A Look at Ruby

TCS Developer's SIG
February 6, 2007

William Mitchell (whm)
Mitchell Software Engineering (.com)

Introduction

What is Ruby?

Running Ruby

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 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 1.8.5 is the stable, recommended version. Version 1.9 is available.
- There is no written standard for Ruby. The language is effectively defined by MRI—Matz' Ruby Implementation.

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)

Running Ruby

One way to execute Ruby code is with irb, the Interactive Ruby Shell.

irb evaluates expressions as are they typed.

```
% irb
>> 1 + 2
=> 3

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

>> it.upcase
=> "TESTING123"

>> it.class
=> String

>> Math.cos(Math::PI / 3)
=> 0.5
```

My ~/.irbrc defines it as the previous result.

Running Ruby, continued

Source code in a file can be executed with the ruby command.

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 evaluated when encountered. (Later we'll see how to enclose code in a method.)

Existing Ruby implementations have no notion of compilation to a binary. There are no "executables", intermediate code files, etc.

Everything is an object

In Ruby, *every* value is an object.

Methods are invoked using *value.method(parameters...)*.

```
>> "testing".index("i")
=> 4

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

Parentheses can be omitted from an argument list:

```
>> "testing".gsub /[aeiou]/, "-"
=> "t-st-ng"
```

If a method requires no parameters 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

>> (30-40).abs      => 10

>> Math.exp(1).to_s
=> "2.71828182845905"

>> 17**50
=> 33300140732146818380750772381422989832214186835186851059977249

>> it.class         => Bignum
```

Variables have no type

In languages like C and Java, variables are declared to have a type. The compiler ensures that all operations are valid for 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.

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 C# 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 does not attempt to verify any of the above when the code is compiled.

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] Here's one discussion: 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?"

Ruby's philosophy is often "Why not?" Here are some examples, involving overloaded operators:

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

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

>> [1, 3, 15, 1, 2, 1, 3, 7] & [4, 3, 2, 1]
=> [1, 3, 2]

>> [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
- 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.

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 number of escapes are available.

```
>> "newline: \012, escape: \x1b, return: \cm"
=> "newline: \n, escape: \e, return: \r"
```

In a string literal using apostrophes only \' and \\ are recognized as escapes:

```
>> '\n\'\t'.length       Five characters: backslash, n, apostrophe, backslash, t
=> 5
```

An even more literal form is provided by %q{ ... }

```
>> %q{ just testin' this! ~@$%^&*()"[ ]<>,.}
=> " just testin' this! ~@$%^&*()\"[ ]<>,."
```

There's a fourth way, too, similar to "here documents" in UNIX shells.

How many ways to do something is too many?

Strings, continued

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

```
>> "abc".public_methods.sort
=> ["%", "*", "+", "<", "<<", "<=", "<=>", "==", "===", "=~", ">", ">=", "[ ]", "[ ]=",
"__id__", "__send__", "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, C#, 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
=> "testing"

>> x.upcase!
=> "TESTING"

>> y
=> "TESTING"
```

x and y now reference the same instance of String.

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

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:

```
>> "apple" == "ap" + "ple"        => true

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

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

There is also a "general" 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 fetched 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          The ASCII code for 'a'

>> s[1]             => 98

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

>> s[100]           => nil
```

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"      Start at 2, length of 3.
>> s[2,1]            => "p"        Remember that s[n] yields a number, not a string.
>> s[2..-1]          => "place"    2..-1 creates a Range object.
>> 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                      => "planaria"
```

If a string is the subscript it specifies the first occurrence of that string.

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

>> s                      => "plankton"
```

Speculate: What does `s["xyz"] = "abc"` produce?

Strings, continued

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

```
>> s = "2 + 2 = #{2 + 2}"        => "2 + 2 = 4"

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

The << operator appends to a string (imperatively!) and produces the new string.

```
>> s = "just"                    => "just"

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

Speculate: What's the result of "a" << "b"?

Numbers

On most machines, integers in the range -2^{30} to $2^{30}-1$ are represented by instances of Fixnum. Integers outside that range are represented with a Bignum.

