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.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Ruby Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>3+4</td>
<td>&quot;call 555-1212&quot; =~ /[0-9]+/</td>
</tr>
<tr>
<td>_.class</td>
<td>[&quot;$, $&amp;, $&quot;] # just after above</td>
</tr>
<tr>
<td>&quot;x&quot;.&lt;TAB&gt;&lt;TAB&gt;</td>
<td>&quot;9-3&quot;.gsub(/\d/) {</td>
</tr>
<tr>
<td>(&quot;abc&quot;*3_500).size.to_s</td>
<td>{1 =&gt; 2.0, :three =&gt; [!nil, 3]}</td>
</tr>
<tr>
<td>s = ('a'..'z').to_a.join</td>
<td>if 1 &lt; 2 then 3 else 4 end</td>
</tr>
<tr>
<td>s[5,9]</td>
<td>while line = gets; p line; end</td>
</tr>
<tr>
<td>s[5..9]</td>
<td>print 3 unless 4 &lt; 5</td>
</tr>
<tr>
<td>s[-3..]</td>
<td>3.+(4)</td>
</tr>
<tr>
<td>&quot;a test here&quot;.split.map {</td>
<td>s</td>
</tr>
<tr>
<td>10[3]</td>
<td>1.i**2</td>
</tr>
<tr>
<td>s=<code>date; hostname</code></td>
<td></td>
</tr>
<tr>
<td>s.gsub!(&quot;2&quot;,&quot;TWO&quot;); print s</td>
<td></td>
</tr>
</tbody>
</table>

Let's see what we can learn about Ruby by trying some expressions using `irb` on lectura.
irb is the commonly used REPL for Ruby. Let's run it on lectura:

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

irb's prompt changes if an expression is incomplete:

```ruby
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
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`
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?
```

```ruby
>> "ostentatious".tr("aeiou","12345")
```

```ruby
>> "testing".length()
```

Repeat: In Ruby, every value is an object.

What are some values in Java that are not objects? How about Python?
Everything is an object, continued

Of course, "everything" includes numbers:

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

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

>> 17**37
=> 3362095853201812742282475234995233875224247377

>> _.pred()

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

The TAB key shows completions in `irb`:

```
>> 7.<TAB><TAB>
7.bit_length
7.digits
7.lcm
7.gcd
7.rationalize
7.next
7.gcdlcm
7.denominator
7.numerator
7.upto
7.chr
7.inspect
7.size
7.succ
7.ord
7.to_int
```

```
7.to_s
7.to_i
7.to_f
7.to_r
7.div
7.divmod
7.fdiv
7.coerce
7.modulo
7.remainder
7.abs
7.magnitude
7.integer?
7.floor
7.ceil
```

(119 in all...)

CSC 372 Fall 2022, Ruby Slide 16
Parentheses are optional, sometimes

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)
```

Commas 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.
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!
```
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:

```
$ 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
% cat hello2.rb     # fall22/ruby/hello2.rb
def say_hello
  puts "Hello, world!"
end

say_hello

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

The definition of `say_hello` must precede the call.

We'll see later that Ruby is somewhat sensitive to newlines.
A line-numbering program

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:

```
Usage:
% cal | ruby tac.rb

27 28 29 30
20 21 22 23 24 25 26
13 14 15 16 17 18 19
 6  7  8  9 10 11 12
 1  2  3  4  5
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.
Strings and string literals

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}"
=> "🍕 + 🐟 = 😊"
```
String literals, continued

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

```ruby
>> puts \n\t
\n\t
=> nil

>> \n\t'.length

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

>> '\\\\'.length
```

CSC 372 Fall 2022, Ruby Slide 29
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
```
Strings are mutable

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

>> x
```

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

>> x
```

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
```

**********

CSC 372 Fall 2022, Ruby Slide 33
Applicative vs. imperative methods, continued

In contrast, an *imperative* method potentially changes its receiver.

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

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

```ruby
>> s
```

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"
```

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

These (common) operators work with most other types, too!
String comparisons, continued

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

Behavior:

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

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

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

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

What are the Java and C analogs for `<=>` when applied to strings?
Substrings

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

```
>> s = "abcd"

>> s[0]       # Positions are zero-based

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

>> s[100]
```

