UCSC Extension Silicon Valley

University of California Santa Cruz Extension
21341 Ruby, Introduction

Instructor: Wayne Vucenic, B.S.

Course Description:
Ruby is a dynamically-typed, object-oriented programming language, which has recently experienced a surge in interest because the popular Rails web programming framework is written in Ruby. Ruby is a simple, elegant language that allows programs to be expressed clearly and concisely. A Ruby program often takes only half as many lines of code as the equivalent program in Java or C#. This contributes significantly to programmer productivity.

Ruby is open-source, and runs on Mac OS X, Windows, Unix and Linux. It is general purpose, and can be used for command scripts, system administration, text processing, GUI programs, networked and distributed applications, and, of course, web development. Ruby works well with Test-Driven Development and Agile Methodologies.

This course is an introduction to Ruby, and will provide a solid foundation for further study. Programming with a dynamic language is different, and we'll try to jump start your learning by emphasizing material not easily found elsewhere.

Skills Needed: Some familiarity with at least one object-oriented programming language (Java, C#, C++, Smalltalk, CLOS, etc.)

Course Objectives:
At the conclusion of the course, participants should be able to:

- Describe the major features of the Ruby programming language
- Discuss whether Ruby is an appropriate language for a given project
- Explain the differences between Static and Dynamic typing

Performance Evaluation:
Take home test requiring the student to write several short pieces of Ruby code.

Recommended Texts:
Programming Ruby, 2nd edition, Dave Thomas, et al., Pragmatic Bookshelf, ISBN-10: 0974514055, ISBN-13: 978-0974514055.

Programming Ruby 1.9, 3rd edition, Dave Thomas, et al., Pragmatic Bookshelf, ISBN-10: 1934356085, ISBN-13: 978-1934356081.

The Well-Grounded Rubyist, 1st edition, David A. Black, Manning Publications, ISBN-10: 1933988657, ISBN-13: 978-1933988658.

Course Outline:

Class 1:

Overview of Ruby, including Ruby's design principles
Using IRB (interactive Ruby)
Numbers
Strings
Calling a method on an object, and bang(!) methods
Ranges
Arrays and Hashes
Variables and Constants
Fahrenheit to Centigrade sample program, defining a method
Conditional Execution and Statement Modifiers
Case Expressions
Loops
Detailed look at Code Blocks
Static vs. Dynamic Typing
OOP Extended Example - Course Enrollment
Instance Variables
Exceptions and Exception Handling
Invariants
Symbols
Quicksort in Ruby

Introduction to Ruby

By Wayne Vucenic

waynev@gmail.com

Revised August 2012

Agenda

- Introduction to Ruby. But not everything.
- Goal: Provide a solid foundation for further learning. Material that's not in the books.
- Tools, tricks, tips – jumpstart your learning of Ruby. Programming with a dynamic language is different.
- Interactivity: demos, hands-on exercises. Ruby supports a different teaching style.
- (Agile Methodologies)

Thanks To

Some of these slides were adapted from the presentation "Ruby for Perl Programmers" given by Hal Fulton on 4/17/2002.

"Use freely but acknowledge the source"

Other material derived from:

Ruby for Rails, David A. Black (book)
Programming Ruby 2nd ed., Dave Thomas, Chad Fowler and Andy Hunt (book)
Object-Oriented Scripting in Ruby, Yukihiro Matsumoto (presentation)

A Brief History of Ruby

- 1993 Conceived by Yukihiro Matsumoto ("Matz")
- 1995 First release, Version 0.95 (Japan)
- 2000 *Programming Ruby* published (October); comp.lang.ruby created
- 2001 Version 1.6 released
- 2003 Version 1.8 released
- 2004 (July) Rails released. Ruby's "Killer app"
- 2007 (Dec 25th) Version 1.9.0 released
- 2009 (Jan 30th) Version 1.9.1 released
- 2010 (Aug 18th) Version 1.9.2 released
- 2011 (Oct 30th) Version 1.9.3 released

Programming Language Design

"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."
- Yukihiro Matsumoto

Programming Language Design

"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."
- Yukihiro Matsumoto

"C++ is … designed to make programming more enjoyable for the serious programmer" – Bjarne Stroustrup, Preface to *The C++ Programming Language*

Ruby's Design Principles

- Human-oriented design. Computer time is plentiful, human time is scarce.
- Principle of Least Surprise (POLS), fundamentally means Matz' own surprise.
- "Naturalness", consistency
- Flexibility and dynamism

Where does Ruby get its ideas?

