits for input)
Global variables

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

```ruby
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. Prints $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
Ruby has a number of predefined global variables:

```ruby
>> puts global_variables.sort * "","
$!, $", $$, $&, $', $*, $+, $., $-0, $-F, $-I, $-K, $-W, $-a, $-d, $-i, $-l, $-p, $-v, $-w, $., $/, $0, $1, ..., $9, $:, $;,$<, $=, $>, $?, $@, $DEBUG, $FILENAME, $KCODE, $LOADED_FEATURES, $LOAD_PATH, $PROGRAM_NAME, $SAFE, $VERBOSE, $\, $_, $`, $binding, $stderr, $stdin, $stdout, $~
```

A few:

- **$0**  Program name  (>> $0 => "irb")
- **$,** Used by `print` and `Array.join`.  (>> $, = "-"; [1,2].join=> "1-2")
- **$:** Directories where `load` and `require` look for files.  (Try it!)
- **$_**  Last string read by `Kernel.gets` and `Kernel.readline`.  (Try it!)

More: [ruby-doc.org/core-2.7.0/doc/globals_rdoc.html](http://ruby-doc.org/core-2.7.0/doc/globals_rdoc.html).
Wikipedia says,

"In computer programming, a sigil is a symbol attached to a variable name, showing the variable's datatype or scope, usually a prefix, as in $\text{foo}$, where $\$\$ is the sigil.

"Sigil, from the Latin sigillum, meaning a 'little sign', means a sign or image supposedly having magical power.'"

"In 1999 Philip Gwyn adopted the term 'to mean the funny character at the front of a Perl variable'."

Are reverse! and nan? examples of sigils?

Lots more: https://en.wikipedia.org/wiki/Sigil_(computer_programming)
Local variables are local

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

% cat scope1.rb
x = 10

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

def f
  puts "in f: x = #{x}"
end

f

% ruby scope1.rb

Top-level: x = 10
scope1.rb:6:in 'f': undefined local variable or method `x' for main:Object (NameError)
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.

```ruby
>> MAX_ITEMS = 100
```

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

```ruby
>> MAX_ITEMS = 200
(lood):9: warning: already initialized constant MAX_ITEMS
(lood):8: warning: previous definition of MAX_ITEMS was here
=> 200
```

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

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

>> L.push 30
=> [10, 20, 30]
```
Like globals, constants can be accessed in all methods.

```ruby
% cat constant1.rb
MAX_LEVELS = 1000

def f
  puts "f: max levels = #{MAX_LEVELS}"
end

def f
  puts "top-level: max levels = #{MAX_LEVELS}"
end

% ruby constant1.rb
f: max levels = 1000
top-level: max levels = 1000
```

Find out: Can a constant be created in a method?
Pitfall: If a method is given a name that begins with a capital letter, it compiles ok but it can't be run!

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

>> Hello
NameError: uninitialized constant Hello
```
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).
Type Checking
"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 17 Edition*

Example: assume the following...
```java
int i = ...;  String s = ...;  Object o = ...;  static long f(int n);
```

What are the types of the following expressions?

- `i + 5`
- `i + s`
- `s + o`
- `o + o`
- `o.hashCode()`
- `f(i.hashCode()) + 3F`
- `i = i + s`

Did we need to know any values or execute any code to know those types? No. Our analysis only required types and operations.
If there's a type error in a Java program, `javac` doesn't produce a `.class` file—there's nothing to run.

```
% cat X.java
public class X {
    public static void main(String args[]) {
        System.out.println("Running!");
        System.out.println(3 + new int[5]);
    }
}
```

```
% javac X.java
X.java:4: error: bad operand types for binary operator '+'
        System.out.println(3 + new int[5]);
        ^
```

```
% java X
Error: Could not find or load main class X
• The file X.class wasn't created.
• What would a similar Python program do?
Java does type checking based on the declared types of variables and the intrinsic types of literals.

Haskell supports type declarations but we know that it also provides type inferencing:

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

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

- The type for all expressions is determined when a body of code is compiled/loaded/etc.

- All* type inconsistencies that exist are discovered at that time.

- Execution “not allowed” if errors exist.
  - Sometimes manifested by no "executable" file being produced.
  - With ghci, functions in a file with an error aren't loaded.
Important:
A statically typed language lets us guarantee that certain kinds of errors don't exist without having to run any code.

If a Java program compiles, we can be absolutely sure no errors of the following sort (and more) exist:

- Dividing a string by a float
- Concatenating a number with a list
- Subscripting a value that's not an array
- Invoking a method with the wrong number of arguments
- Forgetting a `return` at the end of a method that returns a value
- Misspelling the name of a method (true?)

Exception: Errors due to casts and boxing/unboxing can still exist!
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?
   With others?

Paul Hudak of Yale wrote,
"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."

Do you agree with Hudak?
In Java, variables are declared to have a type.

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

```ruby
>> x = 10
>> x.class  # What's the class of the object held by x?
=> Integer

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

>> x = 2**100
>> x.class
=> Integer
```
Array's `sample` method returns a random element of the receiver.

```ruby
>> a = ["one", 2, [3], 4.0]
>> a.sample #=> [3]
>> a.sample #=> "one"
```

What's the type of `a.sample + a.sample`?

```ruby
>> (a.sample + a.sample).class
=> String

>> (a.sample + a.sample).class
=> Float
```

The type of `a.sample`, `a.sample + a.sample`, `a.sample[0]`, etc. is unknown until the expression is evaluated!
Ruby is a dynamically typed language.

Consider this Ruby method:
```ruby
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 a dynamically typed language, types are not checked until an expression is evaluated.

Another way to say it:
There is no static analysis of the types in Ruby expressions.
Dynamic typing, continued

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

Consider this Ruby code:

```ruby
while line = gets
  puts(line[-x] + y)
end
```

What types must be checked each time through that loop?

- Is `gets` a method or a value?
- Can `x` be negated?
- Is `line` subscriptable with `-x`?
- Can `line[-x]` and `y` be added?
- Is `puts` a method?
Performance implications with dynamic typing

One performance cost with a dynamically typed language is execution-time type checking.

Evaluating the expression \( x + y \) might require decision-making like this:

- if both \( x \) and \( y \) are integers
  - add them
- else if both \( x \) and \( y \) are floats
  - add them
- else if one is a float and the other an integer
  - convert the integer to a float and add them
- else if both \( x \) and \( y \) are strings
  - concatenate them

...and more...

If that \( x + y \) is in a loop, that decision making is done every time around.

Note: The above "implementation" can be improved upon in many ways.
In contrast, consider this Java method:

```java
int count(int wanted, int[] values) {
    int result = 0;
    for (int value: values)
        if (value == wanted)
            result += 1;
    return result;
}
```

Generated virtual machine code:

```
0:  iconst_0
1:  istore_2
2:  aload_1
3:  astore_3
4:  aload_3
5:  arraylength
6:  istore 4
8:  iconst_0
9:  istore 5
11: iload 5
13: iload 4
15: if_icmpge 39
18: aload_3
19: iload 5
21: iaload
22: istore 6
24: iload 6
26: iload_0
27: if_icmpne 33
30: iinc 2, 1
33: iinc 5, 1
36: goto 11
39: iload_2
40: ireturn
```

See also [https://en.wikipedia.org/wiki/Java_bytecode_instruction_listings](https://en.wikipedia.org/wiki/Java_bytecode_instruction_listings)
With respect to static typing, what are the implications of dynamic typing for...

Loading ("compilation") speed?
Probably faster.

Execution speed?
Probably slower.

The likelihood of code to be correct?
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 does the user of software care about?

- Dynamic vs. static typing? Ruby vs. Java?
  - No!

- Software that works
  - Facebook game vs. radiation therapy system

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

- Better sooner than later
  - "First to volume" can be the key to success for a company.
  - A demo that's a day late for a trade show isn't worth much.

- Affordable
  - "People will pay to stop the pain." – Doug Higgins
  - I'd pay a LOT for a great system for writing slides.
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 never detected.

Ruby, Python, and Icon have no static type checking whatsoever, but type errors during execution are always 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.
Sidebar: Interpreted or compiled?

A common misunderstanding:
Python and Ruby are interpreted languages.
Java and C are compiled languages.

The fact:
Interpretation and compilation are attributes of an implementation of a language, not the language itself!

A simple, polarized viewpoint:
• Interpreters execute source code as-is.
• Compilers translate source code into machine code.

Reality:
• Language implementations use a variety of techniques that each fall along a broad spectrum from interpretation to compilation.
• A particular implementation of any language can be made to fall at either end of that spectrum, or anywhere in the middle.
REPLACEMENTS!
Discard 133-157 in the prior set!

Duck Typing
Duck typing

Definition from Wikipedia (c.2015):

*Duck typing* is a style of typing in which an object's methods and properties determine the valid semantics, rather than its inheritance from a particular class or implementation of an explicit interface.

Two examples of Ruby's `for` loop:

