Containers

- In everyday life we often encounter collections
  - course catalog -- collection of course descriptions
  - car lot -- collection of cars
- Mathematicians also work with collections
  - matrix
  - sequence (e.g. 1, 1, 2, 3, 5, 8, ...)
- In Computer Science we make a collection by defining a "data structure" that includes references to other objects
- Using programming terminology an object that holds references to other objects is a container
  - items in a container are objects
  - the container itself is also an object

Containers (cont’d)

- Programming languages have libraries that deal with many different kinds of containers

Containers (cont’d)

- Containers are defined by
  - the relationship between items in the container
  - the operations that can be performed on the container itself
- Example:
  - in a list there is a first item
  - each item except the last has a successor
- Operations on a list include
  - inserting a new item
  - searching for items
  - sorting the items
  - counting the number of items
Arrays

- A container we’ll use in the next lab is called an **array**
  - these containers are actually lists
  - they were called “arrays” in Perl, and the name was adopted by Ruby
  - the distinction is not important in CIS 170, so we’ll just call them “arrays”

Lists grow and shrink, items can be inserted anywhere

Arrays have a fixed size, some locations can be empty

Operations on Arrays

- If we ask Ruby what a is, it shows us it’s an array:
  
  ```ruby
  >> a
  ==> [1, 1, 2, 3, 5, 8, 13, 21]
  ```

- To perform an operation on an array, write the name of the array, a period, and the name of the operation:
  
  ```ruby
  >> a.length
  ==> 8
  ```

Array Objects

- The easiest way to make an array: write a list of objects enclosed in “square brackets”
  
  ```ruby
  >> a = [1, 1, 2, 3, 5, 8, 13, 21]
  ```

- This statement is an assignment just like those we saw previously
  - Ruby allocates space in its “object store”
  - creates an object to represent the array, associates the name a

Array Methods

- The `length` operation and other operations are defined by methods
  - Earlier we saw methods that “stand alone”
    
    ```ruby
    >> include Math
    >> sin(1.0)
    ==> 0.841470984807897
    ```

- Array methods behave like other methods: we can call them, pass them parameters, and they return a result
  
  ```ruby
  >> n = a.length
  ==> 8
  >> n + 2
  ==> 10
  ```
Array Methods (cont’d)

- Here are some examples
  - we’ll be introducing more methods as we need them for projects

```ruby
>> a = [1,1,2,3,5,8,13,21]
=> [1, 1, 2, 3, 5, 8, 13, 21]
>> a.first
=> 1
>> a.last
=> 21
>> a.include?(8)
=> true
```

Array Methods (cont’d)

- Some methods are identified by operators
  - one way to attach a new item to the end of an array is to use the `<<` operator
  - the name of this operator is two `<` symbols, with no space in between

```ruby
>> a << 34
=> [1, 1, 2, 3, 5, 8, 13, 21, 34]
```

“Pop Quiz”

- What do you think Ruby will print as the result of these expressions?

```ruby
>> colors = ['red','green','blue']
=> ['red', 'green', 'blue']
>> colors.length
=> 3
>> colors << 'yellow'
=> ['red', 'green', 'blue', 'yellow']
>> colors.first
=> 'red'
>> colors.last
=> 'yellow'
>> colors << 10
=> ['red', 'green', 'blue', 'yellow', 10]
```

Aside: Printing Values on the Terminal

- Ruby has several methods that can be used to print values
  - `puts` stands for “put string”
  - use it when you want to print the value of a string variable in the terminal window

```ruby
>> s = "hello"
=> "hello"
>> puts s
hello
=> nil
```

- It seems superfluous now -- in IRB all we have to do is type a variable’s name to see its value -- but `puts` will be very useful later
- one way to see what a program is doing is to have it print values periodically as it is running
**Iteration**

- After building a container we often want to do something with each item
- The idea is to “step through” the container to “visit” each object
- This type of operation is called **iteration**
  - from the Latin word *iter*, for “path” or “road”

To find the largest item in an (unsorted) array an algorithm needs to visit each item

```
8 K 9 2 A 2 K
```

**each**

- Ruby uses special methods called *iterators* to visit the items in a collection
- The simplest iterator is named `each`
- A typical call to each looks like this:

```
array.each { |x| ... x ... }
```

- The `array` to iterate over
- A variable name written between two vertical bar symbols
- One or more expressions involving `x`

Ruby selects the items in `array` one by one and stores them in `x`.
After putting an item in `x`, Ruby evaluates the expression(s) in the block.