- Smalltalk (dynamic features, OOP) "Ruby is Smalltalk minus the unfamiliar syntax plus..." – Matz
- LISP (mixins, dynamic features)
- Perl (scripting power, Perl5 compatible regular expressions, some syntax, $ variables)
- Python (exceptions, etc.)
- CLU (iterators, `yield`)
- Scheme (closures, methods end in ? or !)
- C (operators (+=, ?:), printf/sprintf, etc.)
- C++ (<< for output)
- Fortran (2**10)

Why do I *personally* like Ruby?

- Almost 15 years fulltime C++ programmer. I *really* like C++, in part because it's a "power tool".
- Ruby is similarly a power tool. Redefine +.
- Highly dynamic and dynamically typed
- Ruby community and new ideas

Sample Code

```
puts "Hello World"   # this is a complete program
Hello World

puts "Hello".length
5

print Time.now
Sun Jul 05 13:50:34 -0700 2009

A comment begins with # and goes until the end of line
# this is a comment
a = 1   # so is this
```

IRB 1 – Interactive Ruby

Type "irb" or "irb --prompt simple" at a command prompt.

```
C:\ruby>irb --prompt simple
>> 2+3
=> 5
>> 2**100
=> 1267650600228229401496703205376
>> a = 11
=> 11
>> a + 1
=> 12
# Use "help" to access documentation:
>> help "Array#reverse"
>> help "Array"
                              IRB documentation:
http://phrogz.net/ProgrammingRuby/irb.html
```


IRB 2 – Tab Completion and Load

```
# Tab completion
>> require 'irb/completion'
>> 1.ro<tab>  ➔ 1.round
>> 1.<tab><tab> ➔ all methods

# If file test.rb contains the line
#   puts "hello"
# load will read and execute it:

>> load 'test.rb'
hello
=> true
```

IRB 3

Underscore (_) contains the value of the last expression evaluated by irb:

```
>> 2**10
=> 1024
>> _
=> 1024
>> _+1
=> 1025
>> _+1
=> 1026
```


IRB 4

If a line you type into IRB is incomplete or erroneous, IRB will often respond with a different prompt

```
>> 1 +
?>
```

If you can't figure out how to correct the line, entering control-D will usually clear the problem and restore the normal prompt. If that doesn't work, try control-C, but that will often terminate IRB, so you'll have to restart it.

Standard Types – Numbers

Integers and floating point numbers.

Integers can be any length, up to the available memory on the system.

Integers that can fit in 30 (or 62) bits are of class **Fixnum**. Larger integers are **Bignum**.

Transparent conversion **Fixnum** ←→ **Bignum**.

Floats correspond to C **double** data type. (Original Ruby interpreter, "Matz' Ruby Interpreter – MRI", is written in C.)

Sample Code – Numbers

```
123456          # Fixnum
-987            # negative Fixnum
123456789123456789 # Bignum
0x7FFFFFFF          # hexadecimal Fixnum
3.14159265358979    # Float
1.0e99           # Float
1.0E-99          # Float

puts 2**100
1267650600228229401496703205376
```

Standard Types - Strings

- In Ruby 1.8.x, strings are sequences of 8-bit bytes.
- In Ruby 1.9.x, strings are sequences of characters.
- Strings can hold binary data, including 0.
- String literal with single-quotes supports only two escape sequences: \\ → \ , \' → '

```
'it\'s a backslant \\' → it's a backslant \
'c:\temp\test.txt '    → c:\temp\test.txt
```

- Double-quoted literals support many escape sequences: \n, \t, \xFF, #{expr}, etc.

Sample Code – Strings

```
s = "a string"
s = "a" + "b" + "c"   → "abc"

s = "this is a string"
s[0]                  → 116 # RUBY UP THRU 1.8.7
s[0]                  → "t" # CHANGED IN RUBY 1.9.0 on
  #                       116 is the ASCII code for 't'

s[0].chr              → "t"
s[0..0]               → "t"              (ranges will be
s[0..3]               → "this"            covered later)

s = "abc"             # << appends to a string
s << "def"            # s is now "abcdef"
```

String Interpolation

```
a = 10
puts "a is #{a}" → a is 10

s = "a is #{if a < 10 then 'small'
     else 'large' end}"
# s is "a is large"
```


Calling a method on an object

```
s = "abcd"
s.length        → 4
"abcd".length → 4
-10.abs         → 10
```

<object to which method belongs>.<method name>

Bang (!) Methods

By convention methods that end with ! are "dangerous" methods, typically because they modify their receiver

```
s = "stressed"

puts s.reverse      → "desserts"
puts s              → "stressed"

puts s.reverse!     → "desserts"
puts s              → "desserts"

# But some methods without ! modify receiver
puts s.concat(" are nice")→ "desserts are nice"
puts s                    → "desserts are nice"

It's usually best not to modify the receiver.
```


