Ruby (on Rails)

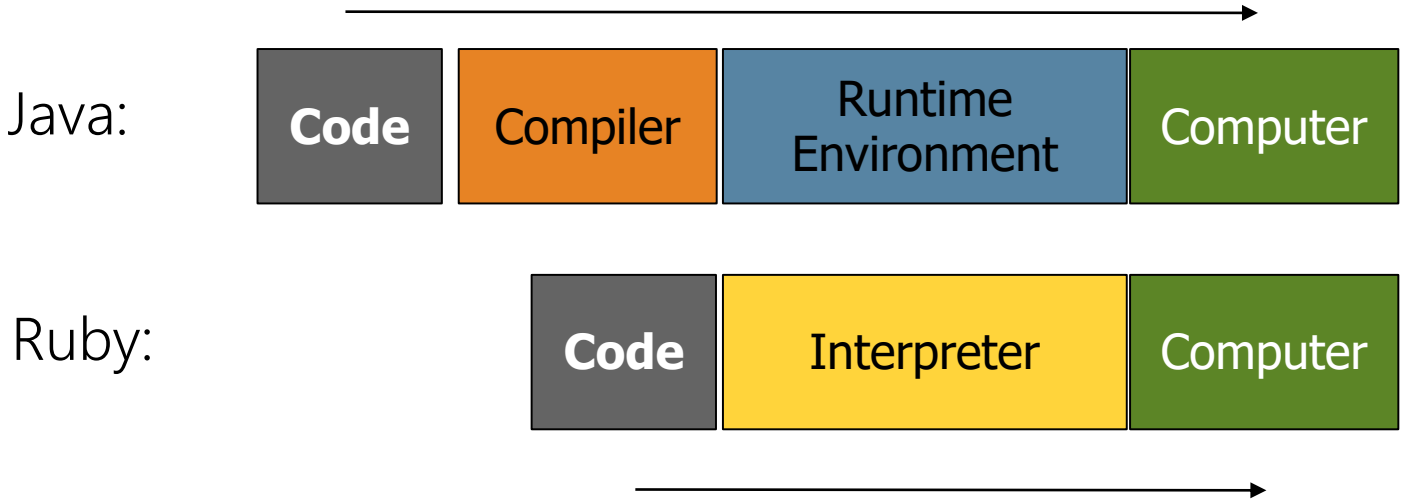
Introduction

What is Ruby?

- Programming Language
- Object-oriented
- Interpreted

Interpreted Languages

- Not compiled like Java
- Code is written and then directly executed by an **interpreter**
- Type commands into interpreter and see immediate results



My First Ruby Program

File: hello.rb

```
puts "hello world!"
```

Running Ruby Programs

- Use the Ruby interpreter

  ruby hello.rb

  - "ruby" tells the computer to use the Ruby interpreter
- Interactive Ruby (irb) console

  irb

  - Get immediate feedback
  - Test Ruby features

puts vs. print

- "puts" adds a new line after it is done
  - analogous System.out.println()

- "print" does not add a new line
  - analogous to System.out.print()

Comments

```
# this is a single line comment

=begin
  this is a multiline comment
  nothing in here will be part of the code
=end
```

Variables

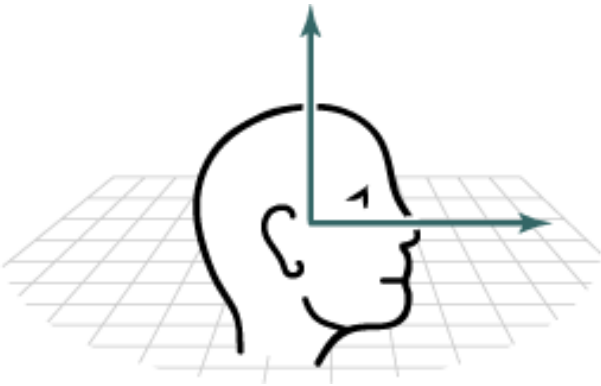
- Declaration – No need to declare a "type"
- Assignment – same as in Java
- Example:

```
x = "hello world"        # String
y = 3                    # Fixnum
z = 4.5                  # Float
r = 1..10                # Range
```

Ruby is Truly Object-Oriented

All classes derived from **Object** including **Class** (like Java) but there are no primitives (not like Java at all). EVERYTHING is an Object.

- Common Types (Classes): Numbers, Strings, Ranges



Ruby is Truly Object-Oriented

- You can find the class of any variable

```
x = "hello"
x.class
```

- You can find the methods of any variable or class

```
x = "hello"
x.methods
String.methods
```

REMEMBER!