```ruby
for i in 1..20 do ...statements... end

for word in line.split do ...statements... end
```

`for` only requires the "in" expression to produce an object that has an `each` method.

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.*

The value produced by the "in" expression qualifies as a "duck" if it has an `each` method.
Duck typing, continued

For reference:

*Duck typing* is a style of typing in which an object's methods and properties determine the valid semantics, rather than its inheritance from a particular class or implementation of an explicit interface.

—Wikipedia (c.2015)

- Duck typing is both a technique and a mindset.
- Ruby both facilitates and uses duck typing.
- We don't say Ruby is duck typed. We say that Ruby allows duck typing.
Duck typing, continued

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

Consider this method:

```ruby
def double x
  x * 2  # think of this as x.*(2)
end
```

What operation(s) does `double` require that `x` support?
- `x` must have a `.*(n)` method that works for `n = 2`. 
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

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

• Is double polymorphic?

For reference:

```ruby
def double x
  x * 2
end
```
Here's `wrap` from slide 98. What does it require of `s` and `wrapper`?

```ruby
def wrap s, wrapper = "()"
    wrapper[0] + s + wrapper[-1]
end
```

```ruby
>> wrap "test", "<>"
=> "<test>"
```

Will the following work?

```ruby
>> wrap "test", ["<<<",">>>"]
=> "<<<test>>>"
```

```ruby
>> wrap [1,2,3], [["..."]]  
=> ["...", 1, 2, 3, "..."]
```

```ruby
>> wrap 10,3
=> 11
```
Should a method check the type of parameters?

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

What does the following suggest?

```ruby
>> double(Complex 'i')
RuntimeError (Can't double a Complex!)
```

- It's easy to forget a type that should be allowed!
- What about types added later that do handle .*(2)?

Bottom line: Checking for types is the antithesis of duck typing.
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.

Does the following Java method exemplify duck typing?

```java
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
In Ruby, an iterator is a method that can invoke a block.

One of the many iterators in the Array class is each. Example:

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

What's the block in this case?

How many times is the block invoked?

What is |e|? Is it required?

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

However, with some iterators the value returned is of principle interest.

See if you can predict what the following iterators do.

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

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

```
>> [5,4,10,7].inject([[[0,0]]]) { |a, e| a + [[e, a[-1][1]+e]]}
=> [[[0, 0], [5, 5], [4, 9], [10, 19], [7, 26]]
```

What do these remind you of?
We can use `[30, 20, 10, 40].sort` to sort an array but we can also call `Array#sort` with a block:

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

Speculate: What values is `sort` passing to the block?

```ruby
>> [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`?
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 does `Array#sort` do if not given a block?
```
>> "a longer try first".split.sort
=> ["a", "first", "longer", "try"]
```

TODO: maybe a good example would be to make `[3,"5"]` .sort work
We can query the "ancestors" of a class like this:

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

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

**Enumerable** has a number of iterators. Here are some:

```ruby
>> [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
```
At hand:

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

Another Enumerable iterator is max:

```ruby
>> ["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-2.7.0/Enumerable.html
 Numeric iterators

```ruby
>> 3.times { puts "Ding!" }
Ding!
Ding!
Ding!
=> 3

>> 3.downto(1) { |i| puts i }
3
2
1
=> 3

>> 1.step(2, Rational(1,3)) { |i| puts i }
1
4/3
5/3
2/1
=> 1
```
In Ruby 1.9, **String** lost its `each` method!

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

Use `each_char` instead:

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

Any ideas about the behavior of the following?

```ruby
>> i = 0
>> "Mississippi".gsub("i") { (i += 1).to_s }
=> "M1ss2ss3pp4"
```
Sidebar: Iterate with `each` or use a `for` loop?

Which is better? This,

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

or this?

```ruby
"x".methods.each { |name| puts name if name.to_s.include? "!" }
```
The "do" syntax for blocks

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

```ruby
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.
sumnums.rb computes some simple statistics for lines of zero or more integers read from standard input:

```bash
$ 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
```

Let's write it! Notes:
- Use nested iterators/blocks. (Don't use *for* !)
- Kernel.readlines returns an array of all lines of standard input:
  ```ruby
  readlines => [
    "5 10 0 50\n", " 200\n", "1 2 3 4 5 6 7 8...\n"]
  
  " 10  20   30\n".split => ["10", "20", "30"]
  
  "10".to_i => 10
  ```
One solution:

```ruby
total = n = 0
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
```

Or, if you can't let go of Haskell...

```haskell
nums = STDIN.read.split.map { |s| s.to_i }
total = nums.inject(0) { |sum,e| sum + e }
n = nums.size
printf(...) if n != 0
```
Scoping rules with blocks

If a variable isn't created before a block, it is local to the block.

```
[7,1,3].each { |e| x = e * 2; ...}
puts x                # x is undefined here!
```

If a variable is created before a block, references inside the block apply to that outer variable.

```
x = 5
[7,1,3].each { |e| x = e * 2; ...}
puts x                # prints 6
```

We can follow a block's parameter(s) with a list of block-local variables.

```
x = 5; y = 6
[7,1,3].each { |e; x, y| ... x = 10; y = 20; ...} # best practice
puts [x,y]        # prints [5,6]
```

You can try the above cases and more with `blockscope2.rb`. 
Various types of iteration side-by-side

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

Both invoke blocks with each element in turn for side-effect(s).
Result of `each` uninteresting.

```ruby
>> [10,20,30].map { |x| x * 2 } #=> [20, 40, 60]
Invokes block with each element in turn and returns array of block results.
```

```ruby
>> [1,2,4,5].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.
```

```ruby
>> [[1,2], "a", [3], "four"]).select { |v| v.size == 1 } #=> ["a", [3]]
Invokes block to determine membership in final result.
```

```ruby
>> "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
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 any 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
```
At hand:

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

Subscripting with a key fetches the associated value.

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

What will happen with a non-existent key?

```ruby
>> numbers["three"]
=> nil
```
At hand:

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

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

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

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

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

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

Here's the literal syntax with key/value pairs:

```ruby
>> h = {"i" => 7, "a" => [3,1,5], "s" => "test"}
```

The value associated with a key can be changed in various ways:

```ruby
>> h["i"] += 3

>> h["a"].unshift 11

>> h["s"] << "er"

>> h
  => {"i"=>10, "a"=>[11, 3, 1, 5], "s"=>"tester"}
```
Python:

```python
>>> d = {}

>>> d[[10,20,30]] = 3
TypeError: unhashable type: 'list'

>>> d[d] = d
TypeError: unhashable type: 'dict'
```

Ruby:

```ruby
>> h = {}

>> h[[10,20,30]] = 3

>> h[h] = h  # (Ralph Griswold felt strongly that this should work!)

>> h
=> {
  [10, 20, 30] => 3,
  {...} => {...}
}
```
A hash to work with:

```ruby
h = {"a" => [10], 2 => [3,5,3], true => [2,5]}
```

Let's iterate over the key/value pairs in `h`:

```ruby
h.each { |k,v| puts "k=#{k}, v=#{v}" }
```

- `k=a, v=[10]`
- `k=2, v=[3, 5, 3]`
- `k=true, v=[2, 5]`

=> `{"a"=>[10], 2=>[3, 5, 3], true=>[2, 5]}`

Here's `select` on a Hash:

```ruby
s = h.select { |k,v| v.size.odd? }
```

=> `{"a"=>[10], 2=>[3, 5, 3]}`
Default values

An earlier simplification: If a key is not found, \texttt{nil} is returned.
Full detail: If a key is not found, the \textit{default value} of the hash is returned.

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

\begin{verbatim}
>> 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!"
\end{verbatim}
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? (Don't peek!)
Solution:

```ruby
# 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
```

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

Contrast with `while`/`for` vs. iterators:
```ruby
counts = Hash.new(0)
while line = gets do
  for word in line.split do
    counts[word] += 1
  end
end

# Like puts counts.inspect
p counts
```
The output of `tally.rb` is not customer-ready!

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

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

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

Problem: Produce nicely labeled output, like this:

<table>
<thead>
<tr>
<th>Word</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>be</td>
<td>2</td>
</tr>
<tr>
<td>not</td>
<td>1</td>
</tr>
<tr>
<td>or</td>
<td>1</td>
</tr>
<tr>
<td>to</td>
<td>2</td>
</tr>
</tbody>
</table>
At hand:

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

Solution:

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

Notes:

- The minus in the format `%–7s` left-justifies, in a field of width seven.
- As a shortcut for easy alignment, the column headers are put at the start of the array, as a fake key/value pair.
- 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

- By default, key/value pairs are sorted by keys.

- We can override that behavior with a block.

- The block is repeatedly invoked with two key/value pairs, like 

  ```ruby
  ["be", 2] and ["or", 1].
  ```

How does the following block affect the sort?