Exercise 1

```
"a" + "bc" + "d" → "abcd"

"abc".reverse    → "cba"

"abC".upcase     → "ABC"

"AbC".downcase   → "abc"

"aBc".capitalize → "Abc"

"AbC".swapcase   → "aBc"

"abc".length     → 3

Exercise 1:  Change last letter

to uppercase    "abcde" → "abcdE"
```

Standard Types - Ranges

Ranges as sequences

```
 1..5          → 1,2,3,4,5  # ..  is inclusive
 2..2          → 2
-2..2          → -2,-1,0,1,2
 5..3          → empty.  Range must be increasing
 1...5         → 1,2,3,4    # ... is exclusive
 'a'..'e'      → 'a','b','c','d','e'

 an_array = ["zero", "one", "two", "three", "four"]
 an_array[1..3] → ["one", "two", "three"]

 0...an_array.length → all array indices
```


Standard Types – Arrays and Hashes

- Arrays and Hashes are indexed collections of objects that can be accessed with a key
- Arrays – key is an integer, 0 relative
- Hashes – key can be any object
- Both can hold objects of different types, and grow as needed. Elements are accessed by [ ]

```
a = [ 1, 'foo', -11.3, ['x', 7], 123 ]

a[0] → 1          a[3][0] → 'x'
a[4] → 123        a[1..2] → ['foo', -11.3]
a[5] → nil        a[3..2] → [] # cannot index backwards
```

nil doesn't reference any object (like null in Java or C#, or null pointer in C)

Standard Types – Arrays and Hashes 2

```
a = [0, 1, 2, 3, 4]
a[1..3] = ['a', 'b']  → [0, 'a', 'b', 4]
a[0..1] = []          → ['b', 4]

h = { 1 => 2, 'a' => [1,2], [3,4] => 'cat' }
h['a'] → [1,2]
h[[3,4]] → 'cat'
h[2] → nil  # nil is default for missing entries,
            # default can be changed when hash
            # is created:   h_zero = Hash.new(0)
h[2] = 777   # add new entry
h[2] → 777
h[2] = -1    # modify existing entry
h[2] → -1
```


Standard Types – Arrays and Hashes 3

```
Arrays are indexed from 0, or from the end starting with
  -1

a = ["zero", "one", "two", "three"]
       0       1       2       3
      -4      -3      -2      -1

a[-1]     → "three"
a[-3..-1] → ["one", "two", "three"]
a[0..-3]  → ["zero", "one"]  # can mix positive and neg
a[-5]     → nil
a[-1..-2] → []   # cannot index backwards

<< appends to an array
a << "four"
# a is now ["zero", "one", "two", "three", "four"]
```

Useful String and Array methods

```
"a big cat".split → ["a", "big", "cat"]
"cat".split("") → ["c", "a", "t"]
["c", "a", "t"].join → "cat"
["c", "a", "t"].join(" ") → "c a t"
["a", 7, [4, 6]].length → 3
[3,1,4,1,5].sort → [1,1,3,4,5]
[1,2,3,4,5].reverse → [5,4,3,2,1]
[1,2] + [3] + [4,5]  → [1,2,3,4,5]
"aaaaabbaaaaa".squeeze →   "aba"
```


Exercise 2

2-a) Sort the words in a string alphabetically then determine how long the alphabetically first word is.

For "this is a sentence" the answer should be 1 (because 'a' sorts first.)

2-b) What are the unique characters in a string?

For "aardvark" the answer should be "adkrv"

Variables and Constants

Variable names and method names consist of the characters a-z, A-Z, 0-9, and _ starting with a lowercase letter or _. Note that methods and variables look the same. By convention, these names are all lowercase with an underscore between words:

a, x, some_value, the_last_word_on_the_page, x1

Constants are named like variables, except they start with an uppercase letter. By convention, they are all uppercase:

PI, EPSILON, MAX_VALUE

Class names are constants, so they start with an uppercase letter. By convention, they are written without underscores, with each word capitalized (CamelCase):

Person, Shape, FileInformation


```
f = 212
c = ((f - 32) * 5) / 9
puts c
```

If we saved the above as the file ftoc.rb, we could run it as follows:

```
c:\ruby>ruby ftoc.rb
100
```

```
print "Enter Fahrenheit: "
input = gets        # Read from stdin
f = input.to_i
c = ((f - 32) * 5) / 9
puts "Centigrade: #{c}"
```

