Introduction

What is Ruby?

Experimenting with Ruby using irb

Executing Ruby code in a file

Everything is an object

Variables have no type

Ruby's philosophy is often "Why not?"

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".

Ruby was invented by Yukihiro Matsumoto ("Matz"), a "Japanese amateur language designer" (his own words). Here is a second-hand summary 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."
>
> http://www.rubygarden.org/faq/entry/show/5

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 and try Ruby. I'm sure you'll find out for yourself."

What is Ruby?

Ruby is a language in flux.

- Version 1.8.4 is installed on lectura. Version 1.9 is available.
- There is no written standard for Ruby. The language is effectively defined by MRI—Matz' Ruby Implementation.

There is good on-line documentation:

- The first edition of our text, the "pickaxe book", is available for free:
  www.ruby-doc.org/docs/ProgrammingRuby
- Documentation for the core classes is at www.ruby-doc.org/core.
- The above and more is collected at www.ruby-lang.org/en/documentation.
- Chapter 22 of the text is a fairly concise language reference for Ruby. It will soon be available through the UA library as an E-Reserve. Watch for a link on our website.

What is Ruby?

Ruby is getting a lot of attention and press at the moment. Two popular topics:

- Ruby on Rails, a web application framework.
- JRuby, a 100% pure-Java implementation of Ruby. With JRuby, among other things, you can use Java classes in Ruby programs. (jruby.codehaus.org)

Experimenting with Ruby using irb

There are two common ways to execute Ruby code: (1) Put it in file and execute the file with the "ruby" command. (2) Use irb, the Interactive Ruby Shell. We'll start with irb.

NOTE: The examples on these slides assume a particular configuration for irb, so before you run irb the first time, copy our .irbrc file into your home directory on lectura:

```
cp /home/cs372/fall06/ruby/.irbrc ~
```

irb's default prompt contains a little more information than we need at the moment, so we'll ask for the "simple" prompt:

```
% irb --prompt simple
>>
```

To exit irb, use control-D.

irb, continued

irb evaluates expressions as are they typed.

```
>> 1 + 2
=> 3

>> "testing" + "123"            (NOTE: No trailing semicolon!)
=> "testing123"
```

One of the definitions in our .irbrc allows the last result to be referenced with "it":

```
>> it
=> "testing123"

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

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

```
>> 1.23 +
?> 1e5
=> 100001.23
```

Executing Ruby code in a file

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

By convention, Ruby files have the suffix .rb.

Here is "Hello" in Ruby:

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

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

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

Executing Ruby code in a file, continued

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

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

say_hello

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

The definition of `say_hello` must precede the call.

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

A line-numbering program

Here is a program that reads lines from standard input and writes them, with a line number, to standard output:

```
% cat numlines.rb
line_num = 1

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

Execution:

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

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 already seen.

Everything is an object

In Ruby, *every* value is an object.

Methods can be invoked using the form *value.method(parameters...)*.

```
>> "testing".index("i")         # Where's the first "i"?
=> 4

>> "testing".count("t")         # How many times does "t" appear?
=> 2

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

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

Repeat: In Ruby, every value is an object.

What are some values in Java that are not objects?

Everything is an object, continued

Parentheses can be omitted from an argument list:

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

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

>> puts "number",3
number
3
=> nil

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

If no parameters are required, the parameter list can be omitted.

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

Everything is an object, continued

Of course, "everything" includes numbers:

```
>> 7.class
=> Fixnum

>> 1.2.class
=> Float

>> (3-4).abs
=> 1

>> 17**50
=> 33300140732146818380750772381422989832214186835186851059977249

>> it.succ
=> 33300140732146818380750772381422989832214186835186851059977250

>> it.class
=> Bignum
```

Everything is an object, continued

The TAB key can be used to show completions:

```
>> 100.<TAB>
                                    100.nil?
100.__id__                          100.nonzero?
100.__send__                        100.numerator
100.abs                             100.object_id
100.between?                        100.power!
100.ceil                            100.prec
100.chr                             100.prec_f
100.class                           100.prec_i
100.clone                           100.private_methods
100.coerce                          100.protected_methods
100.denominator                     100.public_methods
100.display                         100.quo
100.div                             100.rdiv
100.divmod                          100.remainder
100.downto                          100.require
```

Variables have no type

In Java, variables are declared to have a type. When a program is compiled, the compiler ensures that all operations are valid with respect to the types involved.

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

```
>> x = 10
=> 10

>> x = "ten"
=> "ten"

>> x.class
=> String

>> x = x.length
=> 3
```

Here's another way to think about this: Every variable can hold a reference to an object. Because every value is an object, any variable can hold any value.

Variables have no type, continued

It is often said that Java uses *static typing*. Ruby, like most scripting languages, uses *dynamic typing*.

Sometimes the term *strong typing* is used to characterize languages like Java and *weak typing* is used to characterize languages like Ruby but those terms are now often debated and perhaps best avoided.

Another way to describe a language's type-checking mechanism is based on *when* the checking is done. Java uses *compile-time type checking*. Ruby uses *run-time type checking*.

Some statically-typed languages do some type checking at run-time. An example of a run-time type error is Java's ClassCastException. C does absolutely no type-checking at run-time. Ruby does absolutely no type-checking at compile-time.

What is ML's type-checking approach?

Variables have no type, continued

In a statically typed language a number of constraints can be checked at compile time. For example, all of the following can be verified when a Java program is compiled:

x.getValue()	*x must have a getValue method*
x * y	*x and y must be of compatible types for **
x.f(1,2,3)	*x.f must accept three integer parameters*

In contrast, a typical compiler for a dynamically typed language will attempt to verify none of the above when the code is compiled. If x doesn't have a getValue method, or x and y can't be multiplied, or x.f requires three strings instead of three integers, there will be no warning of that until the program is run.

For years it has been widely held in industry that static typing is a must for reliable systems but a shift in thinking is underway. It is increasingly believed that good test coverage can produce equally reliable software.[1]

[1] http://www.artima.com/weblogs/viewpost.jsp?thread=4639

Ruby's philosophy is often "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?"

The instructor sees Ruby's philosophy as often being "Why not?" Here are some examples, involving overloaded operators:

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

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

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

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

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

Building blocks

nil

Strings

Numbers

Conversions

Arrays

The value nil

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

```
>> nil
=> nil

>> nil.class
=> NilClass

>> nil.object_id
=> 4
```

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

Speculate: Do uninitialized variables have the value nil?

Strings

Instances of Ruby's String class are used to represent character strings.

One way to specify a literal string is with double quotes. A variety of "escapes" are recognized:

```
>> "formfeed \f, newline \n, return \r, tab \t"
=> "formfeed \f, newline \n, return \r, tab \t"

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

> puts "newline >\n<, return (\r), tab >\t<"
newline >
), tab >        <
=> nil

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

Page 321 in the text has a full list of escapes.

Strings, continued

A string literal may be constructed using apostrophes instead of double quotes. If so, only \' and \\ are recognized as escapes:

```
>> '\n\t'.length        Four characters: backslash, n, backslash, t
=> 4

>> '\'\\'               Two characters: apostrophe, backslash
=> "'\\"

>> it.length
=> 2
```

A "here document" provides a third way to specify a string:

```
>> s = <<SomethingUnique
just
   testing
SomethingUnique
=> "just    \n    testing\n"
```

There's a fourth way, too: `%q{ just testin' this }` How many ways to do something is too many? Which are syntactic sugar?

Strings, continued

The public_methods (and methods) method show the public methods that are available for an object. Here are some of the methods for String:

```
>> "abc".public_methods.sort
=> ["%", "*", "+", "<", "<<", "<=", "<=>", "==", "===", "=~", ">", ">=", "[]", "[]=", "__id__",
"__send__", "all?", "any?", "between?", "capitalize", "capitalize!", "casecmp", "center",
"chomp", "chomp!", "chop", "chop!", "class", "clone", "collect", "concat", "count",
"crypt", "delete", "delete!", "detect", "display", "downcase", "downcase!", "dump",
"dup",  "each", "each_byte", "each_line", "each_with_index", "empty?", "entries",
"eql?", "equal?", "extend", "find", "find_all", "freeze", "frozen?", "gem", "grep", "gsub",
"gsub!", "hash", "hex", "id", "include?", "index", "inject", "insert", "inspect",
"instance_eval", "instance_of?", "instance_variable_get", "instance_variable_set",
"instance_variables", "intern", "is_a?", "kind_of?", "length", "ljust", "lstrip", "lstrip!",
"map", "match", "max", "member?", "method", "methods","min", "next", "next!", "nil?",
"object_id", "oct", "partition", "private_methods", "protected_methods",
"public_methods", "reject", "replace", "require", "require_gem", "respond_to?",
"reverse", "reverse!", "rindex", "rjust", "rstrip", "rstrip!", "scan", "select", "send",
...

>> "abc".public_methods.length
=> 145
```

Strings, continued

Unlike Java, ML, and many other languages, *strings in Ruby are mutable*. If two variables reference a string and the string is changed, the change is reflected by *both* variables:

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

>> y = x              x and y now reference the same instance of String.
=> "testing"

>> x.upcase!          Convention: If there are both applicative and imperative forms of a
                      method, the name of the imperative form ends with an exclamation.
=> "TESTING"

>> y
=> "TESTING"
```

In Java, if s1 and s2 are Strings an assignment such as s1 = s2 produces a shared reference but it's never an issue because instances of String are immutable—no methods change a String.

Strings, continued

The dup method produces a copy of a string.

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

>> y.downcase!
=> "testing"

>> y
=> "testing"

>> x
=> "TESTING"
```

Some objects that hold strings make a copy of the string when the string is added to the object.

Strings, continued

Strings can be compared with a typical set of operators:

```
>> s1 = "apple"
=> "apple"

>> s2 = "testing"
=> "testing"

>> s1 == s2
=> false

>> s1 != s2
=> true

>> s1 < s2
=> true

>> s1 >= s2
=> false
```

Strings, continued

There is also a *comparison operator*. It produces -1, 0, or 1 depending on whether the first operand is less than, equal to, or greater than the second operand.

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

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

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

Strings, continued

A individual character can be extracted from a string but note that the result is an integer character code (an instance of Fixnum), **not** a one-character string:

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

>> s[0]
=> 97             # 97 is the ASCII code for 'a'

>> s[1]
=> 98

>> s[-1]          # -1 is the last character, -2 is next to last, etc.
=> 99

>> s[100]         # Why not produce 0 for an out of bounds reference?
=> nil
```

Note that the position is zero-based. A negative value indicates an offset from the end of the string.

What's a good reason that Java provides s.charAt(n) instead of allowing s[n]?

Strings, continued

A subscripted string can be the target of an assignment.

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

>> s[0] = 65
=> 65

>> s[1] = "tomi"
=> "tomi"

>> s
=> "Atomic"
```

The numeric code for a character can be obtained by preceding the character with a question mark:

```
>> s[0] = ?B
=> 66
>> s
=> "Btomic"
```

Strings, continued

A substring can be referenced in several ways.