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

>> x.class            => Fixnum

>> x += 1             => 1073741824

>> x.class            => Bignum

>> x -= 1             => 1073741823

>> x.class            => Fixnum
```

Unlike many scripting languages, Ruby does not automatically convert between strings and numbers when needed.

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

Numbers, continued

Instances of Float represent 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
```

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

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

Arrays

A Ruby Array is an ordered sequence of values.[1]

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, 10000000000000000000000000000000000000]

>> a3 = [a1, a2, [[a1]]]
=> [[10, 20, 30], ["ten", 20, 30.0, 10000000000000000000000000000000000000],
[[[10, 20, 30]]]]
```

Arrays can hold values of any type. Cyclic structures are permitted.

[1] The speaker often makes the mistake of saying "list" instead of "array".

Arrays, continued

Array elements and subarrays (sometimes called *slices*) are specified with the same notation that is used for string subscripting.

```
>> a = [1, "two", 3.0, %w{a b c d}]      %w{ ... } creates an array of strings.
=> [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
```

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 number of methods are availabe for arrays:

```
>> [ ].methods.sort
=> ["&", "*", "+", "-", "<<", "<=>", "==", "===", "=~", "[]", "[]=", "__id__", "__send__",
"all?", "any?", "assoc", "at" , "class", "clear", "clone", "collect", "collect!", "compact",
"compact!", "concat", "delete", "delete_at", "delete_if",  "detect", "display", "dup",
"each", "each_index", "each_with_index", "empty?", "entries", "eql?", "equal?",
"extend", " fetch", "fill", "find", "find_all", "first", "flatten", "flatten!", "freeze", "frozen?",
"gem", "grep", "hash", "id", "include?", "index", "indexes", "indices", "inject", "insert",
"inspect", "instance_eval", "instance_of?", "instance_variable_get",
"instance_variable_set", "instance_variables", "is_a?", "join", "kind_of?", "last",
"length", "map", "map!", "max", "member?", "method", "methods", "min", "nil?",
"nitems", "object_id", "oid", "pack", "partition", "pop", "private_methods",
"protected_methods", "public_methods", "push", "rassoc", "reject", "reject!", "replace",
"require", "require_gem", "respond_to?", "reverse", "reverse!", "reverse_each",
"rindex", "select", "send", "shift", "singleton_methods", "size", "slice", "slice!", "sort",
"sort!", "sort_by", "taint", "tainted?", "to_a", "to_ary", "to_s", "transpose", "type","uniq",
"uniq!", "unshift", "untaint", "values_at", "zip", "|"]

>> it.length
=> 122
```

Control structures

The while loop

Logical operators

if-then-else

if and unless as modifiers

The for loop

The while loop

Here is a loop that prints the numbers from 1 through 10:

```
i = 1
while i <= 10 do            "do" is optional
   puts i
   i += 1
end                         "end" is required, even if only one statement in body
```

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

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 any non-zero integer value to be "true".

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

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

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

Problem: Given that gets returns nil at end of file, explain how the loop works.

while, continued

The string returned by gets has a trailing newline. String's chomp method can be used to remove it.

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

```
result = ""

while line = gets.chomp
   result += line
end

puts result
```

Why doesn't it work?

Problem: Write a while loop that prints the characters in the string s, one per line. Don't use the length method of String.[1]

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

Logical operators

Conjunction in Ruby is &&, just like the C family, but the semantics are different:

```
>> true && false        => false
>> true && "abc"        => "abc"
>> true && false        => false
>> false && nil         => false
```

The disjunction operator is two "or bars":

```
>> false || 2           => 2
>> "abc" || "xyz"       => "abc"
>> s[0] || s[3]         => 97
>> s[4] || false        => false
```

Challenge: Describe the rule that governs the result of conjunction and disjunction.

Logical operators, continued

Ruby has compound (augmented) assignment, just like the C family. With that in mind, what is the meaning of the following expression?

