CMSC 330: Organization of Programming Languages

Array, Hashes, Code Blocks, Equality
Arrays and Hashes

- Ruby data structures are typically constructed from Arrays and Hashes
  - Built-in syntax for both
  - Each has a rich set of standard library methods
  - They are integrated/used by methods of other classes
Array

- Arrays of objects are instances of class `Array`
  - Arrays may be heterogeneous
    ```ruby
    a = [1, "foo", 2.14]
    ```
- C-like syntax for accessing elements
  - indexed from 0
  - return `nil` if no element at given index
    ```ruby
    irb(main):001:0> b = []; b[0] = 0; b[0]
    => 0
    irb(main):002:0> b[1]  # no element at this index
    => nil
    ```
Arrays Grow and Shrink

- Arrays are **growable**
  - Increase in size automatically as you access elements
    
    ```ruby
    irb(main):001:0> b = []; b[0] = 0; b[5] = 0; b
    => [0, nil, nil, nil, nil, 0]
    
    • `[]` is the empty array, same as `Array.new`

- Arrays can also **shrink**
  - Contents shift left when you delete elements
    
    ```ruby
    a = [1, 2, 3, 4, 5]
    a.delete_at(3)    # delete at position 3; a = [1,2,3,5]
    a.delete(2)       # delete element = 2; a = [1,3,5]
    ```
Iterating Through Arrays

- It's easy to iterate over an array with `while`
  - `length` method returns array's current length

```ruby
a = [1,2,3,4,5]
i = 0
while i < a.length
    puts a[i]
i = i + 1
end
```

- Looping through elements of an array is common
  - We'll see a better way soon, using code blocks
Arrays as Stacks and Queues

Arrays can model stacks and queues

```javascript
a = [1, 2, 3]
a.push("a")  # a = [1, 2, 3, "a"]
x = a.pop     # x = "a"
a.unshift("b") # a = ["b", 1, 2, 3]
y = a.shift   # y = "b"
```

Note that `push`, `pop`, `shift`, and `unshift` all permanently modify the array.
Hash

- A hash acts like an associative array
  - Elements can be indexed by *any kind* of values
  - Every Ruby object can be used as a hash key, because the Object class has a hash method

- Elements are referred to like array elements

  ```ruby
  italy = Hash.new
  italy["population"] = 58103033
  italy["continent"] = "europe"
  italy[1861] = "independence"
  pop = italy["population"] # pop is 58103033
  planet = italy["planet"] # planet is nil
  ```
Hash methods

- **new(o)** returns hash whose default value is o
  - h = Hash.new("fish"); h["go"]  # returns "fish"
- **values** returns array of a hash’s values
- **keys** returns an array of a hash’s keys
- **delete(k)** deletes mapping with key k
- **has_key?(k)** is true if mapping with key k present
  - has_value?(v) is similar
Hash creation

Convenient syntax for creating literal hashes

• Use `{ key => value, ... }` to create hash table

```perl
credits = {
    "cmsc131" => 4,
    "cmsc330" => 3,
}

x = credits["cmsc330"]  # x now 3
credits["cmsc311"] = 3
```

• Use `{ }` for the empty hash
Quiz 1: What is the output

```python
a = {}
a["foo"] = 1
print a["foo"]
print a["bar"]
```

A. 1
B. 1nil
C. Error
D. foobar
Quiz 1: What is the output

```python
a = {}
a["foo"] = 1
print a["foo"]
print a["bar"]
```

A. 1
B. 1nil
C. Error
D. foobar
Quiz 2: What is the output

```ruby
a = {}
a["Spade"] = [ ]
a["Spade"]["Club"] = "Heart"
puts a["Spade"]["Club"]
```

A. Heart  
B. [ ]  
C. Error  
D. { }

Quiz 2: What is the output

```ruby
a = {}
a[“Spade”] = [ ]
a[“Spade”][“Club”] = “Heart”
puts a[“Spade”][“Club”]
```

A. Heart
B. []
C. Error
D. {}
Quiz 3: What is the output

```python
a = {}
a[1] = "foo"
print a[0]
```

A. Error
B. nil
C. foo
D. Nothing is printed.
Quiz 3: What is the output

```python
a ={}
a[1] = "foo"
print a[0]
```

A. Error  
B. nil  
C. foo  
D. Nothing is printed.
Code Blocks

- A code block is a piece of code that is invoked by another piece of code

- Code blocks are useful for encapsulating repetitive computations
Array Iteration with Code Blocks

- The **Array** class has an **each** method
  - Takes a code block as an argument

```
a = [1,2,3,4,5]
a.each { |x| puts x }
```
More Examples of Code Block Usage

- Sum up the elements of an array

  ```ruby
  a = [1,2,3,4,5]
  sum = 0
  a.each { |x| sum = sum + x }
  printf("sum is %d\n", sum)
  ```

- Print out each segment of the string as divided up by commas (commas are printed trailing each segment)
  - Can use any delimiter

  ```ruby
  s = "Student,Sally,099112233,A"
  s.split(',').each { |x| puts x }
  ```
  ("delimiter" = symbol used to denote boundaries)
Yet More Examples of Code Blocks

```
3.times { puts "hello"; puts "goodbye" } 
5.upto(10) { |x| puts(x + 1) }  
[1,2,3,4,5].find { |y| y % 2 == 0 } 
[5,4,3].collect { |x| -x }
```