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

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

>> s[2,1]          Remember that s[n] yields a number, not a string.
=> "p"

>> s[2..-1]        2..-1 creates a Range object.  (More on ranges later.)
=> "place"

>> s[10,10]
=> nil

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

Speculate: What does s[1,100] produce? How about s[-1,-3]?

Strings, continued

A substring can be the target of assignment:

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

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

>> s
=> "place"

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

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

>> s
=> "plankton"
```

Strings, continued

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

```
>> x = 10
=> 10

>> y = "twenty"
=> "twenty"

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

>> s = "String methods: #{"abc".methods}".length
=> 896
```

The << operator appends to a string and produces the new string. *The string is changed.*

```
>> s = "just"
=> "just"
>> s << "testing" << "this"
=> "justtestingthis"
```

Numbers

On lectura, integers in the range -2^{30} to 2^{30}-1 are represented by instances of Fixnum. If an operation produces a number outside of that range, the value is represented with a Bignum.

```
>> x = 2**30-1          The exponentiation operator is **.
=> 1073741823

>> x.class
=> Fixnum

>> y = x + 1
=> 1073741824

>> y.class
=> Bignum

>> z = y - 1
=> 1073741823

>> z.class
=> Fixnum
```

How can we see what methods are available for instances of Fixnum?

Numbers, continued

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

```
>> x = 123.456
=> 123.456

>> x.class
=> Float

>> x ** 0.5
=> 11.1110755554987

>> x * 2e-3
=> 0.246912

>> x = x / 0.0
=> Infinity

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

Numbers, continued

Fixnums and Floats can be mixed. The result is a Float.

```
>> 10 / 5.1
=> 1.96078431372549

>> 10 % 4.5
=> 1.0

>> 2**40 / 8.0
=> 137438953472.0

>> it.class
=> Float
```

Other numeric classes in Ruby include BigDecimal, Complex, Rational and Matrix.

Conversions

Unlike many scripting languages, Ruby does not automatically convert strings to numbers and numbers to strings as needed:

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

The methods to_i, to_f, and to_s are used to convert values to Fixnums, Floats, and Strings, respectively

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

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

>> 10 + 20.9.to_i
=> 30

>> 2**100.to_f
=> 1.26765060022823e+030
```

Speculate: What does "123xyz".to_i produce?

Arrays

An ordered sequence of values is typically represented in Ruby by an instance of Array.

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

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

>> a2 = ["ten", 20, 30.0, 10**40]
=> ["ten", 20, 30.0, 10000000000000000000000000000000000000000]

>> a3 = [a1, a2, [[a1]]]
=> [[10, 20, 30], ["ten", 20, 30.0, 10000000000000000000000000000000000000000],
[[[10, 20, 30]]]]
```

What's a difference between a Ruby array and an ML list?

Arrays, continued

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

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

>> a[0]
=> 1

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

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

>> a[-1][-2][0]
=> 99
```

Note that %w{ ... } provides a way to avoid the tedium of surrounding each string with quotes. Experiment with it!

Arrays, continued

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

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

>> a[1] = "twenty"; a      Note: Semicolon separates expressions; a's value is shown.
=> [10, "twenty", 30, 40, 50, 60]

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

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

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

>> a[10] = [5,6]; a
=> [[1, 2, 3], nil, nil, nil, nil, nil, nil, nil, nil, nil, [5, 6]]
```

Arrays, continued

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

```
>> a = [ ]
=> [ ]

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

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

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

>> a[0].shift
=> 2

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

```
>> a,b = [1,2,3,4], [1,3,5]
=> [[1, 2, 3, 4], [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 == (a | b)[0..3]
=> true
```

Control structures

The while loop

Sidebar: Source code layout

Expressions or statements?

Logical operators

if-then-else

if and unless as modifiers

break and next

The for loop

The while loop

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

```
i = 1
while i <= 10
    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.

The control expression can be optionally followed by do or a colon. Example:

```
while i <= 10 do              # Or, while i <= 10 :
    puts i
    i += 1
end
```

What's a minor problem with Ruby's syntax versus Java's use of braces to bracket multi-line loop bodies?

while, continued

In Java, control structures like if, while, and for are driven by the result of expressions that produce a value whose type is boolean. C has a more flexible view: control structures consider an integer value that is non-zero to be "true".

<u>In Ruby, any value that is not false or nil is considered to be "true".</u>

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

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

Each call to gets returns a string that is the next line of the file. The string is assigned to line and like Java, assignment produces the value assigned. If the first line of the file is "one", then the first time through the loop, what's evaluated is while "one". The value "one" is not false or nil, so the body of the loop is executed and "one" is printed on standard output.

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

while, continued

On UNIX machines the string returned by gets has a trailing newline. The chomp method of String can be used to remove it.

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

```
result = ""

while line = gets.chomp
    result += line
end

puts result
```

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

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.[1]

[1] I bet some of you get this wrong the first time, like I did!

Sidebar: Source code layout

Unlike Java, Ruby does pay some attention to the presence of newlines in source code. For example, a `while` loop cannot be naively compressed to a single line. This does not work:

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

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

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

There is some middle ground, too:

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

Source code layout, continued

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

```
while i <=                      # OK because "i <=" is definitely incomplete
10 do puts i; i += 1 end

while i                    # NOT OK.  "while i" is complete, but then "<= 10"
<= 10 do puts i; i += 1 end     # is flagged as a syntax error.
```

There is a pitfall related to this rule. For 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, put an operator at the end of the line:

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

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

Expression or statement?

Academic writing on programming languages commonly uses the term "statement" to denote a syntactic element that performs an operation but does not produce a value. The term "expression" is consistently used to describe an operation that produces a value.

Ruby literature, including the text, sometimes talks about the "while statement" even though while produces a value:

```
>> i = 1
=> 1

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

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

The text sometimes uses the term "while loop" instead.

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

Logical operators

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

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

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

>> 1 && 2
=> 2

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

>> true && false
=> false

>> true && nil
=> nil
```

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

Logical operators, continued

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

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

>> false || 2
=> 2

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

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

>> s[0] || s[3]
=> 97

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

Logical operators, continued

Just like Java, an exclamation mark inverts a logical value. The resulting value is true or false.

```
>> ! true
=> false

>> ! 1
=> false

>> ! nil
=> true

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

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

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

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

The if-then-else construct

Ruby's if-then-else looks familiar:

```
>> 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
=> "three"
```

What can we say about it?

Speculate: What will 'if 1 > 2 then 3 end' produce?

if-then-else, continued

If there's no else clause and the control expression is false, nil is produced:

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

If a language provides for if-then-else to return a value it raises the issue of what if-then means.

- In the C family, if-then-else doesn't return a value.
- ML simply doesn't allow an else-less if.
- In Icon, an expression like if > 2 then 3 is said to *fail*. No value is produced and that failure propagates to any enclosing expression, which in turn fails.

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

if-then-else, continued

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

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

Speculate: Ruby has both || and or for disjunction. Why was or used above?

The elsif clause

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

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

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

It is not required to have a final else.

How could the code above be improved?

Is elsif syntactic sugar?

if and unless as modifiers

Conditional execution can be indicated by using if and unless as modifiers.

```
>> total, count = 123.4, 5
=> [123.4, 5]

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

>> total, count = 123.4, 0
=> [123.4, 0]

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

The general forms are:

expression1 if *expression2*
expression1 unless *expression2*

What does 'x.f if x' mean?

break and next

The break and next expressions are similar to break and continue in Java.

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

```
while line = gets

    if line[0] == ?. then
        break
    end

    if line[0] == ?# then next end

    puts line
end
```

Recall: (1) If s is a string, s[0] produces an integer. (2) The construct ?c produces the integer code of the character c.

Problem: Rewrite it to use if as a modifier.

break and next, continued

If an expression is specified with break, the value of the expression becomes the value of the while:

```
% cat break2.rb
s = "x"

puts (while true do
    break s if s.size > 30
    s += s
end)

% ruby break2.rb
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
%
```

Two other control flow modifiers used with loops are redo and retry.

The for loop

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

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

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

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

The "in" expression must be an object that has an each method. In the first case, the "in" expression is a Range. In the latter two it is an Array.

break and next have the same meaning as in a while loop.

The for loop, continued

The for loop supports parallel assignment:

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

Output:

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

Is it good or bad that the for loop specifically supports parallel assignment? How inconvenient would it be to do without it?

Of course, while, for, if-then-else and other statements can be arbitrarily interleaved and nested, just like in most languages.

Freestanding Methods

Basics

Where's the class?

Domain and range in Ruby

Varying numbers of arguments

Method definition

Here is a Ruby version of a simple method:

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

The keyword def indicates that a method definition follows. Next is the method name. The parameter list follows.

If the end of a method is reached without encountering a return, the value of the last expression becomes the return value. Here is an equivalent definition:

```
def double(x)
    x * 2
end
```

If no arguments are required, the parameter list can be omitted

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

Method definition, continued

One way to get a method into irb is to use load:

```
% cat double.rb
def double(x)
    x * 2
end

% irb --prompt simple
>> load "double.rb"
=> true

>> double(5)
=> 10
```

Method definition, continued

Alternatively, we can type a definition directly into irb:

```
% irb
irb(main):001:0> def double(x)
irb(main):002:1>     x * 2
irb(main):003:1> end
=> nil

irb(main):004:0> double(5)
=> 10

irb(main):005:0>
```

Note that irb was run without "--prompt simple". The default prompt includes a line counter and a nesting depth.

If double is a method, where's the class?

You may have noticed that even though we claim to be defining a method named double, there's no class in sight.

In Ruby, methods can be added to a class at run-time. A freestanding method defined in irb or found in a file is associated with an object referred to as "main", an instance of Object. At the top level, the name self references that object.

```
>> [self.class, self.to_s]
=> [Object, "main"]            # The class of self and a string representation of it.

>> methods_b4 = self.methods
=> ["methods", "popb", ...lots more...]

>> def double(x); x * 2 end
=> nil

>> self.methods - methods_b4
=> ["double"]
```

We can see that self has one more method (double) after double is defined.

Domain and range in Ruby

For reference:

```
def double(x)
    x * 2
end
```

For the ML analog of `double` the domain and range are the integers. (`int -> int`)

What is the domain and range of `double` in Ruby?

Domain and range in Ruby, continued

Problem: Write a method polysum(A) that produces a "sum" of the values in the array A.

Examples:

```
>> polysum([1,3,5])
=> 9

>> polysum([1.1,3.3,5.5])
=> 9.9

>> polysum(["one", "two"])
=> "onetwo"

>> polysum([["one"], [2,3,4], [[1],[1..10]]])
=> ["one", 2, 3, 4, [1], [1..10]]
```

How can we describe the domain and range of polysum?

Varying numbers of arguments

Unlike some scripting languages, Ruby considers it to be an error if the wrong number of arguments is supplied to a routine.