```ruby
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]]
```

Words with the same count are alphabetically ordered.
Let's turn `tally.rb` into a cross-reference program:

```ruby
% cat xref.1
to be or
not to be is not
to be the question

% ruby xref.rb < xref.1
readlines.each do
  |line|
  line.split(" ").each do
    |word|
    counts[word] += 1
  end
end
```

How can we approach it? (Don't peek!)
Changes:
- Use `each_with_index` to get line numbers (0-based).
- Turn `counts` into `refs`, a `Hash` whose values are arrays.
- For each `word` on a line...
  - If `word` hasn't been seen, add a key/value pair with `word` and an empty array.
  - Add the current line number to `refs[word]`

Revised:

```ruby
refs = {}
readlines.each_with_index do
  |line, num|
  line.split(" ").each do
    |word|
    refs[word] = [] unless refs.member? word
    refs[word] << num unless refs[word].member? num
  end
end
```
If we add "p refs" after that loop, here's what we see:

```
% cat xref.1
 to be or
 not to be is not
 to be the question
```

```
% ruby xref.rb < xref.1
{"to=>[0, 1, 2], "be=>[0, 1, 2], "or=>[0], "not=>[1],
 "is=>[1], "the=>[2], "question=>[2]"
```

We want:

```
% ruby xref.rb < xref.1
 Word       Lines
 be         1, 2, 3
 is         2
 not        2
 ...
At hand:

```
{"to"=>[0, 1, 2], "be"=>[0, 1, 2], "or"=>[0], "not"=>[1], ...}
```

We want:

<table>
<thead>
<tr>
<th>Word</th>
<th>Lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>be</td>
<td>1, 2, 3</td>
</tr>
</tbody>
</table>

...  

Let's get fancy and size the "Word" column based on the largest word:

```
max_len = refs.map { |k,v| k.size}.max
fmt = "%-#{max_len}s %s\n"  # Ex: "%-9s %s\n"
```

```
print fmt % ["Word", "Lines"]
refs.sort.each do
  |k,v|
  printf(fmt, k, v.map { |n| n+1} * ", ", )
end
```
Let's build a hash where the keys are word lengths and the values are lists of words with that length.

For the words
   "it", "and", "or"
we'd want
   \{2=>["it", "or"], 3=>["and"]\}

First attempt:
   >> h = Hash.new []  # Our hope: The initial value for each
      # key will be an empty list.

   >> ["it", "and", "or"].each { |w| h[h.size] <<= w }

   >> h
   => {}
A Hash.new pitfall, continued

Solution:

```ruby
>> h = Hash.new { |h,k| h[k] = [] }

>> ["it", "and", "or"].each { |w| h[w.size] << w }

>> h
=> {2=>["it", "or"], 3=>["and"]}
```

If Hash.new is called with a block, that block is invoked when a non-existent key is accessed.

The block is passed the Hash and the key.

What does the block above do when a key doesn't exist?

It adds a key/value pair that associates the key with a new, empty array.

Challenge: Revise xref.rb to take advantage of this behavior.
Defining classes
Imagine a class named **Counter** that models a tally counter.

Here's how we might create and interact with an instance of Counter:

```ruby
cl = Counter.new
cl.click
cl.click

puts cl   # Output: Counter's count is 2
cl.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`:
```ruby
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.
```ruby
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 @. (A sigil!)

An instance variable comes into existence when it is assigned to. The code above creates @count and @label.

Note: There are no instance variable declarations!

Just like Java, each object has its own copy of instance variables.
Problem: Fill in `click` and `reset` methods.

```ruby
class Counter
  def initialize(label = "Counter")
    @count = 0
    @label = label
  end

  def click
    @count += 1
  end

  def reset
    @count = 0
  end
end
```
In Ruby:
- Only an object's methods can access its instance variables.
  (An object's instance variables cannot be accessed by any other object.)

Problem: Implement `count` (gets the count) and `to_s` for `Counter`:

```ruby
>> c1 = Counter.new "c1"
>> c1.count #=> 0
>> c1.click
>> c1.to_s #=> "c1's count is 1"
```

Solutions:
```ruby
def count
  @count
end

def to_s
  "#{@label}'s count is #{@count}"
end
```
Full source for **Counter** thus far:

```ruby
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
    "#{@label}'s count is #{@count}
  end
end
```

Common error: omitting an instance variable's @ sigil.
An interesting thing about instance variables

Consider this class:  (instvar.rb)

```ruby
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?

```ruby
>> X.new 1 => #<X:0x00000101176838 @x=1>
>> X.new 2 => #<X:0x00000101174970 @y=1>
>> X.new 3 => #<X:0x000001011aaa0 @x=1, @y=1>
```

Instances of a class can have differing sets of instance variables!
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.

```ruby
>> 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)

```ruby
>> class Counter
>>   def label; @label; end
>> end

>> c.label => "ctr 1"
```

What's an implication of this capability?

*We can add methods to classes written by others!*
Icon's 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!

There's no unary ? to overload in Ruby. Instead we'll add a `rand` method to `Integer` and `String`.

If we call `Kernel.rand` with a `Integer` `n` it returns a random `Integer` `r` such that $0 \leq r < n$. 
Here is `random.rb`:

```ruby
class Integer
    def rand
        Kernel.rand(self)+1
    end
end

class String
    def rand
        self[size.rand-1]  # Uses Integer.rand
    end
end
```

>> load "random.rb"
>> 12.times { print 6.rand, " " }
2 1 2 4 2 1 4 3 4 4 6 3

>> 8.times { print "HT".rand, " " }
H H T H T T H H
An interesting thing about class definitions

Observe the following. What does it suggest to you?

```ruby
>> class X
>> end
=> nil
```

```ruby
>> p (class Y; end)
nil
=> nil
```

```ruby
>> 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 defs for methods.

```ruby
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
```

Use:

```ruby
>> 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:...>
```
Kernel.eval parses a string containing Ruby source code and executes it.

```ruby
>> s = "abc"; n = 3

>> eval "x = s * n"
=> "abcabcabc"

>> x
=> "abcabcabc"

>> eval "x[2..-2].length"  #=> 6

>> eval gets
s.reverse
=> "cba"
```

Two of several details about eval and scoping:
- eval uses variables from the current scope.
- An assignment to x is reflected in the current scope.
With `eval` we could write a method that adds methods to `Integer`:

```ruby
>> units("foot/feet=1,yard(s)=3,mile(s)=5280")

>> 10.yards.in_feet
=> 30.0

>> units("second(s)=1,minute(s)=60,hour(s)=3600,day(s)=86400")

>> (1.day + 9.hours + 45.minutes).in_days
=> 1.40625
```

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`.

```ruby
>> 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.add(3,4)
=> 7
>> x.last subroutine
=> "q"
```
Here is `mk_methods.rb`. Note that the body of the class is a `while` loop.

```ruby
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?
Does `eval` pose any risks?

```ruby
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")
Sun Mar 25 23:42:58 MST 2018
? system("rm -rf ...")
...
? system("chmod 777 ...")
...
```
At hand:

```ruby
% ruby eval1.rb
? system("rm -rf ...")
...
? system("chmod 777 ...")
...
```

But, we can do those things without using Ruby!

**eval** gets risky when we can't trust the source of the data. Examples:

- A calculator web app calls **eval** with the user's input. (Bonehead!)
- A friend with a compromised system sends us a data file. (Subtle!)

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.
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.

In Ruby a variable prefixed with two at-signs is a class variable.

Here is `Counter` augmented with a class variable that keeps track of how many counters have been created.