```
x ||= 20
```

The if-then-else expression

Ruby's if-then-else is an expression, not a statement:

```
>> 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"
```

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

Ruby also provides x > y ? 1 : 2 (a ternary conditional operator) just like the C family. Is that a good thing or bad thing?

if and unless as modifiers

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

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

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

>> total, count = 123.4, 0

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

The general forms are:

expression1 if *expression2*
expression1 unless *expression2*

Question: What does 'x.f if x' mean?

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.

Other flow control mechanisms

Ruby also has:

An elsif clause

break, next, redo, retry

A case expression that has two forms

Freestanding Methods

Basics

Where's the class?

Duck typing

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
```

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 C family analog of double the domain and range are the integers.

What is the domain and range of double in Ruby?

Duck typing

For reference:

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

In computer science literature a routine such as double is said to be *polymorphic*—it can operate on data of more than one form.

In the Ruby community it is said that double uses "duck typing".

The term "duck typing" comes from the "duck test":

If it walks like a duck and quacks like a duck, it must be a duck.

What's the "duck test" for x in the routine above?

Duck typing, continued

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

```
>> 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]]
```

What's the duck test for polysum?

Write polysum!

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.

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
```

Sidebar: Iterate with each or use a for loop?

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

The iterator Array#each is commonly used to create side effects of interest, like printing values or changing variables. In contrast, the "work" of some iterators is to produce a value.

```
>> [10, "twenty", 30].map { |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] }     Like C's qsort...
=> ["shake", "burger", "fries"]

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

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

The computation performed by inject is known in functional programming literature as "folding".

Challenge: Perform mapping and selection using inject.

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 every line in the file x.txt, we might do this:

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

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
```

Scoping issues with blocks:

- If a variable is created in a block, the scope of the variable is limited to the block.
- If a variable already exists, a reference to it in a block is resolved to the existing instance.
- It's said that this behavior may change with Ruby 2.0.

Creating iterators with yield

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

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

Here is a simple, chatty 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"
```

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

To some extent, a block can be thought of as an anonymous function; yield can be thought of as a call to that function.

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 is a function-like implementation of select:

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

Usage:

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

Class definition

Counter: A tally counter

An interesting thing about instance variables

Addition of methods

An interesting thing about class definitions

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 name initialize identifies the method as the constructor.

```
c1 = Counter.new

c2 = Counter.new "c2"
```

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

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.

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>
```

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
```

Common error: Omitting 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 @label="ctr 1", @count=0>

>> 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!

```
class Fixnum
   def rand
         raise ArgumentError if self < 1
         Kernel.rand(self)+1
   end
end

class String
   def rand
         raise ArgumentError if size == 0
         self[self.size.rand-1,1]
      end
end
```

Usage:

```
>> (1..10).collect { 5.rand }          => [3, 1, 3, 2, 1, 2, 2, 5, 2, 4]

>> (1..20).collect { "ATCG".rand }.to_s    => "CAGACAATGCTCCATCACAG"
```

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.

pickms.rb

```
class X
   print "What methods would you like? "
   gets.split.each { |m|
      case m
      when "f" then def f; "from f" end
      when "g" then def g; "from g" end
      when "h" then def h; "from h" end
      end
      }
end
```

Execution:

```
What methods would you like? f g
>> c = X.new          => #<X:0x2c2b224>
>> c.f                => "from f"
>> c.h
NoMethodError: undefined method `h' for #<X:0x2c2b224>
```

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 easily 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?

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            # 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?

Operator overloading

Operators as methods

Overloading in other languages

Overloading in Ruby

Mutability, and monkeying with math

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
```

Operator overloading in other languages

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 numeric addition and string concatenation.

Icon: `*x` produces the number of...

- characters in a string
- values in a list
- key/value pairs in a table
- results a "co-expression" has produced
- and more...

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                                          irb uses inspect to print results
      "%g x %g Rectangle" % [@width, @height]
   end
end
```

Usage:

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

>> r.area                      => 12

>> r.width                     => 3
```

Operator overloading, continued

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

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

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

>> a + b                       => 8 x 10 Rectangle

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

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

As shown above, what does Rectangle + Rectangle mean?

Operator overloading, continued

Our vision:

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

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
```

Recall 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")        Raises an exception
   end
end
```

Mutability, and monkeying with math

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 standard operations.

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

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:

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

```ruby
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 @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">
```

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              # Get the "head" (first element) of array a
      a[0]
   end
   def ML.drop a, n         # Return a copy of a with the first n elements removed
      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
```

Note that these methods have one less parameter, operating on self instead of the parameter a. 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:

```
class Array
   include ML
end
```

After loading the above code, we can use those ML methods on arrays:

```
>> 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]
```

This is another example of duck typing. What's the duck test here?