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

>> wrap("testing", "<>")
=> "<testing>"

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

>> wrap("testing", "<", ">")
ArgumentError: wrong number of arguments (3 for 2)
```

Contrast: Icon supplies &null (similar to Ruby's nil) for missing arguments. Extra arguments are ignored.

Varying numbers of arguments, continued

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

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

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

The imagined intention is that if `wrap` is called with one argument it will call the two-argument `wrap` with `"()"` as a second argument.

In fact, the second definition of `wrap` simply replaces the first. (Last `def` wins!)

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

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

Varying numbers of arguments, continued

There's no intra-class method overloading but Ruby does allow default values to be specified for arguments:

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

>> wrap("x", "<>")
=> "<x>"

>> wrap("x")
=> "(x)"
```

Varying numbers of arguments, continued

Any number of defaulting arguments can be specified. Imagine a method that creates a window:

```
def make_window(height = 500, width = 700,
        font = "Roman/12", upper_left = 0, upper_right = 0)
    ...
end
```

A variety of calls are possible. Here are some:

```
make_window

make_window(100, 200)

make_window(100, 200, "Courier/14")
```

Here's something that **DOES NOT WORK**:

```
make_window( , , "Courier/14")
```
Leading arguments can't be omitted!

Sidebar: A study in contrast

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

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

Here is wrap in Icon:

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

Varying numbers of arguments, continued

It can be useful to have a method take an arbitrary number of arguments. printf is a good example.

Here's a Ruby method that accepts any number of arguments and simply prints them:

```
def showargs(*args)

  printf("%d arguments:\n", args.size)

  for i in 0...args.size do                # a...b is a to b-1
      printf("#%d: %s\n", i, args[i])
    end
end
```

If a parameter is prefixed with an asterisk, an array is made of any remaining arguments.

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

Varying numbers of arguments, continued

Problem: Modify polysum so that this works:

```
>> polysum(1,2,3,4)        # Instead of polysum([1,2,3,4])
=> 10
```

Problem: Write a method printf0 that's like printf but simply interpolates argument values as a string (use to_s) where a percent sign is found:

```
>> printf0("x = %, y = %, z = %\n", 10, "ten", "z")
x = 10, y = ten, z = z
=> 23

>> printf0("testing\n")
testing
=> 8
```

Varying numbers of arguments, continued

Sometimes we want to call a method with the values in an array:

```
>> def add(x,y) x + y end

>> pair = [3,4]

>> add(pair[0], pair[1])
=> 7
```

Here's an alternative:

```
>> add(*pair)
=> 7
```

In a method call, prefixing an array value with an asterisk causes the values in the array to become a sequence of parameters.

Speculate: What will be the result of `add(*[1,2,3])`?

Varying numbers of arguments, continued

Recall make_window:

```
def make_window(height = 500, width = 700,
        font = "Roman/12", upper_left = 0, upper_right = 0)
        ...printf to echo the arguments...
end
```

Results of array-producing methods can be passed easily to make_window:

```
>> where = get_loc(...whatever...)
=> [50, 50]

>> make_window(100, 200, "Arial/8", *where)
make_window(height = 100, width = 200, font = Arial/8, at = (50, 50)

>> win_spec = get_spec(...whatever...)
=> [100, 200, "Courier/9"]

>> make_window(*win_spec)
make_window(height = 100, width = 200, font = Courier/9, at = (0, 0)
```

Speculate: Will make_window(*[300,400], "x", *[10,10]) work?

Iterators and blocks

Using iterators and blocks

Iterate with each or use a for loop?

Creating iterators

Iterators and blocks

Some methods are *iterators*. An iterator that is implemented by the Array class is each. each iterates over the elements of the array. Example:

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

>> x.each { puts "element" }
element
element
element
=> [10, 20, 30]
```

The construct { puts "element" } is a *block*. Array#each invokes the block once for each of the elements of the array.

Because there are three values in x, the block is invoked three times and "element" is printed three times.

Speculate: What does (1..50).each { putc ?x } do?

Iterators and blocks, continued

Iterators can pass one or more values to a block as arguments. Array#each passes each array element in turn.

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

We might print the values in an array like this:

```
>> [10, "twenty", 30].each { |e| printf("element: %s\n", e) }
element: 10
element: twenty
element: 30
```

A note about the format %s: In C, the value of the corresponding parameter must be a pointer to a zero-terminated sequence of char values. Ruby is more flexible—%s causes to_s to be invoked on the corresponding value. The result of to_s is used.

Another possibility for the format is %p, which causes inspect to be invoked. However, the second line above would be element: "twenty".

Iterators and blocks, continued

For reference:

```
[10, "twenty", 30].each { |e| printf("element: %s\n", e) }
```

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

Examples:

```
>> sum = 0

>> [10, "twenty", 30].each { ??? }
>> sum
=> 40

>> sum = 0

>> (1..100).each { ??? }

>> sum
=> 5050
```

Sidebar: Iterate with each or use a for loop?

You may recall that the for loop requires the result of the "in" expression to have an each method. Thus, we always have a choice between a for loop,

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

and iteration with each,

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

Which is better?

Iterators and blocks, continued

Array#each is typically used to create side effects of interest, like printing values or changing variables but in many cases it is the value returned by an iterator that is of principle interest.

See if you can describe what each of the following iterators is doing.

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

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

>> ["burger", "fries", "shake"].sort { |a,b| a[-1] <=> b[-1] }
=> ["shake", "burger", "fries"]

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

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

Iterators and blocks, continued

We have yet to study inheritance in Ruby but we can query the ancestors of a class like this:

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

Because an instance of Array is an Enumerable, we can apply iterators in Enumerable to arrays:

```
>> [2, 4, 5].any? { |n| n % 2 == 0 }
=> true

>> [2, 4, 5].all? { |n| n % 2 == 0 }
=> false

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

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

Iterators and blocks, continued

Many classes have iterators. Here are some examples:

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

>> "abc".each_byte { |b| puts b }
97
98
99

>> (1..50).inject(1) { |product, i| product  * i }
=> 30414093201713378043612608166064768844377641568960512000000000000
```

To print each line in the file x.txt, we might do this:

```
IO.foreach("x.txt") { |line| puts line }
```

A quick way to find the iterators for a class is to search for "block" in the documentation.

Blocks and iterators, continued

As you'd expect, blocks can be nested. Here is a program that reads lines from standard input, assumes the lines consist of integers separated by spaces, and averages the values.

```
total = n = 0
STDIN.readlines().each {
    |line|
    line.split(" ").each {
        |word|
        total += word.to_i
        n += 1
        }
    }

printf("Total = %d, n = %d, Average = %g\n", total, n, total / n.to_f) if n != 0
```

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

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

Notes:

- STDIN represents "standard input". It is an instance of IO.
- STDIN.readlines reads standard input to EOF and returns an array of the lines read.
- The printf format specifier %g indicates to format the value as a floating point number and select the better of fixed point or exponential form based on the value.

Some details on blocks

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

```
a.each do
    |element|
    printf("element: %s\n", element)
  end
```

do/end has lower precedence than braces but that only becomes an issue if the iterator is supplied an argument that is not enclosed in parentheses. (Good practice: enclose iterator argument(s) in parentheses, as shown in these slides.)

Note that do, {, or a backslash (to indicate continuation) must appear on the same line as the iterator invocation. The following will produce an error

```
a.each
  do                    # "LocalJumpError: no block given"
    |element|
    printf("element: %s\n", element)
  end
```

Some details on blocks, continued

Blocks raise issues with the scope of variables. If a variable is created in a block, the scope of the variable is limited to the block:

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

>> [1].each { x = 10 }     => [1]

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

If a variable already exists, a reference in a block is resolved to that existing variable.

```
>> x = "test"              => "test"

>> [1].each { x = 10 }     => [1]

>> x                       => 10
```

Sometimes you want that, sometimes you don't. It's said that this behavior may change with Ruby 2.0.

Creating iterators with yield

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

The yield expression invokes the block associated with the current method invocation.

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

```
def simple()
    puts "simple: Starting up..."
    yield 3

    puts "simple: More computing..."
    yield 7

    puts "simple: Out of values..."
    "simple result"
end
```

Usage:

```
>> simple() { |x| printf("\tx = %d\n", x) }
simple: Starting up...
        x = 3
simple: More computing...
        x = 7
simple: Out of values...
=> "simple result"
```

The iterator (simple) prints a line of output, then calls the block with the value 3. The iterator prints another line and calls the block with 7. It prints one more line and then returns, producing "simple result" as the value of simple() { |x| printf("\tx = %d\n", x) }.

Notice how the flow of control alternates between the iterator and the block.

yield, continued

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

```
>> from_to(1,10) { |i| puts i }
1
2
...
10
=> nil

>> from_to(0,100,25) { |i| puts i }
0
25
50
75
100
=> nil
```

yield, continued

If a block is to receive multiple arguments, just specify them as a comma-separated list for yield.

Here's an iterator that produces consecutive pairs of elements from an array:

```
def elem_pairs(a)
    for i in 0..(a.length-2)
        yield a[i], a[i+1]
    end
end
```

Usage:

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

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

yield, continued

Recall that Array#select produces the elements for which the block returns true:

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

Speculate: How is the code in select accessing the result of the block?

yield, continued

The last expression in a block becomes the value of the yield that invoked the block.

Here's how we might implement a function-like version of select:

```
def select(enumerable)
    result = [ ]
    enumerable.each do
        |element|
        if yield element then
            result << element
        end
    end
    return result
end
```

Usage:

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

Note the we pass the array as an argument instead of invoking the object's select method.

yield, continued

Problem: Implement in Ruby an analog for ML's foldr.

```
>> foldr([10,20,30], 0) { |e, thus_far| e + thus_far }
=> 60

>> foldr([10,20,30], 0) { |e, thus_far| 1 + thus_far }
=> 3

>> foldr([5, 1, 7, 2], 0) { |e, max| e > max ? e : max }
=> 7
```

Here's a weakness in the instructor's implementation:

```
>> foldr(1..10, [ ]) { |e,thus_far| thus_far + [e] }
NoMethodError: undefined method `reverse_each' for 1..10:Range
```

What can we learn from it?

A Batch of Odds and Ends

Constants

Symbols

The Hash class

Constants

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

Ruby allows a constant to be changed but a warning is generated:

```
>> A = 1
=> 1

>> A = 2; A
(irb): warning: already initialized constant A
=> 2
```

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

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

>> L << 30; L
=> [10, 20, 30]

>> L = 1
(irb): warning: already initialized constant L
```

Constants, continued

You may have noticed that the names of all the standard classes are capitalized. That's not simply a convention; Ruby requires class names to be capitalized.

```
>> class b
>> end
SyntaxError: compile error
(irb): class/module name must be CONSTANT
```

If a method is given a name that begins with a capital letter, it can't be found:

```
>> def M; 10 end
=> nil
>> M
NameError: uninitialized constant M
```

Constants, continued

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

ARGV

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

FALSE, TRUE, NIL

Synonyms for `false`, `true`, and `nil`.