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
>> s[-1] = ""
>> s
```
A substring can be referenced with

\[ s[start, length] \]

\[
\begin{align*}
\text{\texttt{\>> s = "replace"}} \\
\text{\texttt{\>> s[2,3]}} \\
\text{\texttt{\>> s[3,100]}} \\
\text{\texttt{\>> s[-4,3]}} \\
\text{\texttt{\>> s[10,10]}} \\
\end{align*}
\]

Substrings, continued
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]

>> s[r] = ""

>> s
```
Substrings with ranges, continued

It's more common to use range literals with strings:

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

>> s[2...-4]

>> s[-8..-4]

>> s[-4..-8]

>> s[2..]
```
A substring can be the target of an assignment:

```
>> s = "replace"

>> s[0,2] = ""

>> s

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

>> s

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

>> s
```
Expression interpolation in string literals

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
```

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
=> 39317695287172535490534173386882756704761607664135852855034678556753487133293648186980649622260361388994869790176611

>> (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)
```
Numbers, continued

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

>> (0.0/0.0).nan?
```
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'

> >> _ * _
```
There's `Rational`, too.

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

```ruby
>> it * 300
```

```ruby
>> Rational 0.5
```

```ruby
>> Rational 0.6
```

```ruby
>> Rational 0.015625
```
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.

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

>> x.class
=> Fixnum

>> x += 1
=> 4611686018427387904

>> x.class
=> Bignum

>> x -= 1
=> 4611686018427387903

>> x.class
=> Fixnum
```
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"

>> 10 + "20".to_f

>> 10 + 20.9.to_i
```

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

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.

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

>> a[0]

>> a[1,2]  # a[start, length]

>> a[-1]

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

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

>> a[1] = "twenty"; a

>> a[2..4] = %w{a b c d e}; a

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

Semicolon separates expressions. We make a change and show new value.
A few more:

```ruby
>> a
=> [10]

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

>> a[4] = [5,6]; a

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

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

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

>> a.reverse; a

>> a.reverse!; a
```
Arrays, continued

A few more:

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

g>> a[0].shift


g>> a

g>> a.unshift "a","b","c"


g>> a.shuffle.shuffle
```
Even more!

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

>> a + b

>> a - b

>> a & b

>> a | b

>> ('a..'+'zzz').to_a.size
```

TODO: show intercalation!
Comparing arrays

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

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

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

>> [1,2,[3,"bcd"]]) == [1,2] + [[3,"abcde"[1..-2]]]
```
Comparing arrays

Comparison of arrays with `<=>` is lexicographic.

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

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

Speculate: Will the following work?

```ruby
>> [1,2] < [3,4]
```
Comparison with `<=>` produces `nil` if differing types are encountered.

```ruby
>> [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?

```ruby
>> ["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:

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

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

What does Icon do better? What does Icon do worse?

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]
>> a.push a
>> a.size
>> a[-1]
>> a[-1][-1][-1]
```

Arrays can be cyclic
Control Structures
Here's a loop to print the integers from 1 through 10, one per line.

\[
i=1\\
\text{while } i \leq 10 \text{ do } \quad \# \text{"do" is optional} \\
\quad \text{puts } i \\
\quad i += 1 \\
\end{\text{while}}
\]

When \( i \leq 10 \) produces \textbf{false}, control branches to the code following \textbf{end}, if any.

The body of the \textbf{while} is always terminated with \textbf{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.
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?

```
Traceback (most recent call last):
```

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`)

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

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

Is the following ok?

```
while i
  <= 10 do puts i; i += 1 end
```
The incomplete expression rule does have some pitfalls.

Example: Ruby considers
\[ x = a + b + c \]

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

Rule of thumb: If breaking an expression across lines, end lines with an operator:
\[ x = a + b + c \]

Alternative: Indicate continuation with a backslash at the end of the line.
Academic writing on programming languages commonly uses the term "statement" to denote a syntactic element that performs operation(s) but does not produce a value.

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

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

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

Dilemma: 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

>> true && "abc"

>> nil && 1
```

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
```
```ruby
>> false || 2
```
```ruby
>> "abc" || "xyz"
```
```ruby
>> s = "abc"
>> s[0] || s[3]
```
```ruby
>> s[4] || false
```

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

An exclamation mark inverts a logical value. The resulting value is always true or false.