If we saved the above as the file ftoc2.rb, we could run it as follows:

```
c:\ruby>ruby ftoc2.rb
Enter Fahrenheit: 32
Centigrade: 0
```


Define a method

```
def ftoc(f)
  # return keyword is optional
  return ((f - 32) * 5) / 9
end

print "Enter Fahrenheit: "
f = gets.to_i
c = ftoc(f)
puts "Centigrade: #{c}"
```

IRB 5

Use irb to experiment with code you've written:

```
C:\ruby>irb --prompt simple
>> load "ftoc2.rb"
Enter Fahrenheit: 212
Centigrade: 100
=> true
>> ftoc(32)
=> 0
```


Conditional Execution

```
if x < 100
  puts x
end

if x < 100 then puts x end

if x < 100
  puts "x is small"
else
  puts "x is large"
end
```

Conditional Execution 2

```
if x < 100
  puts "x is small"
elsif x < 500
  puts "x is medium"
else
  puts "x is large"
end
```


Conditional Execution 3

These all do the same thing:

```
if x != 1
  ...

if ! (x == 1)
  ...

if not x == 1
  ...

unless x == 1
  ...
```

Statement Modifiers

```
c = b/a if a != 0
# is the same as
if a != 0
  c = b / a
end

# these two lines do the same thing
puts i if i >= 0
puts i unless i < 0
```


Case Expressions 1

```
case x
when -1
  puts 'negative one'
when 0, 2, 4, 6, 8
  puts 'even'
when 1, 3, 5, 7, 9
  puts 'odd'
when 10
  puts 'ten'
else
  puts 'unexpected value'
end
```

Case Expressions 2

```
case
when x < -10
  puts 'small'
when x == 10, x == 100, x == 1000
  puts 'power of ten'
when x > 10
  puts 'large'
when z == 0
  puts 'z is zero'
else
  puts 'some other value'
end
```


Loops

```
for index in 1..10 do      # all three loops produce
  puts index               # the same output
end

index = 1
while index <= 10
  puts index
  index += 1               # note index++ not supported
end                        # in Ruby

index = 1
until index > 10
  puts index
  index += 1
end

# --------------------

do_something while not_finished
do_something until finished
```

Code Blocks - Overview

- A block is a list of statements within { } or within do end optionally starting with parameters within | |
- Blocks may only appear adjacent to a method call. Cannot arbitrarily enclose statements in { } as in C/C++/C#, etc.
- A call to _any_ method can be followed by a block.
- The block might not be executed. How and when the block is executed is determined by the method.
- An iterator is any method that can invoke a block. It calls "yield" whenever it generates a new value. No need for helper classes to hold the iterator state, as in C++, C# and Java

Code Blocks 2

```
object.method_name {
  # the block code goes here
}

object.method_name do
  # the block code goes here
end
```

Code Blocks 3 - yield

```
def block_test
  puts "before yield"
  yield
  puts "after yield"
end

block_test { puts "in the block"}

before yield
in the block
after yield
```


Code Blocks 4 – no yield

```
def block_test2
  puts "in block_test2"
end

block_test2 { puts "doesn't get called"}

in block_test2
```

Code Blocks 5

```
def block_test3
  yield 1
  yield "hi"
  yield 11.99
end

block_test3 { |x| puts x}

1
hi
11.99
```


Code Blocks 6

```
def block_test4
  yield 1, 2
  yield "hi", 77
  yield 11.99
  yield 1, 2, 3
end

block_test4 {|x,y| puts "#{x}   #{y}" }

1   2
hi   77
11.99   # y is nil, which does not print
1   2   # the 3 is silently discarded
```

Code Blocks 7

```
def print_conversions(values)
  for value in values
    converted = yield value
    puts "#{value} converts to #{converted}"
  end
end

print_conversions([212, 32, 0]) {|v| ((v -
  32) * 5) / 9}

212 converts to 100
32 converts to 0
0 converts to -18
```


Code Blocks 8

```
def print_conversions(values)
  for value in values
    converted = yield value
    puts "#{value} converts to #{converted}"
  end
end

print_conversions([212, 32, 0]) {|v| v*v}

212 converts to 44944
32 converts to 1024
0 converts to 0
```

Code Blocks 9

```
[1, 2, 3, 4, 5].each {|x| print x}
12345

# Set implements a collection of unordered
# values with no duplicates
require 'set'
s = [1, 2, 3, 4, 5].to_set
# s is now the Set containing 1,2,3,4,5

s.each {|x| print x}
12345
```