STDIN, STDOUT

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

Symbols

An identifier preceded by a colon creates a Symbol. A symbol is much like a string but a given identifier always produces the same symbol:

```
>> s = :length                 => :length

>> s.object_id                 => 42498

>> :length.object_id           => 42498
```

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

```
>> "length".object_id          => 23100890

>> "length".object_id          => 23096170
```

If you're familiar with Java's String.intern method, note that Ruby's String#to_sym is roughly equivalent:

```
>> "length".to_sym.object_id       => 42498
```

For the time being, it's sufficient to simply know that :*identifier* creates a Symbol.

The Hash class

Ruby's Hash class is similar to Hashtable and Map in Java. It can be thought of as an array that can be subscripted with values of any type, not just integers.

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

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

>> numbers.class              => Hash
```

Subscripting a hash with a "key" and assigning a value to it stores that key/value pair in the hash:

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

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

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

>> numbers.size               => 2
```

Hash, continued

At hand:

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

To fetch the value associated with a key, simply subscript the hash with the key. If the key is not found, nil is produced.

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

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

The value associated with a key can be changed via assignment. A key/value pair can be removed with Hash#delete.

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

>> numbers.delete("one")          => 1        # The associated value, if any, is
                                              #  returned.

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

Speculate: What is the net result of numbers["two"] = nil?

Hash, continued

There are no restrictions on the types that can be used for keys and values.

```
>> h = { }                  => { }

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

>> h[true] = { }            => { }

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

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

>> h[h[1000] + [3]] << 40   => [4, 40]

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

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

Hash, continued

It was said earlier that if a key is not found, nil is returned. That was a simplification. In fact, the *default value* of the hash is returned if the key is not found.

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

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

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

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

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

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

It is not discussed here but there is also a form of Hash#new that uses a block to produce default values.

Hash example: tally.rb

Here is a program that reads lines from standard input and tallies the number of occurrences of each word. The final counts are dumped with inspect.

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

STDIN.readlines.each {
    |line|
    line.split(" ").each {
        |word|
        counts[word] += 1
        }
    }
puts counts.inspect                # Equivalent: p counts
```

Usage:

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

tally.rb, continued

The output of puts counts.inspect is not very user-friendly:

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

Hash#sort produces a list of key/value lists ordered by the keys, in ascending order:

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

Problem: Produce nicely labeled output, like this:

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

tally.rb, continued

At hand:

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

Solution:

```
([["Word","Count"]] + counts.sort).each {
    |k,v| printf("%-10s\t%5s\n", k, v)         # %-10s left-justifies in a field of width 10
    }
```

As a shortcut for easy alignment, the column headers are put at the start of the list. Then, we use %5s instead of %5d to format the counts and accommodate "Count". (Recall that this works because %s causes to_s to be invoked on the value.)

Is the shortcut a "programming technique" or a hack?

tally.rb, continued

Hash#sort's default behavior of ordering by keys can be overridden by supplying a block.

The block is repeatedly invoked with two arguments: a pair of list elements.

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

The block is to return -1, 0, or 1 depending on whether a is considered to be less than, equal to, or greater than b.

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

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

How could we put ties on the count in ascending order by the words? Example:

```
[["be", 2], ["to", 2], ["not", 1], ["or", 1]]
```

Hash initialization

It is tedious to initialize a hash with a series of assignments:

```
numbers = { }
numbers["one"] = 1
numbers["two"] = 2
...
```

Ruby provides a shortcut:

```
>> numbers = { "one", 1, "two", 2, "three", 3 }
=> {"three"=>3, "two"=>2, "one"=>1}
```

There's a more verbose variant, too:

```
>> numbers = { "one" => 1, "two" => 2, "three" => 3 }
=> {"three"=>3, "two"=>2, "one"=>1}
```

One more option: (but note that both keys and values are strings)

```
>> Hash[ * %w/a 1 b 2 c 3 d 4 e 5/ ]
=> {"a"=>"1", "b"=>"2", "c"=>"3", "d"=>"4", "e"=>"5"}
```

Regular Expressions

A little theory

Good news and Bad news

The match operator

Character classes

Alternation and grouping

Repetition

split and scan

Anchors

Grouping and references

Iteration with gsub

Application: Time totaling

A little theory

In computer science theory, a language is a set of strings. The set may be infinite.

The Chomsky hierarchy of languages looks like this:

Unrestricted languages	("Type 0")
Context-sensitive languages	("Type 1")
Context-free languages	("Type 2")
Regular languages	("Type 3")

Roughly speaking, natural languages are *unrestricted languages* that can only specified by *unrestricted grammars*.

Programming languages are usually *context-free languages*—they can be specified with a *context-free grammar*, which has very restrictive rules. Every Java program is a string in the context-free language that is specified by the Java grammar.

A *regular language* is a very limited kind of context free language that can be described by a *regular grammar*. A regular language can also be described by a *regular expression*.

A little theory, continued

A regular expression is simply a string that may contain metacharacters. 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 have ab repeated some number of times followed by zero or more c's. Some strings in the language are ab, ababc, and abababababccccccc.

The regular expression

```
(north|south)(east|west)
```

describes a language with four strings: {northeast, northwest, southeast, southwest}.

Good news and bad news

UNIX tools such as the ed editor and grep/fgrep/egrep introduced regular expressions to a wide audience.

Many languages provide a library for working with regular expressions. Java provides the java.util.regex package. The command man regex produces some documentation for the C library's regular expression routines.

Some languages, Ruby included, have a regular expression datatype.

Regular expressions have a sound theoretical basis and are also very practical. Over time, however, a great number of extensions have been added. In languages like Ruby, regular expressions are truly a language within a language.

Chapter 22 of the text devotes four pages to its summary of regular expressions. In contrast, integers, floating point numbers, strings, ranges, arrays, and hashes are summarized in a total of four pages.

Good news and Bad news, continued

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!#$%&'*+\-/=?^_`{|}~]+\x20*|"((?=[\x01-\x7f])[^"\\]|\\[\x01-\x7f])*"\x20*)*(?
<angle><))?((?!\.)(?>\.?[a-zA-Z\d!#$%&'*+\-/=?^_`{|}~]+)+|"((?=[\x01-\x7f])[^"\\]|\\[\x01-\
x7f])*")@(((?!-)[a-zA-Z\d\-]+(?<!-)\.)+[a-zA-Z]{2,}|\[(((?(?<!\[)\.)(25[0-5]|2[0-4]\d|[01]?\d?
\d)){4}|[a-zA-Z\d\-]*[a-zA-Z\d]:((?=[\x01-\x7f])[^\\\[\]]|\\[\x01-\x7f])+)\])(?(angle)>)$
```

It describes RFC 2822 email addresses.

The instructor believes that regular expressions have their place but grammar-based parsers should be considered more often than they are, especially when an underlying specification includes a grammar.

We'll cover a subset of Ruby's regular expression capabilities.

A simple regular expression in Ruby

One way to create a regular expression (RE) in Ruby is to use the /*pattern*/ syntax:

```
>> re = /a.b.c/         => /a.b.c/

>> re.class             => Regexp
```

In an RE, a dot is a *metacharacter* (a character with special meaning) that will match any (one) character.

Alphanumeric characters and some special characters simply match themselves.

The meaning of a metacharacter can be suppressed by preceding with a backslash.

The RE /a.b.c/ matches strings that contain the five-character sequence a<*anychar*>b<*anychar*>c, like "albacore", "barbecue", "drawback", and "iambic".

How many strings are in the language specified with the regular expression /a.b.c/?

The match operator

The binary operator =~ is called "match". One operand must be a string and the other must be a regular expression. If the string contains a match for the RE, the position of the match is returned. nil is returned if there is no match.

```
>> "albacore" =~ /a.b.c/          => 0

>> /a.b.c/ =~ "drawback"          => 2

>> "abc" =~ /a.b.c/               => nil
```

What does the following loop do?

```
while line = gets do
    puts line if line =~ /a.b.c/
end
```

How could we invert the operation of the loop?

Problem: Write a program that prints lines longer than the length specified by a command line argument. For example, `longerthan 80 < x` prints the lines in x that are 81 characters or more in length. (Don't use String#length or size!)

The match operator, continued

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

$`	Is the portion of the string that precedes the match. (That's a backquote.)
$&	Is the portion of the string that was matched by the regular expression.
$'	Is the portion of the string following the match.

Example:

```
>> "limit=300" =~ /=/          => 5

>> $`                          => "limit"

>> $&                          => "="

>> $'                          => "300"
```

The match operator, continued

Here is a handy utility routine from the text:

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

Usage:

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

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

Handy: Put show_match in your ~/.irbrc file. Maybe name it sm.

Regular expressions as subscripts

As a subscript, a regular expression specifies the portion of the string, if any, matched by it.

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

>> s[/.../] = "*"           => "*"

>> s                        => "*ting"
```

Another example:

```
>> %w{albacore drawback iambic}.map { |w| w[/a.b.c/] = "(a.b.c)"; w }
=> ["(a.b.c)ore", "dr(a.b.c)k", "i(a.b.c)"]
```

Character classes

The pattern [*characters*] is an RE that matches any one of the specified *characters*.

[^characters] is an RE that matches any character not in the set. (It matches the *complement* of the set.)

A dash between two characters in a set specifies a range based on ASCII codes.

Examples:

/[aeiou]/ matches a string that contains a lower-case vowel

```
>> show_match("testing", /[aeiou]/)
=> "t<<e>>sting"
```

/[^0-9]/ matches a string that contains a non-digit

```
>> show_match("1,000", /[^0-9]/)
=> "1<<,>>000"
```

/[a-z][0-9][a-z]/ matches strings that somewhere contain the three-character sequence lowercase letter, digit, lowercase letter.

```
>> show_match("A1b33s4ax1", /[a-z][0-9][a-z]/)
=> "A1b33<<s4a>>x1"
```

Character classes, continued

Ruby provides abbreviations for some commonly used character classes:

\d	Stands for [0-9]
\w	Stands for [A-Za-z0-9_]
\s	Whitespace—blank, tab, carriage return, newline, formfeed

The abbreviations \D, \W, and \S produce a complemented set for the corresponding class.

Examples:

```
>> show_match("Call me at 555-1212", /\d\d\d-\d\d\d\d/)
=> "Call me at <<555-1212>>"

>> "fun double(n) = n * 2".gsub(/\w/,".")
=> "... ......(.) = . * ."

>> "FCS 202, 12:30-13:45 TH".gsub(/\D/, "~")
=> "~~~~202~~12~30~13~45~~~"
```

gsub's replacement string can be any length, as you'd expect.