- `a.b` means: call method `b` on object `a`
  - `a` is the *receiver* to which you *send* the method call, assuming `a` will *respond to* that method
- *does not mean*: `b` is an instance variable of `a`
- *does not mean*: `a` is some kind of structure of which `b` is a member

- Example: 1+ 2 ⇔ 1.send(:+, 2)

Understanding this distinction will save you from much grief and confusion

Ruby is Truly Object-Oriented

- There are many methods that all Objects have
- Include the "?" in the method names, it is a Ruby naming convention for boolean methods
  - nil?
  - eql? or equal?
  - ==, !=
  - to_s

Numbers

- Numbers are objects
- Different Classes of Numbers

3.eql? 2	→	false
-42.abs	→	42
3.4.round	→	3
3.2.ceil	→	4
3.8.floor	→	3
3.zero?	→	false

String Methods

"hello world".length → 11

"hello world".nil? → false

"ryan" > "kelly" → true

"hello_world!".instance_of? String → true

"hello" * 3 → "hellohellohello"

"hello" + " world" → "hello world"

"hello world".index("w") → 6

Arrays

- Ruby arrays…
  - Are indexed by zero-based integer values
  - Store an assortment of types within the same array
  - Are declared using square brackets, [], elements are separated by commas
- Example:

```
a = [1, 4.3, "hello", 3..7]
a[0]    →    1
a[2]    →    "hello"
```

Sorting

```
a = [5, 6.7, 1.2, 8]
a.sort          →   [1.2, 5, 6.7, 8]
a               →   [5, 6.7, 1.2, 8]
a.sort!         →   [1.2, 5, 6.7, 8]
a               →   [1.2, 5, 6.7, 8]
a[4] = "hello"  →   [1.2, 5, 6.7, 8, "hello"]
a.sort          →   Error
```

Negative Integer Index

- Negative integer values can be used to index values in an array

- Example:

```
a = [1, 4.3, "hello", 3..7]
a[-1]          →    3..7      # the last one
a[-2]          →    "hello"
a[-3] = 83.6
a              → [1, 83.6,  "hello", 3..7]
```

Symbols

- Symbols seem to be peculiar to Ruby. They begin with a colon.
  - :name, :age, :course
- Symbols are not Strings
  - a = :name
  - a.class
  - a.eql? :name
  - a.eql? :age
  - :name.eql? 'name'

Hashes

- Arrays use integers as keys for a particular values (zero-based indexing)
- Hashes, also known as "associative arrays", have Objects as keys instead of integers
- Declared with curly braces, {}, and an arrow, "=>", between the key and the value

Hashes with symbols

Example:

```
h = {"greeting" => "hello", "farewell”=>"goodbye"}
h["greeting"]                    #→"hello”

person = {:name => "Mike", :age => 30}
puts person

person = { name: "Mike", age: 30}
puts person
```

if/elsif/else/end

- Must use "elsif" instead of "else if“
- Example:

```
if (age < 35)
  puts "young whipper-snapper"
elsif  (age < 105)
  puts "80 is the new 30!"
else
  puts "wow… gratz..."
end
```

Inline "if" statements

- Original if-statement

```
if age < 105
  puts "don't worry, you are still young"
end
```

- Inline if-statement

```
if age < 105 then puts "don't worry, you are still young"  end

puts "don't worry, you are still young" if age < 105

puts age < 105 ? "don't worry, you are still young"  : ""
```

unless

- "unless" is the logical opposite of "if"

- Example:

```
unless age >= 105          # if (age < 105)
    puts "young."
else
    puts "old."
end
```

loops

- Example :

```
num = 5
loop do
   num+=1
   puts num
   if num==10 then
      break
   end
end
```

while-loops

- Example:

```
i=7
while i<=10
  print i,"\n"
  i+=1;
end
```

until

- Similarly, "until" is the logical opposite of "while"
- Can specify a condition to have the loop stop (instead of continuing)
- Example

```
i = 0
until (i >= 5)          # while (i<5)
  puts i
  i +=1
end
```

Blocks

- Blocks are simply "blocks" of code
- They are defined by curly braces, {}, or a do/end statement
- They are used to pass code to methods and loops

Block as Parameter

- New block

```
p = Proc.new { puts 'This is a block' }
p.call
```

- Receive block

```
def meth1(p1, p2, &block)
  puts "Class is " + block.class.to_s
  block.call
end
meth1(1, 2)  { puts "This is a block" }
```

Blocks

- In Java, we were only able to pass parameters and call methods
- In Ruby, we can pass code through blocks
- For examplet, the times() method takes a block:

```
3.times { puts "hello" }
    # the block is the code in the {}
```

Blocks and Parameters

- Blocks can also take parameters
- For example, our times() method can take a block that takes a parameter. It will then pass a parameter to the block
- Example

```
3.times {|n| puts "hello" + n.to_s}
```

- Here "n" is specified as a parameter to the block through the vertical bars "|"

Iterators

- An iterator is simply "a method that invokes a block of code repeatedly" (Pragmatic Programmers Guide)
- Iterator examples: Array.each, Range.each,…
- Examples:

```
[1,2,3,4,5].each { |i| puts i * i }
(1..5).each { |i| puts i*i }
"hello".each_byte {|n| puts n.chr}
```