Code Blocks 10

We can easily write our own version of each. Note that all classes are always open, so we can add or replace methods.

```
class Array
  def my_each
    for i in 0...self.length
       yield self[i]
    end
  end
end

[1, 2, 3, 4, 5].my_each {|x| print x}
12345
```

Code Blocks 11

```
require 'set'
a = [1, 2, 3, 4, 5, 6]
s = [1, 2, 3, 4, 5, 6].to_set

a.select {|x| x < 4}  #  [1, 2, 3]
s.select {|x| x < 4}  #  [1, 2, 3]

a.reject {|x| x == 3} #  [1, 2, 4, 5, 6]
s.reject {|x| x == 3} #  [1, 2, 4, 5, 6]
```


Code Blocks 12

```
>> [1,9,3,2,4].sort
=> [1, 2, 3, 4, 9] # default sort behavior without any block

# the following block is the same as the default
# comparison, so the result will be the same as above
>> [1,9,3,2,4].sort {|a,b| a <=> b}
=> [1, 2, 3, 4, 9]

# we supply a block which reverses the order of the
# comparison, so this now sorts in descending order
>> [1,9,3,2,4].sort {|a,b| b <=> a}
=> [9, 4, 3, 2, 1]
```

the standard Ruby comparison operator, <=>, returns -1 if the first value is less than the second, 0 if the values are equal, or +1 if the first value is greater than the second.

Break

Static vs. Dynamic Typing

- In statically typed languages like C/C++/C#/Java, variables have types but data does not.
- In dynamically typed languages like Ruby, data has types but variables do not.

```
a = 1
a = "abc"
```

- Cannot specify type of a variable in Ruby.
- "Static typing is premature optimization"
- Dynamic typing helps keep software "soft". Supports refactoring and testing.

"The Far Side" by
Gary Larson,
October 1987

Testing socket code

```
require 'socket'

def send_hello_world(socket)
  socket << "hello "
  socket << "world"
end

tcp_socket = TCPSocket.new("127.0.0.1", 20000)
send_hello_world(tcp_socket)
```

Testing socket code 2

```
require 'socket'

def send_hello_world(socket)
  socket << "hello "
  socket << "world"
end

test_socket = ""
send_hello_world(test_socket)
p test_socket

"hello world"
```


Testing socket code 3

```
require 'socket'

def send_hello_world(socket)
  socket << "hello "
  socket << "world"
end

test_socket = []
send_hello_world(test_socket)
p test_socket

["hello ", "world"]
```

Object Oriented Programming

- Everything is an object – no wrappers as in Java or C#

```
-3.abs          → 3
4.+(3)          → 7
4.+ 3           → 7  # parentheses are optional
4 + 3           → 7  # syntactic sugar
```

- Even literal constants are objects

```
"this is a test".length     → 14
"Madam, I'm Adam".reverse   → "madA m'I ,madaM"
"hello " * 3                → "hello hello hello "
```

- Every procedure is a method. All code in Ruby is a method of some object.

OOP Example - Course Enrollment

We have courses, which have a title and a maximum number of students that can enroll. We would like to be able to enroll a student in a particular class.

We would like the course to be able to tell us its title, its maximum number of students, and the number of students currently enrolled.

```
class Course              # version 1 on handout
  def title
    "Object-Oriented Programming in Ruby"
  end

  def capacity
    40
  end

  def enrolled
    9
  end
end
```



```
c = Course.new
puts c.title
puts "#{c.enrolled} students are enrolled out of a
maximum of #{c.capacity}"

Object-Oriented Programming in Ruby
9 students are enrolled out of a maximum of 40

puts c
#<Course:0x399464>
```

```
def to_s       # version 2 on handout
  "#{title}\n#{enrolled} " +
  "students are enrolled out of a maximum of " +
  "#{capacity}"
end
puts c

Object-Oriented Programming in Ruby
9 students are enrolled out of a maximum of 40
```


Instance (object) Variables

```
# we do not need to (and cannot) declare @enrolled
# @enrolled is not accessible from outside this class
def initialize              # version 3 on handout
  @enrolled = 0
end

def enrolled
  @enrolled
end

def enrolled=(value)
  @enrolled = value
end
```

OOP – "Tell, don't Ask" – Adding a Student

```
n = c.enrolled
n = n + 1
c.enrolled = n

# -----  old above, new below  -----
# no longer need enrolled=() method
def add_student     # version 4 on handout
  @enrolled += 1
end

puts c.enrolled → 0
c.add_student
puts c.enrolled → 1
```



```
# version 5 on handout
def available?   # note ? as part of method name
  enrolled < capacity
end

puts c.available?
True

if c.available?
  puts "There's room in the course"
else
  puts "Sorry, the course is full"
end
There's room in the course
```