Alternatives and grouping

Alternatives can be specified with a vertical bar:

```
>> %w{one two three four}.select { |s| s =~ /two|four|six/ }
=> ["two", "four"]
```

Parentheses can be used for grouping. Consider this regular expression:

```
/(two|three) (apple|biscuit)s/
```

It corresponds to a regular language with four strings:

```
two apples
three apples
two biscuits
three biscuits
```

Usage:

```
>> "I ate two apples." =~ /(two|three) (apple|biscuit)s/      => 6

>> "She ate three mice." =~ /(two|three) (apple|biscuit)s/ => nil
```

Creating regular expressions at run-time

The method Regexp.new(s) creates a regular expression from the string s.

```
counts = %w{two three four five}
foods = %w{apples oranges bananas}

re = ""; sep = ""
counts.each {
    |count| foods.each {
        |food|
        re << sep << count << " " << food
        sep = "|"
    }
}
puts re
re = Regexp.new(re)
while line = (printf("Query? "); gets)
    if line =~ re then
        puts "Yes: #{$`}[#{$&}]#{$'}"
    else puts "No"
    end
end
```

Execution:

```
% ruby re2.rb
two apples|two oranges|two bananas|three
apples|three oranges|three bananas|four
apples|...

Query? Are there four apples?
Yes: Are there [four apples]?

Query? We sold two bananas.
Yes: We sold [two bananas].

Query? Three oranges were thrown at me!
No
```

Repetition with *, +, and ?

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.

Examples:

/ab*c/ Matches strings that contain an 'a' that is followed by zero or more 'b's that are followed by a 'c'. Examples: ac, abc, abbbbbbc, back, and cache.

/-?\d+/ Matches strings that contain an integer. What strings are matched by /-?\d*/? What would show_match("maybe --123.4e-10 works", /-?\d+/) produce?

/a(12|21|3)*b/
Matches strings like ab, a3b, a312b, and a3123213123333b.

Repetition, continued

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

Repetition, continued

More examples of greed:

```
>> show_match("xabbbbc", /a.*b/)           => "x<<abbbb>>c"

>> show_match("xabbbbc", /ab?b?/)          => "x<<abb>>bbc"

>> show_match("xabbbbc", /ab?b?.*c/        => "x<<abbbbc>>"

>> show_match("maybe --123.4e-10 works", /-?\d+/)
=> "maybe -<<-123>>.4e-10 works"
```

Why are `*`, `+`, and `?` greedy?

Repetition, continued

Describe the strings matched by...

```
/[a-z]+[0-9]?/
/a...b?c/
/..1.*2../
/..*.+.*./
/((ab)+c?(xyz)*)?/
```

Specify an RE that matches...

Strings corresponding to ML int lists, like [10], [5,1,~700], and [ ]. Assume there are no embedded spaces.

Lines that contain only whitespace and a left or right brace.

Strings that match /^[A-Za-z_]\w*$/ commonly occur in programs. What are they?

split and scan with regular expressions

It is possible to split a string on a regular expression:

```
>> "  one, two,three / four".split(/[\s,\/]+/)    # Note escaped backslash in class
=> ["", "one", "two", "three", "four"]
```

Unfortunately, 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:

```
>> "10.0/-1.3...5.700+[1.0,2.3]".scan(/-?\d+\.\d+/)
=> ["10.0", "-1.3", "5.700", "1.0", "2.3"]
```

Here's a way to keep all the pieces:

```
>> "  one, two,three / four".scan(/(\w+|\W+)/)
=> [["  "], ["one"], [", "], ["two"], [","], ["three"], [" / "], ["four"]]
```

A list of lists is produced because of the grouping. We'll see a use for this later.

Anchors

The metacharacter ^ is an *anchor*. It doesn't match any characters but it constrains the following regular expression to appear at the beginning of the string being matched against.

Another anchor is $. It constrains the preceding regular expression to appear at the end of the string.

```
 $ grep.rb  ^bucket  <  $words
bucket
bucketed
bucketeer

$ grep.rb  bucket$  <  $words
bucket
gutbucket
trebucket
```

Problems:

Specify an RE that will match words that are at least six characters long, start with an 'a', and end with a 'z'.

Count the number of empty lines in x.rb. (Yes, you can't use String#size!)

Groups and references

In addition to providing a way to override precedence rules, parentheses create *references* (also called *back references*) to the text matched by a group.

Here is a regular expression that matches strings consisting of digits where the first and last digit are the same:

```
/^(\d)\d*\1$/
```

Piece by piece:

`^`	Require the following RE to be at the beginning of the string.
`(\d)`	Match one digit and retain it as the text of "group 1".
`\d*`	Match zero or more digits.
`\1`	The text of group 1.
`$`	Require the preceding RE to be at the end of the string.

Groups and references, continued

For reference:

```
/^(\d)\d*\1$/
```

Usage:

```
>> show_match("121", /^(\d)\d*\1$/)          => "<<121>>"

>> show_match("12", /^(\d)\d*\1$/)           => "no match"

>> show_match("3013", /^(\d)\d*\1$/)         => "<<3013>>"

>> show_match("3", /^(\d)\d*\1$/)            => "no match"
```

A little fun:

```
>> (1000..2000).select { |n| (7**n).to_s =~ /^(\d)\d*\1$/ }
=> [1000, 1012, 1020, 1021, 1023, 1032, 1044, 1046, 1052, 1053, 1055, 1064, 1075,
1084, 1096, 1107, 1116, 1128, 1130, 1136, 1137, 1139, 1148, 1168, 1180, 1191,
1200, 1212, 1220, 1221, 1223, 1232, 1246, 1252, 1255, 1264, 1275, 1284, ... ]
```

Groups and references, continued

In addition to setting $`, $&, and $', a successful match also sets $1, $2, ..., $9 to the text of the corresponding group.

Strictly to illustrate the mechanism, here is a method that swaps the first three and last characters of a string:

```
def swap3(s)
    if s =~ /(...)(.*)(...)/ then
        "#{$3}#{$2}#{$1}"
    else
        s
    end
end
```

Usage:

```
>> swap3 "abc-def"          => "def-abc"
>> swap3 "aaabbb"           => "bbbaaa"
>> swap3 "abcd"             => "abcd"
```

In actual practice what's a better way to perform this computation?

Groups and references, continued

As a more practical example, here is a method that rewrites infix operators as function calls:

```
$ops = { "-" => "sub", "+" => "add", "mul" => "mul", "div" => "div" }  # global variable
def infix_to_function(line)
    if line =~ /^(\w+)\s*(([-+]|(mul|div)))\s*(\w+)$/ then
        fcn = $ops[$2]
        return "#{fcn}(#{$1},#{$5})"
    else
        return nil
    end
end
```

Usage:

```
>> infix_to_function("3 + 4")             => "add(3,4)"

>> infix_to_function("limit-1500")        => "sub(limit,1500)"

>> infix_to_function("10mul20")           => "mul(10,20)"
```

Could we generate the regular expression from the hash? Do we really need a character class or would just alternation suffice?

Groups and references, continued

If the argument to scan has one or more groups, a list of lists is produced:

```
>> "1:2  33:28  100:7".scan(/(\d+):(\d+)/)
=> [["1", "2"], ["33", "28"], ["100", "7"]]

>> "1234567890".scan(/(.)(.)(.)/)
=> [["1", "2", "3"], ["4", "5", "6"], ["7", "8", "9"]]
```

Recall this example:

```
>> "  one, two,three / four".scan(/(\w+|\W+)/)
=> [["  "], ["one"], [", "], ["two"], [","], ["three"], [" / "], ["four"]]
```

Iteration with gsub

Recall String#gsub:

```
>> "fun double(n) = n * 2".gsub(/\w/,".")
=> "... ......(.) = . * ."
```

gsub has a one argument form that is an iterator. The result of the block is substituted for the match.

Here is a method that augments a string with a running sum of the numbers it contains:

```
def running_sums(s)
    sum = 0
    s.gsub(/\d+/) {
        sum += $&.to_i
        $& + "(%d)" % sum
        }
end
```

Usage:

```
>> running_sum("1 pencil, 3 erasers, 2 pens")
=> "1(1) pencil, 3(4) erasers, 2(6) pens"
```

Application: Time totaling

Consider an application that reads elapsed times on standard input and prints their total:

```
% ttl.rb
3h
15m
4:30
^D
7:45
```

Multiple times can be specified per line:

```
% ruby ttl.rb
10m, 20m
3:30 2:15 1:01
^D
7:16
```

Times in an unexpected format are ignored:

```
% ttl.rb
10 2:90
What's 10?  Ignored...
What's 2:90?  Ignored...
```

Time totaling, continued

```
def main
    mins = 0
    while line = gets do
        line.scan(/[^\s,]+/).each {|time| mins += parse_time(time) }
    end
    printf("%d:%02d\n", mins / 60, mins % 60)
end

def parse_time(s)
    case
    when s =~ /^(\d+):([0-5]\d)$/
        $1.to_i * 60 + $2.to_i
    when s =~ /^(\d+)([hm])$/
        if $2 == "h" then  $1.to_i * 60
                  else $1.to_i end
    else
        print("What's #{s}?  Ignored...\n");  0
    end
end
main
```

Class definition

Counter: A tally counter

An interesting thing about instance variables

Addition of methods

An interesting thing about class defintions

Sidebar: Fun with eval

Class variables and methods

A little bit on access control

Getters and setters

A tally counter

Imagine a class named Counter that models a tally counter.

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

```
c1 = Counter.new
c1.click
c1.click
puts c1            # Output: Counter's count is 2
c1.reset

c2 = Counter.new "c2"
c2.click
puts c2            # Output: c2's count is 1

c2.click
printf("c2 = %d\n", c2.count)   # Output: c2 = 2
```

Counter, continued

Here is a partial implementation of Counter:

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

The reserved word class begins a class definition; a corresponding end terminates it. A class name must begin with a capital letter.

The method name initialize is special. It identifies the method that is called when the method new is invoked:

```
c1 = Counter.new

c2 = Counter.new "c2"
```

If no argument is supplied to new, the default value of "Counter" is used.

Obviously, initialize is the counterpart to a constructor in Java.

Counter, continued

For reference:

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

The constructor initializes two instance variables: @count and @label.

Instance variables are identified by prefixing them with @.

An instance variable comes into existence when a value is assigned to it.

Just like Java, each object has its own copy of instance variables.

Unlike variables local to a method, instance variables have a default value of nil.

Counter, continued

For reference:

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

When irb displays an object, the instance variables are shown:

```
>> a = Counter.new "a"
=> #<Counter:0x2c61eb4 @label="a", @count=0>

>> b = Counter.new
=> #<Counter:0x2c4da04 @label="Counter", @count=0>

>> [a,b]
=> [#<Counter:0x2c61eb4 @label="a", @count=0>,
     #<Counter:0x2c4da04 @label="Counter", @count=0>]
```

Counter, continued

Here's the full source:

```
class Counter
    def initialize(label = "Counter")
        @count = 0; @label = label
    end
    def click
        @count += 1
    end
    def reset
        @count = 0
    end
    def count              # Note the convention: count, not get_count
        @count
    end
    def to_s
        return "#{@label}'s count is #{@count}"
    end
end
```

A very common error is to omit the @ on a reference to an instance variable.