```ruby
class Counter
  @@created = 0 # Must precede any use of @@created
  def initialize(label = "Counter")
    @count, @label = 0, label
    @@created += 1
  end
end
```

Note: Unaffected methods are not shown.
To define a class method, simply prefix the method name with the name of the class:

```ruby
class Counter
  @@created = 0
  ...
  def Counter.created
    @@created
  end
end
```

Usage:

```ruby
>> 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`.

```ruby
class X
  def f; puts "in f"; g end  # Note: calls g
  private
  def g; puts "in g" end
end
```

Usage:

```ruby
>> x = X.new
>> x.f
in f
>> x.g
NoMethodError: private method `g' ...
```

Speculate: What are `private` and `public`? Keywords? Methods in Module! (Module is an ancestor of Class.)
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.

```ruby
def count= n
  puts "count=(#{n}) called"  # Just for observation (LHtLAL)
  @count = n unless n < 0
end
```

Usage:

```ruby
>> c = Counter.new
>> c.count = 10
  count=(10) called
>> c
=> Counter's count is 10
```
Here's a class to represent points on a Cartesian plane:

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

Usage:
```ruby
>> p1 = Point.new(3,4)  #=> #<Point:0x..193320 @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` creates getter methods.

Here's an equivalent definition of `Point`:
```ruby
class Point
  def initialize(x, y)
    @x = x
    @y = y
  end
  attr_reader :x, :y  # :x is a Symbol. "x" would work.
end
```

Usage:
```ruby
>> p = Point.new(3,4)
>> p.x               #=> 3
>> p.x = 10
NoMethodError: undefined method `x=' for #<Point: ...>
```

Why does `p.x = 10` fail?
If you want both getters and setters, use **attr_accessor**.

```ruby
class Point
  def initialize(x, y)
    @x = x
    @y = y
  end
  attr_accessor :x, :y
end
```

Usage:

```ruby
>> p = Point.new(3,4)
>> p.x
=> 3
>> p.y = 10
```

It's important to appreciate that **attr_reader** and **attr_accessor** are methods that create methods. (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 numeric 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

Icon:  
3 + "4" is 7 (addition); 3 || "4" is "34" (concatenation)
We've seen that Ruby operators can be expressed as method calls:

\[ 3 + 4 \text{ is } 3.+(4) \]

Here's what subscripting means:

"abc"[2] is "abc".[](2)
"testing"[2,3] is "testing".[](2,3)

Unary operators are indicated by adding @ after the operator:

-5 is 5.-@()

!"abc" is "abc".!@()

Challenge:

Find a binary operation that can't be expressed as a method call.
We'll use a dimensions-only rectangle class to study overloading in Ruby:

```ruby
class Rectangle
  def initialize(w,h)
    @width, @height = w, h
  end

  attr_reader :width, :height

  def area; width * height; end

  def inspect                          # irb uses inspect
    "#{width} x #{height} Rectangle"
  end
end
```

Usage:

```ruby
  >> r = Rectangle.new(3,4) => 3 x 4 Rectangle
  >> r.area              => 12
  >> r.width             => 3
```
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
```
Operator overloading, continued

Our vision:

```ruby
>> a = Rectangle.new(3,4); b = Rectangle.new(5,6)
>> a + b => 8 x 10 Rectangle
```

Here's how to make it so:

```ruby
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.
For reference:
    def + rhs
        Rectangle.new(self.width + rhs.width, self.height + rhs.height)
    end

Here is a **faulty implementation** of rectangle addition:
    def + rhs
        @width += rhs.width;
        @height += rhs.height
    end

Let's try it...
    >> 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 overloaded operator.

Here is a method for Rectangle that defines unary minus to be imperative "rotation" (a clear violation of the Principle of Least Astonishment!)

```ruby
def -@      # Note: @ suffix to indicate unary form of -
    @width, @height = @height, @width
    self
end
```

```ruby
>> a = Rectangle.new(2,5) => 2 x 5 Rectangle
>> -a                  => 5 x 2 Rectangle
>> a                  => 5 x 2 Rectangle
```

Goofy?
Operator overloading, continued

At hand:
```ruby
def -@
    @width, @height = @height, @width
    self
end
```

How could we get more sensible behavior, like the following?

```ruby
>> a = Rectangle.new(5,2) => 5 x 2 Rectangle
>> -a                       => 2 x 5 Rectangle
>> a                        => 5 x 2 Rectangle
>> a + -a                   => 7 x 7 Rectangle
```

Solution:
```ruby
def -@
    Rectangle.new(height, width)
end
```
Problem: Implement "scaling" a rectangle by some factor. Example:

```ruby
>> 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
```

Solution:
```ruby
def * rhs
    Rectangle.new(self.width * rhs, self.height * rhs)
end
```

What does the following do?

```ruby
>> 3 * Rectangle.new(3,4)
TypeError: Rectangle can't be coerced into Integer
```

What's wrong?

We've implemented only `Rectangle * rhs`
Imagine a case where it's useful to reference width and height uniformly, via subscripts:

\[
\begin{align*}
\text{a} &= \text{Rectangle.new}(3,4) \quad \Rightarrow 3 \times 4 \text{ Rectangle} \\
\text{a}[0] &\Rightarrow 3 \\
\text{a}[1] &\Rightarrow 4 \\
\text{a}[2] &\text{RuntimeError: out of bounds}
\end{align*}
\]

Note that \(a[n]\) is \(a.[\ ](n)\)

Implementation:

```ruby
def [] n
  case n
  when 0 then width
  when 1 then height
  else raise "out of bounds"
  end
end
```

CSC 372 Fall 2022, Ruby Slide 210
A language is considered to be extensible 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 \( a = b + c \times 2 \) with Rectangles look like in Java?
   Maybe: \( \text{Rectangle } a = b.\text{plus}(c.\text{times}(2)) \);

How about in C?
   Would \( \text{Rectangle } a = \text{rectPlus}(b, \text{rectTimes}(c, 2)) \); be workable?

Haskell goes further with extensibility, allowing new operators to be defined.
Ruby is mutable

Ruby is not only extensible; it is also mutable—we can change the meaning of expressions.

If we wanted to be sure that a program never used integer addition, we could do this:

```ruby
class Integer
  def + x
    raise "boom!"
  end
end
```

Contrast:
- C++ is extensible, but not mutable.
- 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:

```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
    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.
class Circle < Shape
  def initialize(label, radius)
    super(label)
    @radius = radius
  end

  attr_reader :radius

  def area
    Math::PI * radius * radius
  end

  def perimeter
    Math::PI * radius * 2
  end

  def inspect
    "Circle #{label} (r = #{radius})"
  end
end
Inheritance in Ruby has a lot of behavioral overlap with Java:

- Subclasses inherit superclass methods.
- Methods in a subclass can call superclass methods.
- Methods in a subclass override superclass methods of the same name.
- Calls to a method \( f \) resolve to \( f \) in the most-subclassed (most-extended) class.

There are differences, too:

- Subclass methods can always access superclass fields.
- Superclass constructors aren't automatically invoked when creating an instance of a subclass.
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:

```ruby
class Shape
  def area
    raise "Shape#area is abstract"
  end
end
```

There is also an `abstract_method "gem"` (a package of code and more):

```ruby
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:

```java
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:

```ruby
def sumOfAreas(shapes)
    area = 0.0
    for shape in shapes do
        area += shape.area
    end
    area
end
```

Does it rely on inheritance in any way?

Even simpler:

```ruby
sum = shapes.inject (0.0) { |acc, shape| acc + 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.

Some common design patterns are simply patterns of working with inheritance hierarchies in statically typed languages.
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 `[start, len]`, and `to_s`.

```ruby
>> r1 = ReplString.new("abc", 2) #=> ReplString(6)

>> r1.size #=> 6

>> r1[0] #=> "a"

>> r1[10] #=> nil

>> r1[2,3] #=> "cab"

>> r1.to_s #=> "abcabc"
```
A **MirrorString** represents a string concatenated with a reversed copy of itself.