Block

- Block in the form of { }

```
prices = [9.00, 5.95, 12.50]
prices.each  { |price| puts "The next item costs "
  + price.to_s }
```

- Block in the form of do…end

```
prices = [9.00, 5.95, 12.50]
prices.each do |price|
  puts "The next item costs " + price.to_s
end
```

- Common to use iterators instead of loops

```
0.upto(5) { |x| puts x }      #prints 0 through 5

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

0.step(10, 2) do |i|
  print i, " "
end                                  # 0 2 4 6 8 10

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

["apple", "orange", "banana",
  "watermelon"].grep(/an/) do |fruit|
   puts fruit
end
```

```
factorial = 1
1.upto(10) do |i|
  factorial*= i
end
puts factorial

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

puts (1..10).reject {|i| i % 3 != 0 }

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

longest = %w{ cat sheep bear }.inject do |memo,word|
    memo.length > word.length ? memo : word
end
print "Longest Word:", longest, "\n"
```

Methods

- Structure

```
def method_name( parameter1 , parameter2,
  ...)
  statements
end
```

- Simple Example:

```
def print_X
  puts "X"
end
```

Parameters and Return

- Ruby methods return the value of the last statement in the method, so…

```
def add(num1, num2)
  sum = num1 + num2
  return sum
end
```

can become

```
def  add num1, num2
  num1 + num2
end
```

Call:

add(1,2) **or** **add 1, 2**

Class

- Writing a new class is simple!

```
class Person
end
```

- But we may want to initialize state (constructor initialize())

```
class Person
  def initialize(name, gender, age)
     @name = name
     @gender = gender
     @age = age
    puts @name
  end
end
people = Person.new('Tom', 'male', 15)
# instance variables is @parameter_name
```

Instantiating New Objects

- We instantiate a new object by calling the new() method on the class we want to instantiate
- Example

```
people = Person.new('Tom', 'male', 15)
```

- How do we get the @name of people?

```
people. @name ?
p. name ?
```

Accessing State

- Instance variables are private by default
- The instance variables for our Person class are

  @name, @gender, @age
- To access them, we must write methods that return their value
  - Remember "encapsulation"

Accessing State

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

  def x
   @x
  end
end

p = Point.new(2, 3)
puts "p.x = " + p.x.to_s        # get value of instance variable
  by calling a method
```

Accessing State

- We do not need to write these methods by hand
- Example:

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

- What if we want to assign values?

Accessing State

- To assign a value to @x, we can write a method
- Example:

```
def set_x(x)
  @x = x
end
p.set_x(7)
```

- Similarly we can use attr_writer

```
attr_writer :x, :y
```

Accessing State

- If we want to read and write all of our instance variables, we can combine attr_reader and attr_writer to simplify our class, replacing them with attr_accessor

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

Inheritance

- Ruby supports single inheritance
- This is similar to Java where one class can inherit the state and behavior of exactly one other class
- The parent class is known as the superclass, the child class is known as the subclass

Inheritance

```
class Student < Person
  attr_accessor :age, :school
  def initialize(name, gender, age, school)
      @name = name
      @gender = gender
      @age = age
      @school = school
  end
end

puts Student.new('Mike', 'Male', 20, 'scut').school
```

Module (dependent of implementation class)

```
module Stringify
  def stringify
    case @value
      when 1..5
        "value: 1..5"
      when 5..10
        "value: 5..10"
      else
        "other values"
    end
  end
end
```

Mixin

- a mixin is a class that is mixed with a module. In other words the implementation of the class and module are joined, intertwined, combined, etc.
- a module as a degenerate abstract class. A module can't be instantiated and no class can directly extend it but a module can fully implement methods.

Mixin

```
module Debug
  def print_info
    puts "Class: #{ self.class.name } ; Object ID: #{
    self.object_id }"
  end
end

class A
  include Debug
end
class B
  include Debug
end
```

Inheritance and Mixin

```
require './module‘            # import definitions in file module.rb
class Number
  def intValue
    @value
  end
end

class BigInteger < Number  # inherit from class Number
    # Add instance methods from Stringify
  include Stringify           # mix methods at the instance level

  # Add class methods from Math
  extend Math                 # mix methods at the class level

  def initialize value
    @value = value
  end
end
```

Using it…

```
# Call class method extended from
  Math
a= BigInteger.add -1, 10
puts a.intValue

# Call a method included from
  Stringify
puts a.stringify
```

Thread and Queue

```
queue = Queue.new
consumers = Thread.new do
  5.times do |i|
    obj = queue.deq
    print "consumer: #{i} ; #{obj.to_s} \n"
    sleep(rand(0.05))
  end
end
producers = Thread.new do
  5.times do |i|
    sleep(0.1)
    print "producer: #{i}\n"
    queue.enq("Item #{i}")
  end
end
producers.join
consumers.join
```

Thank You