An interesting thing about instance variables

Consider this class:

```
class X
    def initialize(n)
        case n
        when 1 then @x = 1
        when 2 then @y = 1
        when 3 then @x = @y = 1
        end
    end
end
```

What's interesting about the following?

```
>> X.new 1                  => #<X:0x2c26a44 @x=1>

>> X.new 2                  => #<X:0x2c257d4 @y=1>

>> X.new 3                  => #<X:0x2c24578 @x=1, @y=1>
```

Addition of methods

In Ruby, a method can be added to a class without changing the source code for the class. In the example below we add a label method to Counter, to fetch the value of the instance variable @label.

```
>> c = Counter.new "ctr 1"
=> #<Counter:0x2c26bac @label="ctr 1", @count=0>

>> c.label
NoMethodError: undefined method `label' for #<Counter:0x2c26bac @label="ctr 1",
@count=0>
        from (irb):4
>> class Counter
>>     def label
>>        @label
>>     end
>> end
=> nil

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

What are the implications of this capability?

Addition of methods, continued

We can add methods to built-in classes!

```
% cat hexstr.rb
class Fixnum
    def hexstr
        return "%x" % self
    end
end
```

Usage:

```
>> load "hexstr.rb"        => true

>> 15.hexstr               => "f"

>> p (10..20).collect { |n| n.hexstr }
["a", "b", "c", "d", "e", "f", "10", "11", "12", "13", "14"]
=> nil
```

An interesting thing about class definitions

Observe the following. What does it suggest to you?

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

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

>> class X; puts "here"; end
here
=> nil
```

Class definitions are executable code

In fact, a class definition is executable code. Consider the following, which uses a case statement to selectively execute defs for methods.

```
class X
    print "What methods would you like? "
    methods = gets.chomp
    methods.each_byte { |c|
        case c
        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
```

Execution:

```
What methods would you like? fg
>> c = X.new                                => #<X:0x2c2a1e4>
>> c.f                                      => "from f"
>> c.h
NoMethodError: undefined method `h' for #<X:0x2c2a1e4>
```

Sidebar: Fun with eval

Kernel#eval **parses a string containing Ruby source code and executes it.**

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

>> n = 3                      => 3

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

>> x                          => "abcabcabc"

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

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

Look carefully at the above. Note that eval uses variables from the current environment and that an assignment to x is reflected in the environment.

Bottom line: A Ruby program can generate code for itself.

Sidebar, continued

Problem: Create a file new_method.rb with a class X that prompts the user for a method name, parameters, and method body. It then creates that method. Repeat.

```
>> load "new_method.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? a
What shall it do? a[-1]
Method last(a) added to class X

What method would you like? ^D

>> c = X.new               => #<X:0x2c2980c>

>> c.add(3,4)              => 7

>> c.last [1,2,3]          => 3
```

Sidebar, continued

Solution:

```
class X
    while true
        print "What method would you like? "
        name = gets || break
        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?

Class variables and methods

Just as 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:

```
class Counter
    @@created = 0                  # Must precede any use of @@created

    def initialize(label = "Counter")
        @count = 0; @label = label
        @@created += 1
    end

end
```

Note: Unaffected methods are not shown.

Class variables and methods, continued

To define a class method, simply prefix the method name with the name of the class:

```
class Counter
  @@created = 0

  ... other methods ...

  def Counter.created          # class method
    return @@created
  end
end
```

Usage:

```
>> Counter.created              => 0
>> c = Counter.new              => #<Counter:0x... @label="Counter", @count=0>
>> Counter.created              => 1
>> 5.times { Counter.new }      => 5
>> 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.

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

  private
    def g; puts "in g" end
end

>> x = X.new                  => #<X:0x2c0cc84>
>> x.f
in f
in g

>> x.g
NoMethodError: private method `g' called for #<X:0x2c0cc84>
```

In Ruby, there is simply no such thing as a public class variable or public instance variable. All access must be through methods.

Getters and setters

If Counter were in Java, we might provide methods like void setCount(int n) and int getCount().

In Counter we provide a method called count to fetch the count.

Instead of something like setCount, we'd do this:

```
def count= n            # IMPORTANT: Note the trailing '='
        print("count=(#{n}) called\n")
        @count = n unless n < 0
end
```

Usage:

```
>> c = Counter.new      => #<Counter:0x2c94094 @label="Counter", @count=0>

>> c.count = 10
count=(10) called

>> c                    => #<Counter:0x2c94094 @label="Counter", @count=10>
```

Getters and setters, continued

Here's class to represent points on a 2d Cartesian plane:

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

Usage:

```
>> p1 = Point.new(3,4)          => #<Point:0x2c72c78 @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:

```
class Point
    def initialize(x, y)
        @x = x
        @y = y
    end
    attr_reader :x, :y          # :x and :y are Symbols.  (But "x" and "y" work, too!)
end
```

Usage:

```
>> p = Point.new(3,4)    => #<Point:0x2c25478 @x=3, @y=4>

>> p.x                   => 3

>> p.y                   => 4

>> p.x = 10
NoMethodError: undefined method `x=' for #<Point:0x2c29924 @y=4, @x=3>
```

Why does p.x = 10 fail?

Getters and setters, continued

If you want both getters and setters, use attr_accessor:

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

    attr_accessor :x, :y
end
```

Usage:

```
>> p = Point.new(3,4)            => #<Point:0x2c298d4 @y=4, @x=3>
>> p.x                           => 3
>> p.y = -20                     => -20
>> p                             => #<Point:0x2c298d4 @y=-20, @x=3>
```

It's important to appreciate that attr_reader and attr_accessor are *methods that create methods*. We could define a method called getters that has the same effect as attr_reader.

Operator overloading

<u>NOTE: This is a replacement set. Please discard the two-sheet handout that has Ruby slides 159-167.</u>

Operators as methods

It is possible to express most operators as method calls. Here are some examples:

```
>> 3.+(4)                 => 7

>> "abc".*(2)             => "abcabc"

>> "testing".[ ](2)       => 115

>> "testing".[ ](2,3)     => "sti"

>> 10.==20                => false
```

In general, *expr1 op expr2* can be written as *expr1.op expr2*

Unary operators require a little more syntax:

```
>> 5.-@()          => -5
```

Problem: What are some binary operations that can't be expressed as a method call in Ruby?

Operator overloading

In most languages at least a few operators are "overloaded"—an operator stands for more than one operation.

Examples:

C:	`+` is used to express addition of integers, floating point numbers, and pointer/integer pairs.
Java:	`+` is used to express addition and string concatenation.
Icon:	`*x` produces the number of... characters in a string values in a list key/value pairs in a table results a "co-expression" has produced and more...

What are examples of overloading in Ruby? In ML?

Operator overloading, continued

As a simple vehicle to study overloading in Ruby, imagine a dimensions-only rectangle:

```
class Rectangle
    def initialize(w,h); @width = w; @height =h; end
    def area; @width * @height; end
    attr_reader :width, :height

    def inspect
        "%g x %g Rectangle" % [@width, @height]
    end
end
```

Usage:

```
>> r = Rectangle.new(3,4)       => 3 x 4 Rectangle

>> r.area                       => 12

>> r.width                      => 3
```

Note that an inspect method is supplied to produce a more concise representation in irb.

Operator overloading, continued

Let's imagine that we can compute the "sum" of two rectangles:

```
>> a = Rectangle.new(3,4)     => 3 x 4 Rectangle

>> b = Rectangle.new(5,6)     => 5 x 6 Rectangle

>> a + b                      => 8 x 10 Rectangle

>> c = a + b + b              => 13 x 16 Rectangle

>> (a + b + c).area           => 546
```

As shown above, what does `Rectangle + Rectangle` mean?

Operator overloading, continued

Our vision:

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

Here's how to make it so:

```
class Rectangle
    def + rhs
        Rectangle.new(self.width + rhs.width, self.height + rhs.height)
    end
end
```

Remember that `a + b` is equivalent to `a.+(b)`. We are invoking the method "+" on `a` and passing it `b` as a parameter. The parameter name, `rhs`, stands for "right-hand side".

Instead of the above, would the following work? Would it be better in some cases?

```
Rectangle.new(@width + rhs.width, @height + rhs.height)
```

Operator overloading, continued

For reference:

```
      def + rhs
         Rectangle.new(self.width + rhs.width, self.height + rhs.height)
      end
```

Here is a faulty implementation, and usage of it:

```
    def + rhs
       @width += rhs.width; @height += rhs.height
    end

    >> a = Rectangle.new(3,4)      => 3 x 4 Rectangle

    >> b = Rectangle.new(5,6)      => 5 x 6 Rectangle

    >> a+b                         => 10

    >> a                           => 8 x 10 Rectangle
```

What's the problem?

Operator overloading, continued

Just like with regular methods, we have complete freedom to define what's meant by an expression using an overloaded operator. For example, a traditionally applicative operator can be turned into an imperative operation.

Here is a method for Rectangle that defines unary minus to be an imperative "rotation":

```
def -@     # Note: '-@' is used instead of just '-' to distinguish the unary form
   @width, @height = @height, @width         # "parallel assignment" to swap
   self
end

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

>> a                           => 2 x 5 Rectangle

>> -a                          => 5 x 2 Rectangle

>> a + -a                      => 4 x 10 Rectangle

>> a                           => 2 x 5 Rectangle
```

Any surprises?

Operator overloading, continued

Consider "scaling" a rectangle by some factor. Example:

```
>> a = Rectangle.new(3,4)      => 3 x 4 Rectangle

>> b = a * 5                   => 15 x 20 Rectangle

>> c = b * 0.77                => 11.55 x 15.4 Rectangle
```

Implementation:

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

A problem:

```
>> a                      => 3 x 4 Rectangle

>> 3 * a
TypeError: Rectangle can't be coerced into Fixnum
```

What's wrong?

Operator overloading, continued

Imagine a case where it is useful to reference width and height uniformly, via subscripts:

```
>> a = Rectangle.new(3,4)      => 3 x 4 Rectangle

>> a[0]                        => 3

>> a[1]                        => 4

>> a[2]                        ArgumentError: out of bounds
```

Remember that `a[0]` is `a.[ ](0)`.

Implementation:

```
def [ ] n
   case n
   when 0 then width
   when 1 then height
   else raise ArgumentError.new("out of bounds")   # Exception!
   end
end
```

Ruby is mutable

The ability to define meaning for operations like Rectangle + Rectangle leads us to say that Ruby is *extensible*.

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

For example, if we wanted to be sure that a program never used integer addition or negation, we could do this:

```
class Fixnum
    def + x
        raise "boom!"
    end
    def -@
        raise "boom!"
    end
end
```

In contrast, C++ is extensible, but not mutable. In C++, for example, you can define the meaning of Rectangle * int but you can't change the meaning of integer addition, as we do above.

Inheritance

Inheritance in Ruby

Java vs. Ruby

Modules and mixins

More to come on inheritance...

Inheritance in Ruby

A simple example of inheritance can be seen with clocks and alarm clocks. An alarm clock is a clock with a little bit more. Here are trivial models of them in Ruby:

```
class Clock
    def initialize time
        @time = time
    end
    attr_reader :time
end
```

```
class AlarmClock < Clock
    attr_accessor :alarm_time
    def initialize time
        super(time)
    end
    def on;  @on = true end
    def off;  @on = false end
end
```

The less-than symbol specifies that AlarmClock is a subclass of Clock.

Just like Java, a call to super is used to pass arguments to the superclass constructor.

Ruby supports only single inheritance but "mixins" provide a solution for most situations where multiple inheritance is useful. (More on mixins later.)

Inheritance, continued

Usage is not much of a surprise:

```
>> c = Clock.new("12:00")            => #<Clock:0x2c44198 @time="12:00">

>> c.time                            => "12:00"

>> ac = AlarmClock.new("12:00")      => #<AlarmClock:... @time="12:00">

>> ac.time                           => "12:00"

>> ac.alarm_time = "8:00"            => "8:00"

>> ac.on                             => true

>> ac
=> #<AlarmClock:0x2c30c38 @on=true, @time="12:00", @alarm_time="8:00">
```

Note that AlarmClock's @on and @alarm_time attributes do not appear until they are set.

To keep things simple, times are represented with strings.

Inheritance, continued

The method alarm_battery creates a "battery" of num_clocks AlarmClocks. The first is set for whenn. The others are set for intervals of interval minutes.