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:

```
class String
   include ML
end

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

>> s.tl                => "esting"

>> s.hd                => 116

>> s.drop 5            => "ng"
```

Could we do something like this in C# or Java?

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

Mixins, continued

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.

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

>> Fixnum.included_modules         => [Precision, Comparable, Kernel]
```

The Enumerable module

Here are the methods in 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 these methods 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.

In other words, if the object has an each method, the object is a duck!

The Enumerable module, continued

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"
```

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 includes Enumerable, and provides each, we can do a lot with it:

```
>> t = Trio.new(10,"twenty",30)      => #<Trio @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
```

Odds and Ends

Word tallying

Time totaling

A JRuby program

Graphics with Tk

What we didn't cover

Learning more about Ruby

Simple application: Word tallying

Imagine a program that tallies occurrences of words found on standard input:

```
% ruby tally.rb
to be or not to be
is not to be discussed
^Z
Word            Count
to                  3
be                  3
not                 2
or                  1
discussed           1
is                  1
```

This is a natural for implementation with Ruby's Hash class, which is a classic data structure known by many names, including associative array, dictionary, map, and table.

A hash holds a collection of key/value pairs. In principle any object whatsoever may be a key but Ruby has difficulties in some unusual cases. For example, using a cyclic array as a key causes a stack overflow.

Word tallying, continued

```
counts = Hash.new(0)

while line = gets
   for word in line.split
      counts[word] += 1
   end
end

pairs = counts.sort { |a,b| b[1] <=> a[1] }

for k,v in [["Word","Count"]] + pairs
   printf("%-10s\t%5s\n", k, v)
end
```

Produce zero for counts[key] *if* key *is not found.*

Would it be better to use each *with a block for these loops?*

For example, counts["be"] += 1

A note about the sort:

Without the block, counts.sort would return an array like this: [[*k1*,*v1*], [*k2*, *v2*], ...] with the pairs ordered by their respective keys, in ascending order.

An invocation of the block might have a = ["to", 3] and b = ["not", 2]. The comparison produces a result that effects a sorted order by descending count.

Simple 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
```

Times in an unexpected format are ignored:

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

Time totaling, continued

A solution using regular expressions:

```
def main
   mins = 0
   while line = gets do
      mins += parse_time(line.chomp)
   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
```

An example of JRuby

```
require 'java'                                          # 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)
```

Ruby graphics with Tk

tkpulse.rb

```
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)
      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(10) { tick }
   end
end

$canvas = TkCanvas.new { width 700; height 500; 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()
```

What we didn't cover

It's possible to make good use of Ruby with only minimal knowledge of it but it's a big language overall. Here are some of the things that were barely mentioned, or not mentioned at all:

- Hashes
- Regular expressions
- The Proc and Kernel classes
- IDEs and debugging tools
- Reflection and metaprogramming
- Threads
- Exceptions
- Tainted data
- Hooks
- RDoc
- Extending Ruby with C
- Libraries for lots of interesting things
- The rake build tool

Learn more about Ruby

- The Ruby home page is ruby-lang.org.

- *Programming Ruby—The Pragmatic Programmers' Guide, 2nd edition,* by Dave Thomas with Chad Fowler and Andy Hunt, also known as the "pickaxe book", is widely recognized as being the best book on Ruby at present.

  The first edition is available for free: www.ruby-doc.org/docs/ProgrammingRuby

- *Ruby Cookbook*, by Lucas Carlson and Leonard Richardson, is packed full of small but practical examples of using Ruby in a wide variety of settings.

- *Agile Web Development with Rails, 2nd edition*, by Dave Thomas et al. is commonly recommended if you're interested in learning about Ruby on Rails. It assumes knowledge of Ruby.

- The ruby-lang channel on irc.freenode.net is pretty good for live Q&A.

- Mitchell Software Engineering offers Ruby training tailored to suit your needs.