```ruby
>> m1 = MirrorString.new("abcdef")
=> MirrorString(12)

>> m1.to_s
=> "abcdefedcba"

>> m1.size
=> 12

>> m1[3,6]
=> "deffed"
```
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
New requirements:
  A VString can be created using either a VString or a String. 
  A ReplString can have a very large replication count.

Will VStrings in constructors work with the implementation 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", 2_000_000_000_000) do?
Here's some behavior that we'd like to see:

```ruby
>>> 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",200000000000000))

>>> s2.size  # => 12000000000000

>>> s2[-1]   # => "a"

>>> s2[s2.size/2 - 3, 6] => "abccba"
```
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 arbitrarily large.
- If `vs` is a `VString`, `vs[pos]` and `vs[pos,len]` produce `Strings`.
- `VString#size` works.
- `VString#to_s` "works" but is problematic with long strings.

How can we make this work?
Let's play computer!

```ruby
>> 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 provides the illusion of very long strings but uses very little memory.

To be continued, on assignment 9!

What data did you need to perform those computations?
Modules and "mixins"
A Ruby *module* can be used to group related methods for organizational purposes.

Some methods for a homesick Haskell programmer at Camp Ruby:

```ruby
module Haskell
  def Haskell.head a  # Class method--prefixed with class name
    a[0]
  end

  def Haskell.tail a
    a[1..]
  end
  ...
  ...
end
```

```ruby
>> a = [10, "twenty", 30, 40.0]

>> Haskell.head a
=> 10

>> Haskell.tail a
=> ["twenty", 30, 40.0]
```
A module can be "included" in a class.
- Such a module 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:

```ruby
module Haskell
  def head
    self[0]
  end

  def tail
    self[1..]
  end
end

Previous version:
module Haskell
  def Haskell.head(a)
    a[0]
  end

  def Haskell.tail(a)
    a[1..]
  end
end
```
We can mix our Haskell methods into the **Array** class like this:

```ruby
% 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:

```ruby
>> 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
```
We can add those same capabilities to **String**, too:

```ruby
class String
    include Haskell
end
```

Usage:

```ruby
>> s = "testing"

>> s.head               #=> "t"

>> s.tail               #=> "esting"

>> s.tail.tail.head     #=> "s"
```

In addition to the **include** mechanism, what other aspect of Ruby facilitates mixins?

   Duck typing! (How?)
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:

```ruby
>> Integer.ancestors
=> [Integer, Numeric, Comparable, Object, Kernel, BasicObject]

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

```ruby
>> load "mixin1.rb"

>> Array.ancestors
=> [Array, Haskell, Enumerable, Object, Kernel, BasicObject]
```
The method `included_modules` shows the modules that a class includes.

```ruby
>> Array.included_modules => [Haskell, Enumerable, Kernel]
```

```ruby
>> Integer.included_modules => [Comparable, Kernel]
```

`instance_methods` can be used to see what methods are in a module:

```ruby
```

```ruby
>> Comparable.instance_methods.sort => [:<, :<=, ==, >, >=, :between?]
```

```ruby
>> Haskell.instance_methods => [:head, :tail]
```
The **Enumerable** module

When talking about iterators we encountered **Enumerable**. It's a module:

```ruby
>> Enumerable.class
=> Module
```

```ruby
>> Enumerable.instance_methods.sort
```

The methods in **Enumerable** use duck typing, requiring only an *each* method. **min**, **max**, and **sort**, also require `<=>` for values operated on.

If a class implements *each* and includes **Enumerable** then all those methods become available to instances of the class.
Here's a class whose instances simply hold three values:

```ruby
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`, lots of stuff works:

```ruby
>> 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"]]
```
Another common mixin is `Comparable`:

```ruby
>> Comparable.instance_methods
=> [:==, :>, :>=, :<, :<=, :between?]
```

`Comparable`'s methods are implemented in terms of `<=>.

Let's compare rectangles on the basis of area:

```ruby
class Rectangle
  include Comparable
  def <=> rhs
    (self.area - rhs.area) <=> 0
  end
end
```
Comparable, continued

Usage:

```ruby
>> 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 include Comparable making the following work?

```ruby
>> [r1,r2,r3].sort
=> [2 x 2 Rectangle, 5 x 2 Rectangle, 3 x 4 Rectangle]

>> [r1,r2,r3].min
=> 2 x 2 Rectangle
```
Some things I threw out when I was panicked...
Discernment:
Levels of support for types
Language-wise, what's an implication of the following examples?

Haskell:
> :t "abc" => "abc" :: [Char]

Python:
>>> type({}) => <class 'dict'>

Ruby:
>> /.s./ =~ "test" => 1

Racket:
> (+ 3/4 1/3 1/5) => 77/60

Icon:
][ 'tim korb' => 'bikmort' (cset)

The respective languages have **syntactic support** for a particular type.
Levels of support

A language's support for a feature can be viewed as at one of three levels:

Syntactic support
- Most languages have syntactic support for strings with "...".
- Scala and ActionScript have syntactic support for XML.
- Python has `x in y` and `x not in y` tests for membership.

Language support:
- Icon has a `table` type but no literal syntax, only `table(default)`.

Library support
- Ruby has `Complex '2+3i'` (but Racket allows just `2+3i`)
- Java and Python have classes for working with REs.
  - Python does have raw strings—`re.match(r"\w+\(\d+\)\", ...)`
- C and Icon have function libraries for working with REs.
"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 not be included?"

Let's explore that question with Ruby.
A "here document" is a third way to literally specify a string in Ruby:

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

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

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

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

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

- `%q` follows single-quote rules.
- `%Q` follows double quote rules.
- Symmetric pairs like `()`, `{}`, and `<>` can be used.
Partial summary of string literal syntax in Ruby:

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

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

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

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

How much is enough?

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 of Ruby's would you remove?
"Why" or "Why not?" as applied to operator overloading

Here are some examples of operator overloading:

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

```ruby
>> "abc" * 5
=> "abcabcabcabcabc"
```

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

```ruby
>> [10, 20, 30] * "...
=> "10...20...30" # "intercalation"
```

```ruby
>> "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.
  - There are separate mental footprints for reading code and writing code.

• 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")

Small: Supporting an operator on a new pair of types

Medium: Support for arbitrarily large integers

Small, medium or large:

• Support for operator overloading?

• Support for object-oriented programming?

  *Classes: An Abstract Data Type Facility for the C Language*

• Support for garbage collection?
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."

And then he added, "But for every feature you add, first find one to remove."
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)
The art of language design, continued

- 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 simpler thing turned out to be Icon.
  - SL5 was an expansion
  - Icon was a contraction
Design example: invocation in Icon

Procedure call in Icon:

```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 trivial example:

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

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

I got an OK to add string invocation to Icon.

Why did Ralph choose to allow this feature?
He felt it would increase the research potential of Icon.
Design example: Parallel assignment

An interesting language design example in Ruby is *parallel assignment*:

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

>> a
=> 10

>> b
=> [20, 30]

>> c, d = b
>> [c,d]
=> [20, 30]

>> a, b, c = "800-555-1211".split "-"

>> [a,b,c]
=> ["800", "555", "1211"]
```
Parallel assignment, continued

How could we do a swap with parallel assignment?

```ruby
>> x, y = 10, 20

>> x, y = y, x
```

Another way?

```ruby
>> x, y = [y, x]
```

Contrast:

Icon has a swap operator: `x :=: y`
Parallel assignment, continued

Speculate: Does the following work?

```ruby
>> a, b, c = [10, 20, 30, 40, 50]

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

Speculate again:

```ruby
>> a, *b, c = [10, 20, 30, 40, 50]

>> [a, b, c]
=> [10, [20, 30, 40], 50]

>> a, *b, *c = [10, 20, 30, 40, 50]
SyntaxError: (irb):57: syntax error, unexpected *
```

Section 4.5.5 in RPL has full details on parallel assignment. It is both more complicated and less general than pattern matching in Haskell. (!)
In conclusion...
My first practical Ruby program

September 3, 2006:

```ruby
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 then "T"; else "H"; end,
      d.strftime("%m/%d/%y"), n)
    n += 1
  end
  d += incs[pos % 2]
  pos += 1
end  # classdays.rb
```

Output:

```
T 08/22/06, # 1
H 08/24/06, # 2
T 08/29/06, # 3
...
```

CSC 372 Fall 2022, Ruby Slide
If we had more time, we'd...