Exceptions

- Eliminate the need to use return codes to indicate an error. The return code technique is notoriously error prone.
- Cleanly separate "normal" and "error handling" code
- Simplify logic by removing "if" logic used to handle errors

```
# raises a RuntimeError with specified message
raise "Description of Runtime Error"

raise ArgumentError, "Argument is negative"
```


Catching Exceptions

```
begin
  x = Math.sqrt(y/z)
  # …
rescue ArgumentError
  puts "Error taking square root."
rescue ZeroDivisionError
  puts "Tried to divide by zero."
end
```

Catching Exceptions – General Case

```
begin
  # …
rescue AnExceptionType
  # Process AnExceptionType
rescue AnotherExceptionType => ex
  # Process AnotherExceptionType
  # Exception stored in ex
rescue
  # Process any StandardError
else
  # Execute if no exceptions
ensure
  # This code is always executed,
  # whether or not an exception occurs
end
```


Exception Hierarchy (Incomplete)

```
Exception
    NoMemoryError
    ScriptError
    SignalException
    StandardError   # caught by rescue with no
                    # exception class specified
        ArgumentError
        IOError
            EOFError
        NameError
            NoMethodError
        RuntimeError
        ZeroDivisionError
```

OOP - Invariants

Invariants are an important part of class design. Without exceptions, invariants are hard to enforce.

One invariant for a Course is:

```
enrolled <= capacity
```


OOP – Invariants

```
def add_student
  @enrolled += 1
end
# -----  old above, new below  -----
def add_student     # version 6 on handout
  if ! available?
    raise "Course capacity exceeded"
  else
    @enrolled += 1
  end
end
```

OOP – Invariants 2

```
c = Course.new
50.times {c.add_student}

:-================ RuntimeError =========
C:\course\oop.rb:30:in `add_student'
        raise "Course capacity exceeded"
C:\course\oop.rb:42
    50.times {c.add_student}
C:\course\oop.rb:42:in `times'
    50.times {c.add_student}
C:\course\oop.rb:42
    50.times {c.add_student}
==============================================
Exception: Course capacity exceeded
```



```
def enrolled
  @enrolled
end
# -----  old above, new below  -----
# version 7 on handout
attr_reader :enrolled



# attr_accessor :enrolled
# would provide both enrolled and enrolled=
```

Symbols

A symbol is written by preceding a name with a colon:

:up, :title, :some_name

Symbols act somewhat like read-only strings (although internally they're represented differently), and are used for labeling or naming. They are typically used to specify the name of a variable or method, without actually invoking that method (see previous slide).

OOP Example - Student Class

Currently, the Course class keeps track of only the number of students that are enrolled. We'd like it to keep track of the individual students.

We'll create a Student class. Student objects will have a name and a student id.

```
class Student      # version 8 on handout
  attr_reader :name, :student_id

  def initialize(name, student_id)
    @name       = name
    @student_id = student_id
  end

  def to_s
    "student #{name}, id #{student_id}"
  end
end

s = Student.new("Joe", 123)
puts s
student Joe, id 123
```


Let's add an array to Course to hold all the students who are in that course:

```
# version 9 on handout
class Course
  attr_reader :enrolled
  attr_reader :students

   def initialize
    @enrolled = 0
    @students = []
  end
```

```
  def add_student
    if ! available?
      raise "Course capacity exceeded"
    else
      @enrolled += 1
    end
  end
# -----  old above, new below  -----
  def add_student(student)
    if ! available?
      raise "Course capacity exceeded"
    else
      @students << student
    end
  end
```


Uniform Access Principle - Bertrand Meyer

```
  attr_reader   :enrolled
# -----  old above, new below  -----
  def enrolled
    @students.length
  end

# but clients still have the same interface:
course = Course.new
puts course.enrolled

# we can now remove these lines from Course:
  attr_reader :enrolled
  @enrolled = 0
```

Using Course and Student

```
joe = Student.new("Joe", 123)
mary = Student.new("Mary", 456)

course = Course.new
course.add_student(joe)
course.add_student(mary)
puts course.enrolled
2

puts course.students
student Joe, id 123
student Mary, id 456
```


Inheritance

```
class OnlineCourse < Course
  def available?
    true
  end

  def to_s
    "#{title}\n#{enrolled} students are enrolled"
  end
end
```

Unit Testing - Failure

```
require 'test/unit'

def add_one(x)
  x               # this is wrong
end

class TestAddOne < Test::Unit::TestCase
  def test_1
    # first parameter is expected, second is actual
    assert_equal(2, add_one(1))
    assert_equal(-4, add_one(-5))
  end
end
```