```
def alarm_battery(whenn, num_clocks, interval)
    battery = [ ]
    num_clocks.times {
        c = AlarmClock.new("now")                  # Imagine this works
        c.alarm_time = whenn
        whenn = add_time(whenn, interval)         # Imagine this method
        battery << c
        }
    battery
end
```

Usage:

```
>> battery = alarm_battery("8:00", 10, 5)     => Array with ten AlarmClocks

>> battery.size                               => 10
>> p battery[2]
#<AlarmClock:0x2c19d94 @alarm_time="8:10", @time="22:06">
```

Inheritance: Java vs. Ruby

Here is a Ruby method to activate all the alarms in a battery:

```
def activate_alarms(clocks)
    for ac in clocks
        ac.on
    end
end
```

Here is an analog in Java:

```
static void activate_alarms ( ArrayList<AlarmClock> clocks ) {
    for (AlarmClock ac: clocks)
        ac.on();
    }
```

Is there a practical difference between the two? Or are these just two ways to express the same computation?

Inheritance: Java vs. Ruby, continued

Another group has developed a WeatherClock. It is essentially an alarm clock but lets the user specify adjustments based on the weather. You might want be awakened earlier than normal if it is snowy, to provide time to shovel out the car. Here is WeatherClock, in Java:

```
class WeatherClock extends Clock {
    public void on() ...
    public void addCondition(WeatherCondition c, AlarmAdjustment a) ...
    ...
    }
```

We'd like to add a WeatherClock to our alarm battery,

```
ArrayList<AlarmClock> battery = alarm_battery(...)   // Java
WeatherClock wc = new WeatherClock(...);
battery.add(wc);      # line 31
```

but there is a problem:

```
clock.java:31: cannot find symbol 'method add(WeatherClock)'
location: class java.util.ArrayList<AlarmClock>
        battery.add(wc);
```

What's wrong? How can we fix it?

Inheritance: Java vs. Ruby, continued

At hand:

```
ArrayList<AlarmClock> battery = alarm_battery(...)
WeatherClock wc = new WeatherClock(...);
battery.add(wc);      # line 31

clock.java:31: cannot find symbol 'method add(WeatherClock)'
```

Recall WeatherClock:

```
class WeatherClock extends Clock { ... }
```

Because WeatherClock is a Clock not an AlarmClock, we can't put it in an ArrayList of AlarmClock.

How about relaxing the types? Can we use ArrayList battery instead of ArrayList<AlarmClock> battery? That is, just have battery hold Objects instead of AlarmClocks?

Inheritance: Java vs. Ruby, continued

Excerpts from an Object-based version:

```
ArrayList battery = alarm_battery(...);        # Holds instances of Object
WeatherClock wc = new WeatherClock(...);
battery.add(wc);                               # OK: WeatherClock is-a Object
activate_alarms(battery);
```

and...

```
static void activate_alarms(ArrayList clocks) {
    for (Object o: clocks) {
        AlarmClock ac = (AlarmClock)o;         # line 24
        ac.on();
        }
    }
```

Result:

```
Exception in thread "main" java.lang.ClassCastException: WeatherClock
        at clock2.activate_alarms(clock2.java:24)
```

Now what's wrong? Does this call for a design pattern?

Inheritance: Java vs. Ruby, continued

In Ruby, things are simpler:

```
class WeatherClock < Clock   # Just like the Java version, WeatherClock is-a Clock
    ...                      # not an AlarmClock.
    def on; ... end
end
```

and...

```
def activate_alarms(clocks)
    for ac in clocks
        ac.on
    end
end
```

and...

```
battery = alarm_battery("8:00", 10, 5)
battery << WeatherClock.new(...)
activate_alarms(battery)
```

It works! (How?!)

This is yet another set of replacements.

The following slides supersede slides 180-184 in the set of 159-184 that was distributed on October 19.

Inheritance: Java vs. Ruby, continued

At hand:

```
def activate_alarms(clocks)
    for ac in clocks
        ac.on
    end
end

battery = alarm_battery("8:00", 10, 5)

battery << WeatherClock.new("x")

activate_alarms(battery)
```

Do we really need `activate_alarms`?

Inheritance: Java vs. Ruby, continued

activate_alarms simply invokes the on method of each element in an array. Here's another way to do the same thing:

```
        battery.each {|c| c.on }
```

Things work nicely because although the misguided WeatherClock folks don't consider their invention to be an AlarmClock, they did happen to choose "on" as the name of the method to activate it.

What could we do if instead of naming it "on" they had named it "activate"?

Inheritance: Java vs. Ruby, continued

Option 1: Add an on method to WeatherClock:

```
class WeatherClock
    def on
        activate
    end
end
```

However, if they've frozen it with WeatherClock.freeze, we'll see this:

```
TypeError: can't modify frozen class
```

Now what?

Inheritance: Java vs. Ruby, continued

Option 2: Borrow a little bit from the ADAPTER design pattern:

```
class AdaptedWeatherClock < WeatherClock
    def on
        activate
    end
end

battery = alarm_battery("8:00", 10, 5)

battery << AdaptedWeatherClock.new("x")

battery.each {|c| c.on }
```

Fact: Much of the complexity in some OO design patterns rises from accommodation of compile-time type checking.

interface in Java

Question:

What capability is provided by Java's interface construct?

One answer:

An interface specifies a set of operations that must be provided by every class that implements the interface. In turn, it can be determined at compile time whether a class claiming to implement the interface provides all the required methods.

For example, if a Java class wants to implement Iterable, it must provide a method called iterator. iterator must return an instance of a class that implements Iterator that in turn must provide hasNext, next, and remove methods.

Challenge: Keeping track of the time you spend, create a Java class X such that this code,

```
X x = new X();
for (Object o: x)
   System.out.println(o);
```

produces this output: x

Sidebar: const methods and in C++

In C++ a method can be declared as const, which indicates that the method won't change the value of any instance variables. For example, getting the length of a list shouldn't cause a change in the list, right? In C++ we'd say this:

```
class IntList {
    public:
        void addValue(int value) { ... }
        int get() { ...removes the first element and returns it...}
        int getLength() const { ... }
        ...
    };
```

const can be applied to a parameters to indicate that the parameter should not be modified:

```
void f(const IntList& ilist)        // w/o const, this is like f(IntList ilist) in Java
{
    int len = ilist.getLength();    // OK

    ilist.addValue(7);              // compilation error — ilist is const
}
```

Question: Is there a situation where getLength() might want to change a value?

Sidebar, continued

Here is C++ code inspired by actual events at one of Tucson's largest software companies:

```
void printInts(const IntList& ilist)
{
    for (int i = 0; i < ilist.getLength(); i++) {
          cout << ilist.get() << endl;
        }
}
```

Judicious use of const can catch bugs at compile time. Is the complexity it adds worth the benefit?

It is common for self-taught adopters of C++ to not see the value of const at first. When they do see the light, they then have a "const Day" to convert all the code over to using const. (As a rule you need to use const everywhere or nowhere; a mix usually doesn't work.)

A Big Question

Java's compile-time type checking can ensure that we won't have a run-time error because we forgot a method. Ruby doesn't provide that assurance. Example:

```
% cat iter3.rb
class X; end

for o in X.new          # Recall that 'for' expects the "in" value to respond to 'each'
   puts o
end

% ruby iter3.rb
iter3.rb:3: undefined method `each' for #<X:0x2838618> (NoMethodError)
```

The fix:

```
class X
   def each
      yield "x"
   end
end
```

A Big Question, continued

Big Question:

Does the benefit of detecting missing methods at compile time outweigh the extra code and complexity required to allow that compile-time detection?

When would a forgotten each method most likely turn up in a Ruby application? In development? In testing? In production?

With a non-trivial iterator in mind, which is harder: remembering to write each or getting the iteration correct? Would there be tests focused on the operation of the iterator?

What does Test Driven Development add to the question?

You might think of statically-typed languages as offering a deal: "If you'll follow these rules I'll be able to detect certain types of errors when the program is compiled."

How does the execution model of a language change this question? For example, is compile-time type-checking as important in Java or Ruby as it is in C or C++?

Consider this: Before generics were added to Java, lots of successful systems were developed using Hashtable and Vector with extensive use of downcasts, like x = (X)v.get(i). Lots of Java shops are still not using generics.

JARWARS

EPISODE III

REVENGE OF THE <T>

http://www.madbean.com/anim/jarwars

Modules

A Ruby *module* can be used to group related methods for organizational purposes.

Imagine a collection of methods to comfort a homesick ML programmer at Camp Ruby:

```
module ML
    def ML.hd a
        a[0]
    end
    def ML.drop a, n
        a[n..-1]
    end
    ...more...
end

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

>> ML.hd(a)                            => 10

>> ML.drop(a, 2)                       => [30, 40.0]

>> ML.tl(ML.tl(ML.tl(a)))              => [40.0]
```

Modules as "mixins"

In addition to providing a way to group related methods, a module can be "included" in a class. When a module is used in this way it is called a "mixin" because it mixes additional functionality into a class.

Here is a revised version of the ML module:

```
module ML
   def hd; self[0]; end

   def tl; self[1..-1]; end

   def drop n; self[n..-1]; end

   def take n; self[0,n]; end
end
```

<u>Note that these methods have one less parameter, operating on self instead of the parameter a</u>. For comparison, here's the first version of tl:

```
def ML.tl a
   a[1..-1]
end
```

Mixins, continued

We can mix our ML methods into the Array class like this:

```
require 'ML'            # loads ML.rb if not already loaded
class Array
   include ML
end
```

<u>After loading the above code, we can use those ML methods on arrays:</u>

```
>> ints = (1..10).to_a       => [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

>> ints.hd                   => 1

>> ints.tl                   => [2, 3, 4, 5, 6, 7, 8, 9, 10]

>> ints.drop 3               => [4, 5, 6, 7, 8, 9, 10]
```

How could we add these same capabilities to the String class?

Mixins, continued

An include is all we need to add the same capabilities to String:

```
require 'ML'
class String
   include ML
end

>> s = "testing"       => "testing"

>> s.tl                => "esting"

>> s.hd                => 116           # How could we get "t" instead?