• Learn how to write iterators (slides 261-275 in ruby.pdf)

• Work with regular expressions in Ruby (slides 276-326 in ruby.pdf)

• Learn about lambdas, blocks as explicit parameters, and call.

• Do some metaprogramming with hooks like method_missing, included, and inherited.

• Experiment with internal Domain Specific Languages (DSLs).

• Look at how Ruby on Rails puts Ruby features to good use.

• Take a peek at BDD (Behavior-Driven Development) with Cucumber or RSpec.

More with Ruby...
Writing Iterators
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, first a 3 and then a 7:

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

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

**Usage:**

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

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

There's no formal parameter that corresponds to a block. 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`:

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

A simple iterator, continued

### Usage:

```ruby
>> simple { |x| puts "\tx = \#{x}" }
```

```
simple: Starting...
  x = 3
simple: Continuing...
  x = 7
simple: Done...
=> "simple result"
```
Problem:

Write an iterator \texttt{around(x, delta)} that yields (i.e., invokes its block with) three values in turn: \texttt{x-delta}, \texttt{x}, and \texttt{x+delta}. It returns the range spanned by the three values.

Examples:

\begin{verbatim}
>> around(5,1) { |x| puts x }
4
5
6
=> 2

>> vals = []; around(Complex(0,1),0.5) { |v| vals << v}
=> 1.0

>> vals
=> [(-0.5+1i), (0.0+1i), (0.5+1i)]
\end{verbatim}
Iterators with parameters, continued

At hand:

```ruby
>> around(11, 2) { |v| print "#{v} " }
9 11 13 => 4
```

Solution:

```ruby
def around(x, delta)
  [-1, 0, 1].each do |m|
    yield x + m*delta
  end
  delta * 2
end
```

Note that parameters are passed to an iterator just like any other method.
Problem: \texttt{from\_to}

Problem:
Write an iterator \texttt{from\_to(f, t, by)} that yields the integers from \texttt{f} through \texttt{t} in steps of \texttt{by}, which defaults to 1. Assume \texttt{f} \leq \texttt{t}. \texttt{from\_to} returns the number of integers yielded.

\begin{verbatim}
>> from_to(1,3) { |i| puts i }
1
2
3
=> 3

>> from_to(0,99,25) { |i| puts i }
0
25
50
75
=> 4
\end{verbatim}
Solution:

```ruby
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:

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

>> from_to(-5,5) {
   |i| print i, " " }
-5 -4 -3 -2 -1 0 1 2 3 4 5 => 11
```
To pass multiple arguments to a block, specify multiple arguments for `yield`.

Imagine an iterator that produces overlapping pairs from an array:

```ruby
>> 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:
```ruby
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
```
When `yield` passes a value to a block the result of the block becomes the value of the `yield` expression.

Here is a trivial iterator to show the mechanics:

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

Usage:

```shell
$ ruby
> 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:
```ruby
def round_trip(x)
    r = yield x
    "yielded #{x} and got back #{r}"
end
```

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

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

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

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

4. `round_trip` returns "yielded 3 and ..."
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:

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

Solution:

```ruby
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?

<table>
<thead>
<tr>
<th>Iterator</th>
<th>Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>if yield [1,2] then  # [1,2].size == 1</td>
<td></td>
</tr>
<tr>
<td><em>Do nothing with [1,2]</em></td>
<td></td>
</tr>
<tr>
<td>if yield &quot;a&quot; then  # &quot;a&quot;.size == 1</td>
<td></td>
</tr>
<tr>
<td><em>Add &quot;a&quot; to result</em></td>
<td></td>
</tr>
<tr>
<td>if yield [3] then  # [3].size == 1</td>
<td></td>
</tr>
<tr>
<td><em>Add [3] to result</em></td>
<td></td>
</tr>
<tr>
<td>if yield &quot;four&quot; then  &quot;four&quot;.size == 1</td>
<td></td>
</tr>
<tr>
<td><em>Do nothing with &quot;four&quot;</em></td>
<td></td>
</tr>
</tbody>
</table>
Is `select` limited to arrays?

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

Why do we see that behavior?
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?
- `iterable` (?)
- `eachable` (?)

```
def select array
  result = []
  for element in array
    if yield element then
      result << element
    end
  end
  result
end
```
Round trips with `yield`, continued

What's the difference between our `select`,

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

And Ruby's `Array.select`?

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

>> select([
"just","a","test"])
=> ["just","test"]

Which is better?
Regular Expressions
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 be specified by unrestricted grammars.

An example of a context-sensitive language is all strings of the form $a^n b^n c^n$.

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 infinite 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.

- A regular language can also be described by a regular expression.
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.

Examples: ab, ababc, and ababababcccccccc.

The regular expression \((\text{north} | \text{south})(\text{east} | \text{west})\) describes a language with four strings: \{\text{northeast}, \text{northwest}, \text{southeast}, \text{southwest}\}.
Regular expressions have a sound theoretical basis and are also very practical.

UNIX tools such as the `ed` editor and the `grep` family introduced regular expressions to a wide audience.

Most current editors and IDEs support regular expressions in searches.

Many languages provide a library for working with regular expressions.
- Java provides the `java.util.regex` package.
- The command `man regex` shows the interface for POSIX regular expression routines, usable in C.

Some languages, Ruby included, have a regular expression type.
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.
Here is a regular expression written by Mark Cranness and posted at RegExLib.com:

```
^((?>[a-zA-Z\d!#$%&'\*+\-/=?_\{\}\~]+)+(?>[^\x01-\x7f])[^"\\]|\\[^\x01-\x7f])*"\x20*\)*(?<angle><)?)?((?!\.)(?>(?[^-\{\}]~)+)+|"((?=\x20*)\)*")@((?!)[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]:((?=\x20*)[^\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 `/regexp/` syntax, for regular expression literals.

```ruby
>> 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/` describes a language of five character strings of this form:

```
a<anychar>b<anychar>c
```

Some words containing strings in that language:

"albacore", "barbecue", "drawback", and "iambic".
The Ruby 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
```
Problem: Write in Ruby a trivial version of the UNIX `grep` command.

Usage:

$ ruby rgrep.rb g.h.i < /usr/share/dict/words
lengthwise

$ ruby rgrep.rb l.m.n < /usr/share/dict/words | wc -l
252

$ ruby rgrep.rb ......................... < /usr/share/dict/words
electroencephalograph's

Hint: #{} interpolation works in /.../ (regular expression) literals:

```ruby
>> s = "ab"
>> /#{s}-#{s.reverse}/ =~ "cab-bat"
=> 1
```
Desired:

$ ruby rgrep.rb g.h.i < /usr/share/dict/words
lengthwise

Solution:

```ruby
while line =(STDIN.gets  # `STDIN` so "g.h.i" isn't opened for input
  puts line if line =~ /#{ARGV[0]}/ 
end
```

```bash
csc 372 fall 2022, ruby slide 285
```
The match operator, continued

After a successful match using =~ we can use some cryptically named predefined global variables to access parts of the string:

$s` Is the portion of the string that precedes the match. (That's a backquote—ASCII code 96.)

$s& Is the portion of the string that was matched by the regular expression.

$s' Is the portion of the string following the match.

Example:

```ruby
>> "limit=300" =~ /=/
=> 5
>> $`
=> "limit" (left of the match)
>> $&
=> "=" (the match itself)
>> $'
=> "300" (right of the match)
```
Here's a handy utility routine from the Pickaxe book:

```ruby
def show_match(s, re)
  if s =~ re then
    "#{'$'}<<#{'$&'}}#{'$'}"
  else
    "no match"
  end
end
```

Usage:

```ruby
>> show_match("limit is 300", /is/)
=> "limit <<is>> 300"
```

```ruby
>> %w{albacore drawback iambic}.each { |w| puts show_match(w, /a.b.c/) }
<<albacore>>ore
dr<<awbac>>k
i<<ambic>>
```

**Great idea**: Put it in your `.irbrc`! Call it "sm", to save some typing!
When learning a language look for opportunities to use the language to learn about the language.

With **show_match** we're using Ruby to learn about Ruby regular expressions.

When teaching Icon I wrote a procedure named **snapshot()** to explore Icon's *string scanning* facility. A simple example:

```ruby
][ "testing" ? while move(3) do snapshot()
&subject = testing
&pos = 4 |

&subject = testing
&pos = 7 |

Failure
```
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("Testing 1, 2, 3...", /\[0-9\]/)
=> "Testing <<1>>, 2, 3..."

