CSci 4223
Principles of Programming Languages

Lecture 16
Prof. Clarkson
Spring 2013

Review
• Ruby: classes, methods, variables
• Today:
  – Data structures: arrays, hashes
  – Closures: blocks, procs

Arrays and Hashes

Ruby Arrays
• Lots of special syntax and many provided methods for the Array class
• Can hold any number of other objects, indexed by number
  – Get via a[i]
  – Set via a[i] = e
• Compared to arrays in many other languages
  – More flexible and dynamic
  – Fewer operations result in errors
  – Less efficient
• "The standard collection" (like lists were in ML)

Using Arrays
• See many examples, some demonstrated here
• Consult the documentation/tutorials
  – If seems sensible and general, probably a method for it
• Arrays make good tuples, lists, stacks, queues, sets, …
• Iterating over arrays typically done with methods taking blocks, not with loops

More collections
• Hashes like arrays but:
  – Keys can be anything: strings and symbols common
  – No natural ordering like numeric indices
  – Different syntax to make them
  Like a dynamic record with anything for field names
  – Sometimes pass a hash rather than many arguments
• Ranges like arrays of contiguous numbers but:
  – More efficiently represented, so large ranges fine

Good style to:
• Use ranges when you can
• Use hashes when non-numeric keys better represent data
Similar methods

- Arrays, hashes, and ranges: each has some methods others don’t
  - E.g., hashes have keys and values
  - E.g., arrays have push and pop
- But also have many of the same methods, particularly iterators
  - Great for duck typing
  - Example:

```ruby
def foo a
  a.count {|x| x*x < 50}
end
foo [3,5,7,9]
foo (3..9)
```

Once again separating “how to iterate” from “what to do”

Blocks

- Blocks are the feature many programmers find strangest compared to other PLs
- ...but many programmers don’t study principles of PLs!
- Blocks are almost just closures
  - Normal: easy way to pass anonymous functions to methods for all the usual reasons
  - Normal: Blocks can take 0 or more arguments
  - Normal: Blocks use lexical scope: block body uses environment where block was defined

Examples:

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

Some strange things

- Can pass 0 or 1 block with any message
  - Callee might ignore it
  - Callee might give an error if you do not send one
  - Callee might do different things if you do/don’t send one
    - Also number-of-block-arguments can matter
- Just put the block “next to” the “other” arguments (if any)
  - Syntax: {e}, {||x| e}, {||x,y| e}, etc. (plus variations)
    - Can also replace { and } with do and end
      - Often preferred for blocks > 1 line

Blocks everywhere

- Standard library makes excellent use of iterators (block-taking methods)
- Ruby has loops but very rarely used
  - Can write {0..1}.each {||j| e}, but often better options
- Examples (consult documentation for many more)

```ruby
a = Array.new(5) {||i| 4*(i+1)}
a.each {||x| puts "hi" }
a.each {||x| puts (x + 2) }
a.map {||x| x * x } #synonym: collect
a.any? {||x| x > 7 }
a.all? {||x| x > 7 }
```

More strangeness

- Callee does not give a name to the (potential) block argument
- Instead, just calls it with yield or yield(args)
  - Silly example:

```ruby
def silly a
  (yield a) + (yield 42)
end
x.silly(5) ||b| b**2 }
```

- See code for slightly less silly example
- Can ask block_given? but often just assume a block is given or that a block’s presence is implied by other arguments
Blocks are “second-class”

All a method can do with a block is `yield` to it:
- Cannot return it, cannot store it in a variable, etc.
- But can turn blocks into real closures
- Closures are instances of class `Proc`  
  - Called with method `call`
- Blocks are “second-class”
- Procs are “first-class expressions”

This is Ruby, so there are several ways to make `Proc` objects:
- One way: method `lambda` of `Object` takes a block and returns the corresponding `Proc`

Example

```
a = [3, 5, 7, 9]
```
- Blocks are fine for applying to array elements
```
b = a.map {|x| x+1 }
i = b.count {|x| x>=6 }
```
- But for an array of closures, need `Proc` objects
  - More common use is callbacks
```
c = a.map {|x| lambda {|y| x>y}}
c[2].call 17  
j = c.count {|x| x.call(5) }
```

Moral

- First-class (“can be passed/stored anywhere”) makes closures more powerful than blocks
- But blocks are (a little) more convenient and cover most uses
- This helps us understand what first-class means
- Language design question: When is convenience worth making something less general and powerful?