- `n.times` runs code block `n` times
- `n.upto(m)` runs code block for integers `n..m`
- `a.find` returns first element `x` of array such that the block returns true for `x`
- `a.collect` applies block to each element of array and returns new array (`a.collect!` modifies the original)
Still Another Example of Code Blocks

```ruby
File.open("test.txt", "r") do |f|
  f.readlines.each { |line| puts line }
end
```

alternative syntax: do … end instead of { … }

- **open** method takes code block with file argument
  - File automatically closed after block executed
- **readlines** reads all lines from a file and returns an array of the lines read
  - Use `each` to iterate
- Can do something similar on strings directly:
- "r1\nr2\n\nr4".each_line { |rec| puts rec }
  - Apply code block to each newline-separated substring
Code Blocks for Hashes

```ruby
population = {}
population["USA"] = 319
population["Italy"] = 60
population.each { |c,p|
  puts "population of #{c} is #{p} million"
}
```

- Can iterate over keys and values separately

```ruby
population.keys.each { |k|
  print "key: ", k, " value: ", population[k]
}

population.values.each { |v|
  print "value: ", v
}
```
Using Yield To Call Code Blocks

- Any method can be called with a code block
  - Inside the method, the block is called with `yield`
- After the code block completes
  - Control returns to the caller after the yield instruction

```ruby
def countx(x)
  for i in (1..x)
    puts i
    yield
  end
end

countx(4) { puts "foo" }
```

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>foo</td>
</tr>
<tr>
<td>2</td>
<td>foo</td>
</tr>
<tr>
<td>3</td>
<td>foo</td>
</tr>
<tr>
<td>4</td>
<td>foo</td>
</tr>
</tbody>
</table>
So What Are Code Blocks?

- A code block is just a special kind of method
  - `{ |y| x = y + 1; puts x }` is almost the same as
  - `def m(y) x = y + 1; puts x end`

- The `each` method takes a code block as a parameter
  - This is called higher-order programming
    - In other words, methods take other methods as arguments
    - We’ll see a lot more of this in OCaml

- We’ll see other library classes with `each` methods
  - And other methods that take code blocks as arguments
  - As we saw, your methods can use code blocks too!
Ranges

1..3 is an object of class Range
• Integers between 1 and 3 inclusively

1…3 also has class Range
• Integers between 1 and 3 but not including 3 itself.

Not just for integers
• ‘a’..’z’ represents the range of letters ‘a’ to ‘z’
• 1.3…2.7 is the continuous range [1.3,2.7)
  ➢ (1.3…2.7).include? 2.0 # => true

Discrete ranges offer the each method to iterate
• And can convert to an array via to_a; e.g., (1..2).to_a
Object Copy vs. Reference Copy

Consider the following code

- Assume an object/reference model like Java or Ruby
  - Or even two pointers pointing to the same structure

```python
x = "groundhog" ; y = x
```

Which of these occur?

- **Object copy**
  - `x` (reference) → "groundhog" (object)
  - `y` → "groundhog"

- **Reference copy**
  - `x` (reference) → "groundhog" (object)
  - `y` → "groundhog"
Object Copy vs. Reference Copy (cont.)

- For
  ```ruby
  x = "groundhog" ; y = x
  ```
  - Ruby and Java would both do a reference copy

- But for
  ```ruby
  x = "groundhog"
  y = String.new(x)
  ```
  - Ruby would cause an object copy
  - Unnecessary in Java since Strings are immutable
Physical vs. Structural Equality

Consider these cases again:

- If we compare \( x \) and \( y \), what is compared?
  - The references, or the contents of the objects they point to?

- If references are compared (physical equality) the first would return false but the second true.
- If objects are compared both would return true.
String Equality

- In Java, `x == y` is physical equality, always
  - Compares references, not string contents
- In Ruby, `x == y` for strings uses structural equality
  - Compares contents, not references
  - `==` is a method that can be overridden in Ruby!
  - To check physical equality, use the `equal?` method
    - Inherited from the `Object` class
- It’s always important to know whether you’re doing a reference or object copy
  - And physical or structural comparison
## Comparing Equality

<table>
<thead>
<tr>
<th>Language</th>
<th>Physical equality</th>
<th>Structural equality</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java</td>
<td>a == b</td>
<td>a.equals(b)</td>
</tr>
<tr>
<td>C</td>
<td>a == b</td>
<td>*a == *b</td>
</tr>
<tr>
<td>Ruby</td>
<td>a.equal?(b)</td>
<td>a == b</td>
</tr>
<tr>
<td>Ocaml</td>
<td>a == b</td>
<td>a = b</td>
</tr>
<tr>
<td>Python</td>
<td>a is b</td>
<td>a == b</td>
</tr>
<tr>
<td>Scheme</td>
<td>(eq? a b)</td>
<td>(equal? a b)</td>
</tr>
<tr>
<td>Visual Basic .NET</td>
<td>a Is b</td>
<td>a = b</td>
</tr>
</tbody>
</table>
Summary

- Scripting languages
- Ruby language
  - Implicit variable declarations
  - Dynamic typing
  - Many control statements
  - Classes & objects
  - Strings