The Plan

- Ruby container data structures
- Blocks and control structures (iterators, etc.)
- Blocks and first-class closures

Later:
  - Duck typing
  - Inheritance
  - Modules and mixins
Containers in Ruby

• Like most scripting languages, Ruby provides very general container classes
• Two major kinds
  – Arrays: ordered by position
  – Hashes: collections of <key, value> pairs
    • Often known as associative arrays, maps, or dictionaries
    • Unordered
Ruby Arrays

- Instances of class **Array**
- Create with an array literal, or **Array.new**

```ruby
words = [ "how","now","brown","cow" ]
stuff = [ "thing", 413, nil ]
seq = Array.new
```

- Indexed with [ ] operator, 0-origin; negative indices count from right

```ruby
words[0]  stuff[2]  words[-2]
seq[1] = "something"
```
Ruby Hashes

• Instances of class `Hash`
• Create with an hash literal, or `Hash.new`
  ```ruby
  pets = { "spot" => "dog", "puff" => "cat" }
  tbl = Hash.new
  ```
• Indexed with `[ ]` operator
  ```ruby
  pets["puff"]     pets["fido"]
  pets["cheeta"] = "monkey"
  ```
  – Can use almost anything as key type; can use anything as element type
Containers and Iterators

• All containers respond to the message “each”, executing a block of code for each item in the container

```ruby
words.each { puts "another word" }
words.each { | w | puts w }
```
Blocks

• A block is a sequence of statements surrounded by `{ … }` or `do … end`
• Blocks must appear immediately following the method call that executes them, on the same line
• Blocks may have 1 or more parameters at the beginning surrounded by `| … |`
  – Initialized by the method that runs (executes, “calls”) the block
Blocks as Closures

• Blocks can access variables in surrounding scopes
  
  wordlist = ""
  words.each { |w| wordlist = wordlist +
                w + " " }

  – These are almost, but not quite, first-class closures (some differences in scope rules compared to Racket)
More Block Uses

• Besides iterating through containers, blocks are used in many other contexts

```ruby
3.times { puts "hello" }

n = 0
100.times { |k| n += k }
puts "sum of 0 + ... + 99 is " + n
```
Block Execution

- Any method call can be followed by a block. The block is executed by the method – when depends on the method
- A block is executed in the context of the method call
  - Block has access to variables at the call location
  - Return in a block returns from surrounding method(!)

```ruby
def search(x, words)
    words.each { |w| if x==w then return end }
    puts "not found"
end
```
yield

- Any method call can be followed by a trailing block. A method “calls” the block with a `yield` statement.

```ruby
def repeat
    yield
    yield
end
repeat { puts "hello" }
```

Output:
```
hello
hello
```
yield with arguments

• If the block has parameters, use expressions with yield to pass arguments

```python
def xvii
    yield 17
end
xvii { | n | puts n+1 }
```

– This is exactly how an iterator works
Blocks are “second-class”

• Blocks (and methods) are not objects in Ruby – i.e., not things that can be passed around as first-class values
• All a method can do with a block is \texttt{yield} to it (i.e., call it)
  – Can’t return it, store it in an object, etc.
  – But can also turn blocks into real closures (next slide)
First-class closures

- Implicit block arguments and yield are often sufficient
- But when you want a closure that can be returned, stored, passed as an argument:
  - The built-in `Proc` class
  - `Lambda` method of `Object` takes a block and makes a `Proc`
  - Instances of `Proc` have a `call` method that can be used to execute them
Creating Procs: examples

• Create a Proc object explicitly

```ruby
p = Proc.new { |x, y| x+y }
...
p.call(x,y)
```

• Use Object’s lambda method

```ruby
is_positive = lambda { |x| x > 0 }
```
Procs vs. Lambdas

• A Proc is a block wrapped in an object – and behaves just like a block
  – In particular, a return in a Proc will return from the *surrounding* method where the Proc’s closure was created
    • Error if that method has already terminated
• A Lambda is more like a method
  – Return just exits from the lambda