Section 8

### Arrays as a List...

**example definition**

```ruby
a = [5, 6, 7, 8]
```

**zero indexed**

```ruby
a[0]
a[1]
```

**no array bounds error in Ruby**

```ruby
a[10]
```

**get the length..**

```ruby
a.size
```

**negative indexes: count from the end of the array**

```ruby
a[-1]
a[-4]
a[-10]
```

**adding an element beyond the end of the array:**

```ruby
# fills in enough spaces with nil
a[10] = 9
```

**of course, dynamically typed**

```ruby
a[3] = "hello world"
```

**append another array**

```ruby
b = a + [true, false]
```

**Initalizing Arrays**

```ruby
# pick initial size at run-time (x == 20)
x = if a[1] < a[0] then 10 else 20 end
y = Array.new(x)
```

**better: initialized with a block (more on blocks soon)**

```ruby
z = Array.new(x) { 0 }
w = Array.new(x) { |i| i*i }
```

### Arrays as a Tuple..

**given that arrays are so flexible in ruby, no need for tuple types**

```ruby
triple = [false, "hi", a[0] + 4]
```

**get last element**

```ruby
triple[2]
```

### Arrays as a Set..

```ruby
s1 = [1, 2, 3]
s2 = [2, 3, 4, 5]
```

**union (distinct elements only)**

```ruby
c = s1 | s2
```

**intersection (joint elements only)**

```ruby
c = s1 & s2
```

**subtraction**

```ruby
c = s2 - s1
```

### Arrays as a Stack..

```ruby
a = []
a.push 5
a.push 7
a.pop
```

**note: returns nil, not an error**

```ruby
a.pop
```

### Arrays as a Queue..

**-------------------------------

Winter 2018

CSE 341: Programming Languages
# Array Summary

*) Arrays are really a generic mapping from numeric indices to values
*) Arrays in Ruby can be used for many purposes:
 *) Lists
 *) Tuples
 *) Stacks
 *) Queues
 *) Sets
 *) ... consult documentation for even more! 
 *) Very Dynamic 
 *) Many operations allowed that would be errors in other languages, ex:
 *) Accessing / assigning out of bounds
 *) Replacing subsets with subsets of a different length

# Array aliasing
# ==============================================================

# e refers to a new array with the same values as a / d
# (could also add an empty array)
# e = a + []

# evidence

da[0]
a[0] = "new"

e[0]

# note, copy is unchanged

e[0]

# Array slices
# ==============================================================

# slice, starting at element 2 and continuing for 4 elements
f[2, 4]

# can also assign this way, don’t need to use the same number of elements
f[2, 4] = [1, 1]

f

# ==============================================================

# Putts just prints to console.

[1, 2, 5, 12].each { |i| puts (i*i)}
a.any? { |x| x > 7 }
a.all? { |x| x > 7 }

# implicit: are elements "true" (i.e., neither false nor nil)
a.all?

---

Lexical Scope
---

i = 7
[4, 6, 8].each { |x| if i > x then puts (x + 1) end }

---

Hashes in Ruby
---

* similar to arrays, but...
  * ) keys can be anything; strings and symbols common
  * ) unordered, since no natural way to order keys
  * ) different syntax to make them

---

Constructing / Basic Lookups
---

# build an empty hash, could also use Hash.new
h1 = {}

# insert records (note: dynamic typing)
h1["my key"] = "my value"
h1[true] = 36

# get keys / values
h1["my key"]
h1.keys
h1.values

# other ways of constructing hashes:
h2 = {"SML"=>1, "Racket"=>2, "Ruby"=>3}

h2["SML"]

# Symbols are like strings, but faster/more efficient.
# Prefix with ":"#
# Often used with hashes.
h3 = {:sml => 1, :racket => 2, :ruby => 3}

---

Removing records
---

h1.delete("my key")

---

Iteration of key / value pairs
---

puts like println

---

Semicolons are like newlines
---

h2.each { [k, v] print k; print ":"; puts v }

---

Hashes: other notes
---

* ) Methods with many possible arguments will sometimes take a hash instead and perform record lookups using symbols

---

Ranges in Ruby
---

* Similar to an array of contiguous numbers, but more efficiently represented

1..1000000

* just has an upper bound and lower bound

* can be iterated over in much the same way as arrays

* very useful for processing over a sequence of contiguous numbers

* Turning into an array

(1..100).to_a

* Can be used for array slicing

---

Example using inject (ruby's name for reduce)

(1..100).inject { |acc, e| acc + e }

* Can be used for array slicing

---

Similar methods => Duck typing
---

---

Arrays, hashes, and ranges all have some methods others don't

* E.g. keys and values

But also have many of the same methods, particularly iterators

* Great for duck typing

---

Duck typing in m

m [3, 5, 7, 9]
m [3..9]
# Separates "how to iterate" from "what to do"
# Comparing basic types
#
# Good style to:
# use ranges when you can
# use hashes when non-numeric keys better represent data
#
# Exploratory Programming Using Reflection
#
# Can determine the "type" (class) of an object by calling its /class/ method
5.class
[1, 2, 3].class
#
# Can determine an object’s methods by calling its /methods/ method
[1, 2, 3].methods
[1, 2, 3].methods - 5.methods  # throw out "boring" methods all objects have
[1, 2, 3].methods.sort - 5.methods  # sort alphabetically to search easier
#
# Example Uses
#
# *) Useful for exploring the language while in the REPL
# *) Can branch depending on which methods are defined by an object
# *) Exploratory Programming => Write programs by fiddling around in the REPL and trying stuff out