>> show_match("Take five!", /\[0-9\]/)
=> "no match"
[^characters] is a RE that matches any single 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] always has a one-character match!
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"
```
String.gsub does global substitution with both plain old strings and regular expressions

```ruby
>> "520-621-6613".gsub("-", "<DASH>")
=> "520<DASH>621<DASH>6613"

>> "B3WF5-NPH41-MVXRP-67C9J-J8A9M".gsub(/[A-Z][0-9]/, "<LD>")
=> "<LD>W<LD>-NP<LD>1-MVXRP-67<LD>J-<LD><LD>M"
```
Some frequently used character classes can be specified with \C
\d  \(\text{Stands for [0-9]}\)
\w  \(\text{Stands for [A-Za-z0-9_]}\)
\s  \(\text{Whitespace—blank, tab, carriage return, newline, formfeed}\)

The abbreviations \D, \W, and \S produce a complemented class.

Examples:
```ruby
>> show_match("Call me at 555-1212", /\d\d\d\d-\d\d\d\d/)
=> "Call me at <<555-1212>>"

>> "fun double(n) = n * 2".gsub(/\w/,"."
=> ".......(.) = .:*.

>> "ILC 119, 15:30-16:45 MW".gsub(/\D/, ""
=> "11915301645"

>> "buzz93@tv-2000.com".gsub(/[\w-]/,"*"
=> "******@*******.***"
Preceding an RE metacharacter with a backslash suppresses its meaning.

```ruby
>> show_match("123.456", /\../)
=> "12<<3.4>>56"

>> "5-3^2*2.0".gsub(/[^\-6]/, "_")
=> "5_3_2*2_0"

>> show_match("x1 = y2[3] + z4", /\d\)/)
=> "x1 = y2<<[3]>> + z4"
```

An old technique with regular expressions is to take advantage of the fact that some metacharacters only have meaning when used in certain positions:

```ruby
>> "5-3^2*2.0".gsub(/[-6^.]/, "_")
=> "5_3_2*2_0"
```
Alternatives can be specified with a vertical bar:

```ruby
>> show_match("a green box", /red|green|blue/)
=> "a <<green>> box"

>> %w{you ate a pie}.select { |s| s =~ /ea|ou|ie/ }
=> ["you", "pie"]
```
Parentheses can be used for grouping. Speculate: What regular language corresponds to the following regular expression?

```
/(two|three) (apple|biscuit)s/
```

```
{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

Problem: Read a list of words and print words that contain a specified pattern of consonants and vowels.

```ruby
% ruby convow.rb cvcvcvcvcvcvcvcvc < web2
c|hemicomineralologic|al
|hepatoperitonitis|
o|verimaginativeness|s
```

- A capital letter means to match exactly that letter, in lowercase.
- An `e` matches either consonant or vowel.

```ruby
% ruby convow.rb vvvDvvv < web2
Chromat|ioideae|
Rhodobacter|ioideae|
```

```ruby
% ruby convow.rb vvvCvvv < web2 | wc -l
24
```

```ruby
% ruby convow.rb vvvvevvv < web2 | wc -l
43
```
Solution: Loop through the command line argument's characters and build up a regular expression of character classes and literal characters. Then look for lines with a match.

```ruby
re = ""
ARGV[0].each_char do |char|
  re += case char
    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
$ ruby convow.rb cvc
[^aeiou][aeiou][^aeiou]
```

```ruby
$ 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 $R_1$ and $R_2$ are regular expressions then $R_1R_2$ 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.
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 \( ab^*c+d \) 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'.

```ruby
>> show_match("acd", /ab*c+d/)
=> "<<acd>>"
```

```ruby
>> show_match("abccccc", /ab*c+d/)
=> "no match"
```

```ruby
>> show_match("abcabccccccddd", /ab*c+d/)
=> "abc<<abcccccd>>dd"
```
At hand:
- \texttt{R*} matches zero or more occurrences of \texttt{R}
- \texttt{R+} matches one or more occurrences of \texttt{R}
- \texttt{R?} matches zero or one occurrences of \texttt{R}

What does the RE \texttt{-?\d+} describe?
Integers with any number of digits

\begin{verbatim}
>> show_match("y is -27 initially", /-?\d+/)
=> "y is \texttt{ <<-27>>} initially"

>> show_match("maybe --123.4e-10 works", /-?\d+/)
=> "maybe \texttt{ <<-123>>.4e-10 works}"

>> show_match("maybe --123.4e-10 works", /-?\d*/)  # *, not +
=> "\texttt{ <<>>maybe --123.4e-10 works}"
\end{verbatim}
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

$(\d?\d?\d)(,\d\d\d)*$

-- Alan Smith, Spring '16
The *, +, and ? quantifiers, continued

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>>>

Piece by piece:
-?     Maybe a minus sign
\d*\.\d+ Zero or more digits, decimal point, one or more digits
(e-?     e possibly followed by a minus sign
\d+)    One or more digits
)?      Maybe!

CSC 372 Fall 2022, Ruby Slide 303
*, +, 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
More examples of greedy behavior:

```
>> show_match("xabbbbc", /a.*b/)
=> "x<<abbbbc>>c"

>> show_match("xabbbbc", /ab?b?/)
=> "x<<abbbc>>bbc"

>> show_match("xabbcxyzc", /ab?b?.*c/)
=> "x<<abbbbcxyzc>>"
```

Why are *, +, and ? greedy?
In the following we'd like to match just 'abc' but the greedy asterisk goes too far:

```ruby
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:

```ruby
>> show_match("x + 'abc' + 'def' + y", '/.*?'/)
=> "x + <<'abc'>> + 'def' + y"
```

?? and +? are supported, too. The three are also called reluctant quantifiers.
Sidebar: primitives vs. idioms

From the previous slide:

```ruby
>> show_match("x + 'abc' + 'def' + y", '/.*?\*/)
=> "x + <<<'abc'>>> + 'def' + y"
```

Years ago, before reluctant quantifiers were introduced, a complemented character class was used to stop at the next occurrence of a character:

```ruby
>> show_match("x + 'abc' + 'def' + y", '/[^\]*/)
=> "x + <<<'abc'>>> + 'def' + y"
```

A common language design trade-off is what primitives to provide vs. how many idioms/techniques users must master.

TODO: Example: C has relatively few rules but they must be well-understood.
Sidebar, continued

Here is a complete summary of Icon's *string scanning* facility: (81 words)

<table>
<thead>
<tr>
<th>Functions that produce positions to be used with <code>tab(n)</code>:</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>many(cs)</code> produces position after run of characters in <code>cs</code></td>
</tr>
<tr>
<td><code>upto(cs)</code> generates positions of characters in <code>cs</code></td>
</tr>
<tr>
<td><code>find(s)</code> generates positions of <code>s</code></td>
</tr>
<tr>
<td><code>match(s)</code> produces position after <code>s</code>, if <code>s</code> is next</td>
</tr>
<tr>
<td><code>any(cs)</code> produces position after a character in <code>cs</code></td>
</tr>
<tr>
<td><code>bal(s, cs1, cs2, cs3)</code> similar to <code>upto(cs)</code>, but used with &quot;balanced&quot; strings.</td>
</tr>
</tbody>
</table>

Other functions:
- `pos(n)` tests if `&pos` is equivalent to `n`
- `move(n)` advances `&pos` by `n`
- `expr1 ? expr2` scans `expr1` with `expr2`

http://www.rexegg.com/regex-quickstart.html: 1,352 words


But, there are many idioms and techniques that need to be mastered to use Icon's string scanning mechanism effectively.
Specific numbers of repetitions

We can use curly braces to require a specific number of repetitions:

```ruby
>> show_match("Call me at 555-1212!", /\d{3}-\d{4}/)
=> "Call me at <<555-1212>>!"
```

An inclusive range can be specified with `R{min,max}`:

```ruby
>> mdy = /\d{1,2}-\d{1,2}-(\d{4}|\d{2})/

>> show_match("3-17-2018", mdy)
=> "<<3-17-2018>>"

>> show_match("12-1-99", mdy)
=> "<<12-1-99>>"

>> show_match("7-555-99", mdy)
=> "no match"
```

Speculate: What do `R{,n}` and `R{n,}` specify?
split and scan with regular expressions

We can split a string using a regular expression:

```ruby
>> "one, two,three / four".split(/\s,\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 strings of interest instead of what separates them, `scan` is a better choice:

```ruby
>> "one, two,three / four".scan(/\w+/)
=> ["one", "two", "three", "four"]
```

```ruby
>> "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:

```ruby
>> "one, two,three / four".scan(/\w+ | \W+/)
=> [" ", "one", ", ", "two", ", ", "three", "/ ", "four"]
```
Reminder: $s \not= /x/$ succeeds if "x" appears anywhere in $s$.

The metacharacter ^ is an anchor when used at the start 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.

$>> \text{show\_match("this is x", } /^x/ / \text{)} \quad \Rightarrow \text{ "no match"}$

$>> \text{show\_match("this is x", } /^this/ / \text{)} \quad \Rightarrow \text{ "<<this>> is x"}$

What will $/^x|y/$ match? Hint: it's not the same as $/^\(x|y\)/$

What does $/^.[^0-9]/$ match?
Another anchor is $$. It constrains the preceding regular expression to appear at the end of the string.