>> s.drop 5            => "ng"
```

Does Java have any sort of mixin capability? What would be required to produce a comparable effect?

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

Modules and superclasses

The Ruby library makes extensive use of mixins.

The class method ancestors can be used to see the superclasses and modules that contribute methods to a class:

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

>> Fixnum.ancestors        => [Fixnum, Integer, Precision, Numeric, Comparable,
                                  Object, Kernel]
```

The method included_modules shows the modules that a class includes.
instance_methods can be used to see what methods are in a module:

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

>> Enumerable.instance_methods
=> ["collect", "detect", "max", "sort", "partition", "any?", "reject", "zip", "find", "min",
"member?", "entries", "inject", "all?", "select", "each_with_index", "grep", "to_a",
"map", "include?", "find_all", "sort_by"]

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

Modules and superclasses, continued

All classes include Kernel. If no superclass is specified, a class subclasses Object.

Example:

```
>> class X; end             => nil

>> X.ancestors              => [X, Object, Kernel]

>> X.included_modules   => [Kernel]

>> X.superclass             => Object
```

The Enumerable module

Some Ruby modules provide a collection of methods implemented in terms of a single operation. One such module is Enumerable:

```
>> Enumerable.instance_methods.sort
=> ["all?", "any?", "collect", "detect", "each_with_index", "entries", "find", "find_all",
"grep", "include?", "inject",  "map", "max", "member?", "min", "partition", "reject",
"select", "sort", "sort_by", "to_a", "zip"]
```

All of the methods of Enumerable are written in terms of a single method, each, which is an iterator.

If class implements each and includes Enumerable then all those 22 methods become available to instances of the class.

Enumerable, continued

Here's a class whose instances simply hold three values:

```
class Trio
    include Enumerable
    def initialize(a,b,c); @values = [a,b,c]; end
    def each
        @values.each {|v| yield v }
    end
end
```

Because Trio provides each and includes Enumerable, we can do a lot with it:

```
>> t = Trio.new(10,"twenty",30)       => #<Trio:0x28e9958 @values=[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"]]
```

The Comparable module

Another common mixin is Comparable. These methods,

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

are implemented in terms of <=>.

Let's compare rectangles on the basis of areas:

```
class Rectangle
    include Comparable
    def <=> rhs
        diff = self.area - rhs.area
        case
        when diff < 0 then -1
        when diff > 0 then 1
        else 0
        end
    end
end
```

Comparable, continued

Usage:

```
>> r1 = Rectangle.new(3,4)     => 3 x 4 Rectangle

>> r2 = Rectangle.new(5,2)     => 5 x 2 Rectangle

>> r3 = Rectangle.new(2,2)     => 2 x 2 Rectangle

>> r1 < r2                     => false

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

>> [r1,r2,r3].min              => 2 x 2 Rectangle

>> r2.between?(r1,r3)          => false

>> r2.between?(r3,r1)          => true
```

Graphics with Tk

Tk Basics

Hello in Tk

Component callbacks

Updating a component

Sidebar: Named parameters

Drawing on a canvas

Mouse events

Grand finale: Pulsing circles

Tk basics

Tk is a library of components for building GUIs. It provides a typical collection of elements including buttons, labels, lists, scrollbars, text fields, and more.

Tk was created by John Outsterout at UCB for use with Tcl ("tickle"), a scripting language.

Many languages, including Ruby, have a library for working with Tk. Aside from Tcl, Tk is perhaps most often used with Perl.

There are other Ruby GUI toolkits but Tk is available on the widest range of platforms. (?)

The Ruby-specific documentation on Tk is pretty skimpy. (Take a look at the Tk section in http://ruby-doc.org/stdlib!) Tk tutorials generally refer readers to the Perl Tk documentation.

Disclaimer: The instructor only has spent a few hours experimenting with Tk. In a few minutes you'll know at least as much as he does about Tk.

Hello in Tk

Here is a nearly minimal Tk program that puts up a window with a label that says "Hello!"

```
# tkhello.rb
require 'tk'                    # Later examples don't show this line but it is still needed.
root = TkRoot.new

TkLabel.new(root) {
    text "Hello!"
    font "Courier 36 bold italic"
    pack
    }

Tk.mainloop
```

Notes:

TkRoot.new creates a new top-level frame.

TkLabel.new creates a label and puts it in the root. It is configured via method calls in the associated block.

pack invokes a geometry manager. Without this call, only an empty frame appears.

Component callbacks

If a button is configured with a command, the associated block is executed when the button is pressed.

Here's a program that creates a button for each command line argument and prints as buttons are clicked. The button text size is proportional to the length of the label.

```
class Button                    # tkclick1.rb
    def initialize(label)
        TkButton.new{
            text label; font "Jokewood #{label.size*10}";
            command { puts "Got a click on '#{label}'" }
            pack
            }
    end
end
ARGV.each { |label| Button.new(label) }
Tk.mainloop
```

This example takes advantage of the fact that we really don't need to specify the root.

Updating a component

The following code creates a label that is updated with a count when a button is pressed:

```
def fmt_clicks n; "Clicks: %d" % n; end          # tkclick2.rb

label = TkLabel.new {
   text fmt_clicks(0); font "HelterSkelter 36"; pack }

TkButton.new {
   text "Push Here";
   font "Jokewood 36";
   clicks = 0
   command { clicks += 1
                 label.configure(:text => (fmt_clicks clicks))
           }
   pack
   }
Tk.mainloop
```

A call to `label.configure` is used to update the text. The syntax '`:text => ...`' creates and passes a hash as the argument. (A sidebar on this follows.)

Sidebar: Named parameters

Some languages support *named parameters*—instead of parameters being specified by position, they are specified by name and are position-independent. Here is a generic example of equivalent calls using named parameters:

```
f(x = 3, y = 4)
f(y = 4, x = 3)
```

Ruby does not provide named parameters but does provide an interesting facility with a hash:

```
def f(a, b, h)
   p [a, b, h]
end

f(10, 20, :x => 10, :y => 2, "x"*5 => [ ])
```

The result:

```
[10, 20, {:x=>10, :y=>2, "xxxxx"=>[ ]}]
```

The hash specification must be the last part of the argument list. It is common to see Symbols used for the keys.

Drawing on a Canvas

The TkCanvas class provides a drawing surface.

```
width = 400              # tkcanvas1.rb
height = 300

canvas = TkCanvas.new { width width+10; height height+10 }
canvas.pack

args = [canvas, 5, 5, width, height]
TkcOval.new *args
TkcLine.new *args
TkcRectangle.new *args

Tk.mainloop
```

Objects are added to the canvas by creating instances of classes like TkcOval (canvas oval).

Mouse events

The bind method of Canvas specifies a "Proc" to be called whenever a particular event is recognized.

```
def do_press(x, y)           # tkmouse.rb
    printf("Button down at %d, %d\n", x, y)
end

def do_motion(x, y); printf("Motion at %d, %d\n", x, y); end

def do_release(x, y); printf("Button up at %d, %d\n", x, y); end

canvas = TkCanvas.new; canvas.pack

canvas.bind("1", lambda {|e| do_press(e.x, e.y)})
canvas.bind("B1-Motion", lambda {|x, y| do_motion(x, y)}, "%x %y")
canvas.bind("ButtonRelease-1", lambda {|x, y| do_release(x, y)}, "%x %y")

Tk.mainloop
```

Kernel#lamba creates a Proc from the associated block. Think of a Proc like an anonymous function in ML.

Grand finale: Pulsing Circles

```
class Circle
    SZ = 200
    def initialize(canvas, x, y)
        @canvas = canvas; @inc = 1; @ux = x - SZ/2; @uy = y - SZ/2
        @lx = @ux + SZ; @ly = @uy + SZ
        @oval = TkcOval.new(@canvas, @ux, @uy, @lx, @ly)
        @canvas.after(1) { tick }
    end
    def tick
        @inc *= -1 if @ux >= @lx or (@ux - @lx).abs > SZ
        @ux += @inc; @uy += @inc; @lx -= @inc; @ly -= @inc
        @oval.coords(@ux, @uy, @lx, @ly)
        @canvas.after(1) { tick }
    end
end

$canvas = TkCanvas.new { width 400; height 300; pack }
$canvas.bind("1", lambda {|e| do_press(e.x, e.y)})

def do_press(x, y);  Circle.new($canvas, x, y); end
Tk.mainloop()
```

Miscellaneous

JRuby

My first practical Ruby program

JRuby

"JRuby is an 100% pure-Java implementation of the Ruby programming language."
—Home page at jruby.codehaus.org

Here's a bash script, jruby, that runs it:

```
java -jar c:/dnload/jruby-0.9.0/lib/jruby.jar $*
```

Usage:

```
% time jruby mtimes.rb mtimes.1 the
user    0m0.061s
...

% time ruby mtimes.rb mtimes.1 the
user    0m0.015s
...
```

So what's the big deal?

JRuby, continued

```
require 'java'   # This is swing2.rb from the JRuby samples.
include_class "java.awt.event.ActionListener"
include_class ["JButton", "JFrame", "JLabel", "JOptionPane"].
            map {|e| "javax.swing." + e}

frame = JFrame.new("Hello Swing")
button = JButton.new("Klick Me!")

class ClickAction < ActionListener
        def actionPerformed(evt)
            JOptionPane.showMessageDialog(nil,
                "<html>Hello from <b><u>JRuby</u></b>.<br>" +
                "Button '#{evt.getActionCommand()}' clicked.")
        end
end
button.addActionListener(ClickAction.new)

frame.getContentPane().add(button) # Add the button to the frame

frame.setDefaultCloseOperation(JFrame::EXIT_ON_CLOSE) # Show frame
frame.pack(); frame.setVisible(true)
```

My first practical Ruby program

September 3, 2006:

```
n = 1
d = Date.new(2006, 8, 22)
incs = [2,5]
pos = 0
while d < Date.new(2006, 12, 6)
    if d != Date.new(2006, 11, 23)
        printf("%s %s, #%2d\n",
            if d.cwday() == 2: "T"; else "H";end, d.strftime("%m/%d/%y"), n)
        n += 1
    end
    d += incs[pos % 2]
    pos += 1
end
```

Output:

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