```ruby
>> ! true

>> ! 1

>> ! nil

>> ! (1 || 2)

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

>> ![nil]
```

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?

We can write this,

\[
x < 2 \text{ and } y > 3 \text{ or } x \times y < 10 \text{ or } z > 20
\]

instead of this:

\[
(x < 2 \text{ and } y > 3) \text{ or } (x \times y < 10 \text{ or } z > 20)
\]

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

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

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

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

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
```

Ruby also provides `1 > 2 ? 3 : 4`, a ternary conditional operator, just like the C family. Is that a good thing or bad thing?
if-then-else, continued

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

```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
```
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:

```ruby
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.
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
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```
The for loop

Here are three examples of Ruby's for loop:

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

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

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

The "in" expression must 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?

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?

Execution:
```ruby
>> hello
Hello!
=> nil
>>
```

Contrast with Python:
```ruby
>>> 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.)
A source file can be loaded with `Kernel.load`:

```
>> 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?

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

Testing methods with `irb`, continued

```ruby
$ 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.

```
>> [self.class, self.to_s]   #=> [Object, "main"]
>> methods_b4 = self.private_methods
>> load "simple.rb"

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

self has three more private methods after loading simple.rb.

Methods can be added to a class at run-time in Ruby!
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.
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
```
Arbitrary number of arguments

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:

```
  >> 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.
Arbitrary number of arguments, continued

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.
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.
Testing methods when there's a "main"

I'd like to load the following file and then test `showline`, but loading it in `irb` seems to hang. Why?

```ruby
% 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...
```
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`:

```
% 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"

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

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` ...
Global variables

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

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

def g
    $x = 100
end

$x = 10
f
g

puts "top-level: $x = #{$x}"
Predefined global variables

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`. 
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 $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?

Local variables are local

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

```ruby
# cat scope1.rb
x = 10

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

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

f
```

```ruby
% ruby scope1.rb
% ruby scope1.rb
```
A rule in Ruby is that if an identifier begins with a capital letter, it represents a *constant*.

The first assignment to a constant is considered initialization.

```
>> MAX_ITEMS = 100
```

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

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

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

```
>> 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

f

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

% 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!

```
>> 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?
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:

```
> :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.
Static typing, continued

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
- Misspelling the name of a method (true?)
- Invoking a method with the wrong number of arguments
- Forgetting a `return` at the end of a method that returns a value

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?
Variables in Ruby have no type

In Java, variables are declared to have a type.

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

```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
```
Methods in Ruby have no type

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
```

```ruby
>> (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!
Dynamic typing

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: astore_4
7: iconst_0
8: iinc 2, 1
9: istore 5
10: iload 5
11: goto 11
12: aload_3
13: aload 3
14: if_icmpne 33
15: if_icmpge 39
16: iconst_0
17: iinc 5, 1
18: aload_3
19: iload 5
20: iinc 2, 1
21: iload 5
22: iload 6
23: iinc 2, 1
24: iload 6
25: iload 6
26: iload 0
27: if_icmpne 33
28: goto 11
29: aload_2
30: invokevirtual
31: return
```

See also [https://en.wikipedia.org/wiki/Java_bytecode_instruction_listings](https://en.wikipedia.org/wiki/Java_bytecode_instruction_listings)
Static vs. dynamic typing

With respect to static typing, what are the implications of dynamic typing for...

- Loading ("compilation") speed?
- Execution speed?
- The likelihood of code to be correct?
Can testing compensate?

A long-standing question in industry:
   Can a good test suite find type errors in dynamically typed code as effectively as static type checking?

What's a "good" test suite?
   Full code coverage? (every line executed by some test)
   Full path coverage? (all combinations of paths exercised)
   How about functions whose return type varies?

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

"Why have to write tests for things a compiler can catch?"
   —Brendan Jennings, SigFig
What ultimately matters?

What does the user of software care about?
• Dynamic vs. static typing? Ruby vs. Java?
  o No!

• Software that works
  o Facebook game vs. radiation therapy system

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

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

• Affordable
  o "People will pay to stop the pain." – Doug Higgins
  o 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
```

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

The *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?
Duck typing, continued

```
>> double 10

>> double "abc"

>> double [1,2,3]

>> double Rational(3)

>> double 1..10
```

For reference:
```
def double x
  x * 2
end
```

- Is it good or bad that `double` operates on so many different types?
- Is `double` polymorphic?
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", ['<<<', '>>>']
```

```ruby
>> wrap [1,2,3], [['...']]
```

```ruby
>> wrap 10,3
```
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!)
```

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

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.