Unit Testing – Failure Output

```
Started
F
Finished in 0.009 seconds.

  1) Failure:
test_1(TestAddOne) [../RubyCourseUnitTesting.rb:9]:
<2> expected but was
<1>.

1 tests, 1 assertions, 1 failures, 0 errors

Program exited with code 0
```

Unit Testing - Success

```
require 'test/unit'

def add_one(x)
  x + 1              # this is correct
end

class TestAddOne < Test::Unit::TestCase
  def test_1
    # first parameter is expected, second is actual
    assert_equal(2, add_one(1))
    assert_equal(-4, add_one(-5))
  end
end
```


Unit Testing – Success Output

```
Started
.
Finished in 0.009 seconds.

1 tests, 2 assertions, 0 failures, 0 errors

Program exited with code 0


For another approach to unit testing in Ruby,
see RSpec: http://rspec.info/
```

Dynamism makes static checking difficult

```
class Test
  def say_hi
    puts "hi"
  end
end

def some_method
  t = Test.new
  t.say_bye
end
```


Dynamism makes static checking difficult

```
class Test
  def say_hi
    puts "hi"
  end
end

def some_method
  t = Test.new
  t.say_bye
end
```

```
# in a separate file:

class Test
  def say_bye
    puts "bye"
  end
end

some_method →  bye
```

How to redefine plus

```
class Fixnum
  def +(other)
    42
  end
end

puts 2+2

42
```


A more useful redefinition of plus

```
class Fixnum
  alias old_plus +

  def +(other)
    puts "adding #{self} and #{other}"
    old_plus(other)
  end
end

puts 2+2

adding 2 and 2
4
```

Complete Quicksort in C++ by Alex DeCastro

```
template<class T>
void quicksort(T a[], int l, int r)
{
    if (l >= r) return;
    int i, j;
    int num_left, num_right;
    T pivot, temp;

    while (l < r) {
        i = l;
        j = r + 1;
        pivot = a[l];
        while (true) {
            do {
                i = i + 1;
            } while ((a[i] < pivot) && (i < r));

            do {
                j = j - 1;
            } while ((a[j] > pivot) && (j > l));

            if (i >= j) break;
            temp = a[i];
            a[i] = a[j];
            a[j] = temp;
        };

        a[l] = a[j];
        a[j] = pivot;
        num_left  = (j-1) - l;
        num_right = r - (j+1);

        if (num_left <= num_right) {
            quicksort(a, l, j-1);
            l = j+1;
        }
        else {
            quicksort(a, j+1, r);
            r = j-1;
        }
    }
}
```

Complete code for Quicksort in C++, written by Alex DeCastro, M.S. degree in Computer Science from U.C. Santa Barbara. Visualization Scientist at the San Diego Supercomputer Center. from http://users.sdsc.edu/~decastro/home/projects/datastruct/quicksort.cpp

Partial code for Quicksort in C++

```
num_left  = (j-1) - l;
num_right = r - (j+1);
if (num_left <= num_right) {
      quicksort(a, l, j-1);
      l = j+1;
}
else {
      quicksort(a, j+1, r);
      r = j-1;
}
```

Quicksort in 7 lines of Ruby

```
def quicksort(a)
  return a if a.size <= 1
  p = a.first     # pivot element
  quicksort(a.select {|i| i < p }) +
            a.select {|i| i == p}  +
  quicksort(a.select {|i| i > p })
end

a = [6,5,9,9,7,3,4,2,3,4,1]
quicksort(a)  → [1,2,3,3,4,4,5,6,7,9,9]
```


Mutable strings and aliasing

In Ruby (as in C and C++) strings are mutable (they can be changed). In contrast, in Java and C# strings are immutable (they cannot be changed after they are created).

```
name = "Sam"
name2 = name
name[0] = "P"

puts name  → "Pam"
puts name2 → "Pam"  # name2 is the same string as name

name3 = name.dup # name3 is a new duplicate of name
name[0] = 'Z'

puts name  → "Zam"
puts name2 → "Zam"
puts name3 → "Pam"
```

Threading

```
$stdout.sync = true

Thread.abort_on_exception = true

Thread.new do
  loop do
    puts 'a'
    sleep rand
  end
end

loop do
  puts 'b'
  sleep rand
end
```

```
Output:
ab

b
b
a
a
b
a
b
a
b
a
b
b
a
b
a
b
a
b
a
```


