



Ruby (on Rails)

CSE 190M, Spring 2009

Week 2

Arrays

- Similar to PHP, 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"
```

Arrays

- You can assign values to an array at a particular index, just like PHP
- Arrays increase in size if an index is specified out of bounds and fill gaps with nil

- Example:

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

```
a[4] = "goodbye"
```

```
a → [1, 4.3, "hello", 3..7, "goodbye"]
```

```
a[6] = "hola"
```

```
a → [1, 4.3, "hello", 3..7, "goodbye", nil, "hola"]
```

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
```

```
a[-2]      →      "hello"
```

```
a[-3] = 83.6
```

```
a          → [1, 83.6, "hello", 3..7]
```

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
- Example:

```
h = {"greeting" => "hello", "farewell" => "goodbye"}  
h["greeting"]      →      "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: comparison of Float with  
String failed
```

```
h = {"greeting" => "hello", "farewell" => "goodbye"}
```

```
h.sort → [{"farewell", "goodbye"}, {"greeting", "hello"}]
```

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

Blocks

- In Java, we were only able to pass parameters and call methods
- In Ruby, we can pass code through blocks
- We saw this last week, 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 are block

- Example

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

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

Yield

- yield statements go hand-in-hand with blocks
- The code of a block is executed when a yield statement called

Yield

- A yield statement can also be called with parameters that are then passed to the block
- Example:

```
3.times { |n| puts n }
```
- The "times" method calls yield with a parameter that we ignored when we just printed "hello" 3 times, but shows up when we accepted a parameter in our block

Yield Examples

Code:

```
def demo_yield
  puts "BEGINNING"
  yield
  puts "END"
end
demo_yield { puts "hello" }
```

```
def demo_yield2
  puts "BEGINNING"
  yield
  puts "MIDDLE"
  yield
  puts "END"
end
demo_yield2{ puts "hello" }
```

Output:

```
BEGINNING
hello
END
```

```
BEGINNING
hello
MIDDLE
hello
END
```

Parameters, Blocks, and Yield

- Example:

```
def demo_yield3
  yield 2
  yield "hello"
  yield 3.7
end
demo_yield3 { |n| puts n * 3}
```

- "n" is the value passed by yield to the block when yield is called with arguments

Iterators

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

- Examples:

```
[1,2,3,4,5].find{ |n| Math.sqrt(n).remainder(1)==0} # finds perfect square
[2,3,4,5].find{ |n| Math.sqrt(n).remainder(1)==0} # finds perfect square
[1,2,3,4,5].each { |i| puts i } #prints 1 through 5
[1,2,3,4,5].each { |i| puts i * i } #prints 1 squared, 2 squared..., 5squared
(1..5).each{ |i| puts i*i } #prints 1 squared, 2 squared..., 5squared
```

Iterators and Loops

- Common to use iterators instead of loops
- Avoids off-by-one (OBO) bugs
- Built-in iterators have well defined behavior
- Examples

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

`0.step(10, 2) { |x| puts x }` # 0, 2, 4, 6, 8, 10

`0.step(10,3) { |x| puts x }` # 0, 3, 6, 9

for...in

- Similar to PHP's foreach:

- PHP

```
$prices = array(9.00, 5.95, 12.50)
foreach($prices as $price){
    print "The next item costs $price\n"
}
```

- Ruby

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


for...in

- Previous example

```
prices = [9.00, 5.95, 12.50]
```

```
for price in prices
```

```
  puts "The next item costs " + price.to_s
```

```
end
```

- Can also be written

```
prices = [9.00, 5.95, 12.50]
```

```
prices.each do |price|
```

```
  puts "The next item costs " + price.to_s
```

```
end
```

Strings

- Strings can be referenced as Arrays
- The value returned is the a Integer equivalent of the letter at the specified index
- Example:

```
s = "hello"
```

```
s[1]           →    101
```

```
s[2]           →    108
```

```
s[1].chr       →    "e"
```

```
s[2].chr       →    "l"
```

More Strings

- `chomp` – returns a new String with the trailing newlines removed
- `chomp!` – same as `chomp` but modifies original string

More Strings

- `split(delimiter)` – returns an array of the substrings created by splitting the original string at the delimiter
- `slice(starting index, length)` – returns a substring of the original string beginning at the "starting index" and continuing for "length" characters

Strings Examples

```
s = "hello world\n"
```

```
s.chomp → "hello world"
```

```
s → "hello world\n"
```

```
s.chomp! → "hello world"
```

```
s → "hello world"
```

```
s.split(" ") → ["hello", "world"]
```

```
s.split("|") → ["he", "", "o wor", "d"]
```

```
s.slice(3,5) → "lo wo"
```

```
s → "hello world"
```

```
s.slice!(3,5) → "lo wo"
```

```
s → "helrld"
```

Iterating over String characters

Code

```
"hello".each {|n| puts n}
```

```
"hello".each_byte {|n| puts n}
```

```
"hello".each_byte {|n| puts n.chr}
```

Output

```
"hello"
```

```
104
```

```
101
```

```
108
```

```
108
```

```
111
```

```
h
```

```
e
```

```
l
```

```
l
```

```
o
```

Files as Input

- To read a file, call `File.open()`, passing it the path to your file
- Passing a block to `File.open()` yields control to the block, passing it the opened file
- You can then call `gets()` on the file to get each line of the file to process individually
 - This is analogous to Java's Scanner's `.nextLine()`

Files as Input

- Example (bold denotes variable names)

```
File.open("file.txt") do |input| # input is the file passed to our block
  while line = input.gets        # line is the String returned from gets()
    # process line as a String within the loop
    tokens = line.split(" ")
  end
end
```


Output to Files

- To output to a file, call `File.open` with an additional parameter, "w", denoting that you want to write to the file

```
File.open("file.txt", "w") do |output|  
  output.puts "we are printing to a file!"  
end
```

Writing from one file to another

- If a block is passed, `File.open` yields control to the block, passing it the file.
- To write from one file to another, you can nest `File.open` calls within the blocks

Writing from one file to another

```
File.open("input_file.txt") do |input|
  File.open("output_file.txt", "w") do |output|
    while line = input.gets
      output.puts line
    end
  end
end
end
```