```ruby
>> [10, "twenty", 30].collect { |v| v * 2 }
```

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

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

What do these remind you of?
The block for `Array.sort` takes two arguments.

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

Speculate: What are the arguments being passed to `sort`'s block? How could we find out?
Problem: Sort the words in a sentence by descending length.

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

What does *Array#sort* do if not given a block?
```
>> "a longer try first".split.sort
```
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
>> 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}
In Ruby 1.9, **String** lost its `each` method!

```ruby
>> "abc".each { |c| puts c }
```

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 }
```
Sidebar: Iterate with `each` or use a `for` loop?

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

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

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

and iteration with `each`,

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

Which is better?
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:

$ 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 fors!)
- Kernel.readlines returns an array of all lines of standard input:
  
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...

```ruby
nums = STDERR.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 appears only in a block, it is local to the block.

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

If a variable appears outside a block, references inside the block apply to the outer variable.

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

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

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

You can try the above cases 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.

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

>> [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.

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

>> "try this first".split.sort { |a,b| b.size <=> a.size } => [...]  
Invokes block an arbitrary number of times; each block result guides further computation towards final result.
```
The **Hash** class
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:
  >> numbers
  => {"one"=>1, "two"=>2}

Subscripting with a key fetches the associated value.

  >> numbers["two"]
  => 2

What will happen with a non-existent key?
  >> numbers["three"]
At hand:

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

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

```ruby
>> numbers.keys

>> numbers.invert

>> numbers.flatten

>> numbers.to_a
```
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

>> 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, nil is returned. Full detail: If a key is not found, the default value of the hash is returned.

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

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

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

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

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

>> h["y"]
```
Problem: write `tally.rb`, to tally occurrences of blank-separated "words" on standard input.

```ruby
% 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
```

```ruby
tally.rb
```

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
p counts
```
The output of `tally.rb` is not customer-ready!

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

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

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

Problem: Produce nicely labeled output, like this:

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

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]]
Let's turn `tally.rb` into a cross-reference program:

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

```
% ruby xref.rb < xref.1
Word            Lines
be              1, 2, 3
is              2
not             2
or              1
question        3
the             3
to              1, 2, 3
```

How can we approach it? (Don't peek!)
xref.rb, continued

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 "\texttt{p refs}" after that loop, here's what we see:

\begin{verbatim}
% cat xref.1
 to be or
 not to be is not
 to be the question
\end{verbatim}

\begin{verbatim}
% ruby xref.rb < xref.1
{"to"=>[0, 1, 2], "be"=>[0, 1, 2], "or"=>[0], "not"=>[1],
 "is"=>[1], "the"=>[2], "question"=>[2]}
\end{verbatim}

We want:

\begin{verbatim}
% ruby xref.rb < xref.1
 Word     Lines
 be       1, 2, 3
 is       2
 not      2
\end{verbatim}
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>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

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

```ruby
max_len = refs.map { |k,v| k.size }.max
fmt = "%-#{max_len}s  %s\n"
p
```
A Hash.new pitfall

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:

   \[\texttt{h = Hash.new }[]\]  \# Our hope: The initial value for each key will be an empty list.

   \[\texttt{\"it", \"and\", \"or\"}.each \{|w| h[h.size] \ll w \}\]

   \[\texttt{h}\]
   \[\Rightarrow \{\}\]
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?
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

cl2 = Counter.new "c2"
cl2.click

puts cl2  # Output: c2's count is 1
cl2.click
puts "c2 = #{cl2.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`:

```ruby
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.

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

  def click
  end

  def reset
  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:
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:0x0000010117aaa0 @x=1, @y=1>
```
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?
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
```

```ruby
>> load "random.rb"
>> 12.times { print 6.rand, " " }

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

Observe the following. What does it suggest to you?

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

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

>> class Z; puts "here"; end
here
=> nil
```
Class definitions are executable code

At hand: A class definition is executable code. The following class definition uses a case statement to selectively executedefs 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:

```
>> 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"

>> x

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

>> eval gets
  s.reverse
```

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.
`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:0x0000010185d930>
>> x.add(3,4) => 7
>> x.last "abcd" => "d"
```
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:
```
>> x = X.new
>> x.f
in f
in g
>> x.g
NoMethodError: private method `g' ...
```

Speculate: What are `private` and `public`? Keywords?