```ruby
>> show_match("ending", /end$/)
=> "no match"

>> show_match("the end", /end$/)
=> "the <<end>>"
```

What does /\d+$/ match?
- Strings that end with one or more digits.

Can /\d+$/ be shortened?
- /\d$/
We can combine the ^ and $ anchors to fully specify a string.

Problem: Write a RE to match lines with only a curly brace and (maybe) whitespace. (Recall that \s matches a single character of whitespace.)

```
>> show_match("  }   ", /^\s*\{\}/\s*$/) => "<<  }   >>"
```

Using grep, print lines in Ruby source files that are exactly three characters long.

```
% grep ^...$ *.rb
```
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+.$/
line =~ /^.*\w+\d+$/
```
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.

```ruby
for fname in files
  f = open(fname)
  while line = f.gets
    ...lots of processing to build a data structure, bdata...
  end
  p bdata  # outputs way too much to easily analyze!!
end
```

We could edit some data files down to a few names. Could we achieve the same effect more easily using regular expressions?

```ruby
for fname in files
  f = open(fname)
  while line = f.gets
    next unless line =~ /^\(John|Dana|Mary\),/  # toomuch.rb
    ...processing...  # toomuch.rb
  end
end
```

Sidebar: Dealing with too much input

Could do a more sophisticated filtering, too!
Recall that `convow.rb` earlier simply does `char.downcase` on any characters it doesn't recognize. `downcase` doesn't change `^` or `$.

The command

```
% ruby convow.rb ^cvc$
```

builds this RE, which matches only three-letter `cvc` words:

```
/^[^aeiou][aeiou][^aeiou]$/
```

Let's explore with it:

```
% ruby convow.rb ^cvc$ < web2 | wc -l
858
% ruby convow.rb ^vcccv$ < web2 | wc -l
15
% ruby convow.rb ^vcccv$ < web2
| oxyphyte |
```
The following regular expression uses three *named groups* to capture the elements of a binary arithmetic expression:

```ruby
>> re = /(?<lhs>\d+)(?<op>[+\-\*/\-])(?<rhs>\d+)/
```

After a successful match, the predefined global `$~`, an instance of `MatchData`, shows us the groups:

```ruby
>> re =~ "What is 100+23?"
=> 8

>> $~
=> #<MatchData "100+23" lhs:"100" op:"+" rhs:"23">

>> $~["lhs"]
=> "100"
```

Named groups are sometimes called *named backreferences* or *named captures*. 
At hand:
/(?<lhs>\d+)(?<op>[+\-\/*\-])(?<rhs>\d+)/

Important: Named groups must always be enclosed in parentheses.

Consider the difference in these two REs:
/x(?<n>\d+)/
    Matches strings like "x10" and "testx7ing"

/x?<n>\d+/
    Matches strings like "<n>10", "ax<n>10", "testx<n>10ing"

Design lesson:
"(" in a RE originally had no meaning, so it provided an opportunity for extension without breaking any existing REs.

For a pitfall/feature, see *NAMED CAPTURES AND LOCAL VARIABLES* in RPL.
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? (Don't peek!)
def main
    mins = 0
    while line = gets do
        line.scan(/[^\s,]+/).each { |time| mins += parse_time(time) }
    end
    printf("%d:%02d
", mins / 60, mins % 60)
end

def parse_time(s)
    if s =~ /^(?<hours>\d+):(?<mins>[0-5]\d)$/
        $~"hours".to_i * 60 + $~"mins".to_i
    elsif s =~ /^(?<n>\d+)(?<which>[hm])$/
        n = $~"n".to_i
        if $~"which" == "h" then n * 60
        else n end
    else
        0 # return 0 for things that don't look like times
    end
end
main
Problem:
Write a method \texttt{pt(s)} that takes a string like "\{(10, 'a'), (3, 'x'), (7, 'o')\}" and returns an array with the sum of the numbers and a concatenation of the letters. If \texttt{s} is malformed, \texttt{nil} is returned.

Examples:
\begin{verbatim}
>> pt "\{(10, 'a'), (3, 'x'), (7, 'o')\}"
=> [20, "axo"]

>> pt "\{(100, 'c')\}"
=> [100, "c"]

>> pt "\{(10, 'x'), (5, 7, 'y')\}"
=> nil

>> pt "\{(10, 'x'), (5, 'y'), \}"
=> nil
\end{verbatim}
Desired:

```ruby
>> pt "[(10,'a'),(3,'x'),(7,'o')]"
=> [20, "axo"]
```

Approach:

1. Remove outer brackets: 
   "(10,'a'),(3,'x'),(7,'o')"
2. Append a comma: 
   "(10,'a'),(3,'x'),(7,'o')," (Why?!)
3. Recognize \textbf{(NUM,LET)}, and replace with 
   
4. Repeat 3. until failure
5. If nothing left but an empty string, success!

Important: By appending that comma we produce a simple repetition,

\textit{(tuple,)+}

rather than

\textit{tuple (,tuple)∗}
Solution:

```ruby
def pt(s) # process_tuples.rb
    if s =~ /^\[(?<tuples>.*))$/ then
        tuples = $~["tuples"] + ","
        sum, lets = 0, ""
        tuples.gsub!(\((?<num>\d+),\(?<let>[a-z])\)),/ do
            sum += $~["num"].to_i
            lets << $~["let"]
        ""  # block result--replaces matched string in tuples
    end
    if tuples.empty? then
        [sum,lets]
    end
end
```

Approach:
1. Remove outer brackets
2. Append a comma
3. Recognize (NUM,LET), and replace with ""
4. Repeat 3. until failure
5. If nothing left but an empty string, success!
Avoiding repetitious REs

Imagine a simple calculator that accepts input lines such as these:
   \[ x=7 \]
   \[ yval=x+10*x \]
   \[ x+yval+z \]

Here's a very repetitious RE that recognizes those lines above:

```plaintext
valid_line = /\^([a-zA-Z][a-zA-Z\d]*)?([a-zA-Z][a-zA-Z\d]* | \d+)(([-+*/\^])([a-zA-Z][a-zA-Z\d]* | \d+))*$/
```

Let's use some intermediate variables to build that same RE.

```plaintext
var = /\[a-zA-Z\d]*\//i  # trailing "i": case insensitive

expr = /\(#\{var\}| \d+)\//

op = /\[-+*/\^]/

valid_line = /\^(#\{var\}=)?\#\{expr\}(#\{op\}\#\{expr\})*$/
```
Good news:

Lots of programs support regular expressions:

• The `grep` family (`global/regular expression/print`, derived from the `ed` editor.)
• Most editors and IDEs
• MySQL has `REGEXP` and `RLIKE` operators
• Lots more...

Most programming languages have some sort of support for regular expressions.

Bad news:

• "Core" RE primitives are very consistently supported but the farther you get from that core, the less consistent support becomes.

• There's deliberate variation, too. Example: `grep`, `egrep`, and `egrep -P` regular expressions differ.
Lots more with regular expressions

Our look at regular expressions ends here but there's lots more, like...

- Back references—/(.).\2\1/ matches 5-character palindromes
- Nested regular expressions
- Nested and conditional groups
- Conditional subpatterns
- Zero-width positive lookahead
- Matching strings with newlines (\A, \Z, \z)

Proverb:

_a programmer decided to use regular expressions to solve a problem.
Then the programmer had two problems._

- Regular expressions are great, up to a 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.