**each Example**

- This example shows how to use `each` to print every item in an array

```
>> colors.each { |x| puts x }
red
green
blue
```

- There are three strings in `colors`
- After placing a string in `x` (a new variable created for this method call) Ruby evaluates `puts x`
- The effect of calling `puts` is that the value of `x` is displayed on the terminal

```
=> ["red", "green", "blue"]
```

**Mnemonic**

- To remember this block notation think of how mathematicians specify the elements in a set
- One way to define the set of all numbers between 1 and 10:

```
{ x | 1 ≤ x ≤ 10 }
```

- read this as “the set of all `x` such that `x` is greater than or equal to 1 and less than or equal to 10”
- the Ruby notation is similar, it just uses two vertical bars:

```
a.each { |x| ... x ... }
```

- key is usually right above the return key
Another Example

Suppose you want to find the sum of the first 8 Fibonacci numbers:

```ruby
a = [1, 1, 2, 3, 5, 8, 13, 21]
sum = 0
a.each { |n| sum = sum + n }
sum
```

Another Example

Print values to put in a table of countertop areas:

```ruby
load "countertop.rb"
sides = [90, 95, 100, 105, 110, 115]
sides.each { |x| puts countertop(x) }
```

Aside: Interpolation

Ruby has a feature it calls “interpolation”:

```ruby
fib = [1, 1, 2, 3, 5, 8, 13, 21]
n = fib.length
s = "There are #{n} numbers in the list"
```

Another Example of each

A slightly fancier version of the previous example:

```ruby
sides = [100, 105, 110, 115, 120]
sides.each { |x| puts "side: #{x} area: #{countertop(x)}" }
```
Iteration Revisited

- Suppose we want to make a list of numbers between 1 and 100
  - it’s too much work to type an expression like
    ```ruby
    a = [1, 2, 3, …, 100]
    ```
  - (and the chance of making a mistake is pretty high)
- This would be a good place to use iteration
  - start by defining an empty array
  - figure out a Ruby expression that would append a number to the array
  - iterate that step 100 times
- The problem: we can’t use `each`, because we don’t have an array to iterate over

The `times` Method

- It turns out Ruby has an iterator that will solve this problem
- The iterator is named `times`
  - it is a method that can be called for any integer object
- Example:
  ```ruby
  >> n = 3
  => 3
  >> n.times { puts "hello" }
  hello
  hello
  hello
  => 3
  ```

The expression in the body of the block is evaluated `n` times.

Making a List of Numbers

- We can use `times` to make a list of integers from 1 to 100
  ```ruby
  >> a = []
  => []
  >> 100.times { |i| a << i }
  => 100
  >> a
  => [0, 1, 2, 3, 4, 5, 6, …]
  >> a.clear
  => []
  >> 100.times { |i| a << i + 1 }
  => 100
  >> a
  => [1, 2, 3, 4, 5, 6, 7, …]
  ```

Almost what we want -- but this list starts with 0, not 1

Removed all the objects in `a`
Summary

- A **container** is an object that holds references to other objects.
  - In Ruby, an **array** can be used to implement a list, e.g. a list of numbers.
  - Operations on containers are performed by calling methods.

- An **iterator** is a method that tells Ruby to evaluate an expression some number of times.
  - `each`: evaluate the expression once for each object in the container.
  - `times`: evaluate the expression a specified number of times.

- Miscellaneous bits:
  - A method named `puts` will print a string.
  - A technique called “interpolation” helps make strings for fancy output.

Don’t Be Overwhelmed

- There are a lot of new ideas in these slides.
- If you have never done any programming it may be daunting to try to absorb all this.
- But remember:
  - Our goal is for you to learn to read Ruby expressions.
  - You won’t be asked to write any new Ruby programs.
  - Labs will be based on programs that include calls to iterators and other methods.
  - We will also explain what the code does, e.g. “this is how to make a list of numbers from 1 to 100.”
  - You will (eventually) be asked to modify some code, e.g. insert a call to `puts` so you can see what is happening as the program runs.

Extra Credit Challenge

- Can you figure out how to make a list of Fibonacci numbers?
- Start by defining a list with the first two numbers:
  
  ```ruby
  >> f = [1, 1]
  => [1, 1]
  ```
- What can you put in the body of this block to have Ruby generate the next 10 numbers in the sequence?
  
  ```ruby
  >> 10.times { ...... }
  ```
  
  **Hint:** If `a` is a list, the expression `a[-1]` returns the item at the end of the list, and `a[-2]` returns the 2nd-to-last item...