Standard Types – Regular Expressions

- Regular expressions are first class objects, as in Perl, awk, and sed. Most other languages implement regular expressions as strings which are passed to functions like compile() or match().
- Supports much of the Perl 5 Regular Expression functionality
- /abc/ does a case-sensitive match for "abc"
- /[abc]/ does a case-sensitive match for any one of "a", "b", or "c"

Sample Code – Regular Expressions

```
if line =~ /abc/
  puts "Found a match: #{line}"
end

while gets      # read from stdin to $_
  print if /error/
end

s = "this is a string"
s.scan("is")        → ["is", "is"]
s.scan(/[aeiou]/)   → ["i", "i", "a", "i"]
```


"name = x" ambiguity

```
class X
  # other code omitted here
  def y
    z = 1
  end
end

somename = something
```

Ruby always assumes that somename is a local variable.

```
self.somename = something
```

Will call the somename= method

```
@somename = something
```

Will assign to the instance variable @somename

File I/O

```
f = open("/etc/passwd")
while line = f.gets()
   print line if /matz/ =~ line
end
f.close
matz:x:1000:1000:Yukihiro Matsumoto:/home/matz

File.foreach("/etc/passwd") { |line|
   print line if /matz/ =~ line
}
matz:x:1000:1000:Yukihiro Matsumoto:/home/matz

puts File.open("/etc/passwd").grep(/matz/)
matz:x:1000:1000:Yukihiro Matsumoto:/home/matz
```


Profiling

```
require 'profile'
1.upto(10) do |i|
  s = i.to_s
  1.upto(20) do |k|
    x = k.to_f
  end
end

# Profiling is implemented in 60 lines of
# Ruby. Look in:
# ruby/src/ruby-1.8.x/lib/profile.rb
```

Gems

on Mac OS X may need to do "sudo gem ..."

C:\>gem install ruby-breakpoint

Successfully installed ruby-breakpoint-0.5.1

Installing ri documentation for ruby-breakpoint-0.5.1...

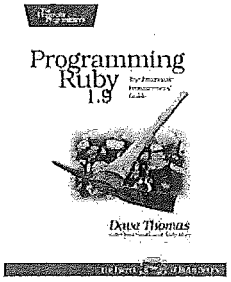
Installing RDoc documentation for ruby-breakpoint-0.5.1...

GUI with Tk

```
require 'tk'

root = TkRoot.new { title "Sample" }

label = TkLabel.new(root) { text "This is a
sample application for the Ruby Course" }

label.pack

Tk.mainloop
```

Recommended Books

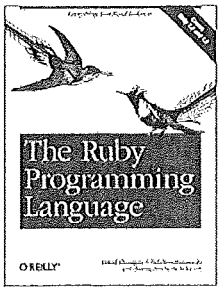
Programming Ruby 1.9 by Dave Thomas with Chad Fowler and Andy Hunt



The first English language book on Ruby, and still the standard. Excellent for learning Ruby.

The second half of the book is documentation on the built-in classes and modules, and on the standard library.

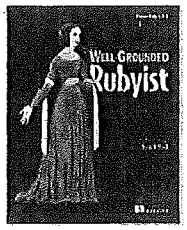
The Ruby Programming Language by David Flanagan and Yukihiro Matsumoto "Matz"



Does the same thing for Ruby that "The Java Programming Language" or "The C++ Programming Language" does for those languages.

Thorough coverage of the entire language, in slightly less detail than an actual language spec. Since Ruby (unfortunately) has no written language spec, this is the next best thing.

The Well-Grounded Rubyist by David A. Black



An excellent introduction to Ruby, especially for the experienced programmer.

Covers a number of topics with a clarity and depth I haven't seen in any other book.

Links

www.ruby-lang.org	Main Ruby Site
www.ruby-doc.org	Ruby documentation
www.ruby-lang.org/en/downloads/	Download Ruby
www.rubyinstaller.org/	Download Ruby for Windows
netbeans.org	NetBeans IDE (free)
www.aptana.com	Aptana Studio IDE (free)
www.embarcadero.com/products/3rdrail	3rd Rail IDE (commercial)
www.jetbrains.com/ruby	RubyMine IDE (commercial)
www.zenspider.com/Languages/Ruby/QuickRef.html	Quick Reference
www.rubycentral.org	RubyCentral non-profit. Organizes RubyConf
rubyforge.org	Open Source Ruby projects
raa.ruby-lang.org	Ruby Application Archive – more projects
blade.nagaokaut.ac.jp/ruby/ruby-talk/index.shtml	RubyTalk mailing list
t-a-w.blogspot.com/2006/11/magichelp-for-ruby.html	IRB help extension
tryruby.org/	Web based interactive Ruby

The End