This lecture

Three mostly separate topics

• Flexible arrays, ranges, and hashes [actually covered in section]
• Ruby's approach to almost-closures (blocks) and closures (Procs)
  – [partially discussed in section as well]
  – Convenient to use; unusual approach
  – Used throughout large standard library
    • Explicit loops rare
    • Instead of a loop, go find a useful iterator
• Subclasses, inheritance, and overriding
  – The essence of OOP, now in a more dynamic language

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 are errors
  – Less efficient
• “The standard collection” (like lists were in ML and Racket)

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
  – Next topic…

Blocks

Blocks are probably Ruby's strangest feature compared to other PLs

But 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:

```
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

- Rampant use of great block-taking methods in standard library
- Ruby has loops but very rarely used
  - Can write `(0..i).each { |j| e}`, but often better options
- Examples (consult documentation for many more)

```
a = Array.new(5) { |i| 4*(i+1)
a .each { puts "hi" }
.a .each { |x| puts (x * 2) }
a .map { |x| x * 2 } # synonym: collect
.a .any? { |x| x > ? }
a .all? { |x| x > ? }
a .inject(0) { |acc, elt| acc+elt }
a .select { |x| x > ? } # non-synonym: filter
```

More strangeness

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

```
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, store it in an object (e.g., for a callback), …
- But can also turn blocks into real closures
- Closures are instances of class `Proc`
  - Called with method `call`

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]
b = a .map { |x| x+1 }
i = b .count { |x| x >= 6 }
```

- Blocks are fine for applying to array elements
- 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?

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
  - Often 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 all have some methods other don’t
  - E.g., keys and values
- 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”

Next major topic

- Subclasses, inheritance, and overriding
  - The essence of OOP
  - Not unlike you have seen in Java, but worth studying from PL perspective and in a more dynamic language

Subclassing

- A class definition has a superclass (Object if not specified)
  ```ruby
  class ColorPoint < Point ...
  ```
- The superclass affects the class definition:
  - Class inherits all method definitions from superclass
  - But class can override method definitions as desired
- Unlike Java/C#/C++:
  - No such thing as “inheriting fields” since all objects create instance variables by assigning to them
  - Subclassing has nothing to do with a (non-existent) type system: can still (try to) call any method on any object

An object has a class

```ruby
p = Point.new(0,0)
cp = ColorPoint.new(0,0,"red")
p.class  # Point
p.class.superclass  # Object
cp.class  # ColorPoint
cp.class.superclass  # Point
cp.class.superclass.superclass  # Object
cp.is_a? Point  # true
cp.instance_of? Point  # false
cp.is_a? ColorPoint  # true
cp.instance_of? ColorPoint  # true
```

- Using these methods is usually non-OOP style
  - Disallows other things that “act like a duck”
  - Nonetheless semantics is that an instance of ColorPoint “is a” Point but is not an “instance of” Point
  - [Java note: instanceof is like Ruby’s is_a?]

Example (to be continued)

```ruby
class ColorPoint < Point
  attr_accessor :color
  def initialize(x,y,c)
    super(x,y)
    @color = c
  end
  def distFromOrigin
    Math.sqrt(x*x + y*y)
  end
  def distFromOrigin2
    Math.sqrt(x*x + y*y)
  end
end
```
Why subclass

• Instead of creating `ColorPoint`, could add methods to `Point` – That could mess up other users and subclassers of `Point`

```ruby
class Point
  attr_accessor :color
  def initialize(x,y,c="clear")
    @x = x
    @y = y
    @color = c
  end
end
```

• Instead of subclassing `Point`, could copy/paste the methods – Means the same thing if you don't use methods like `is_a?` and `superclass`, but of course code reuse is nice

```ruby
class ColorPoint
  attr_accessor :x, :y, :color
  def initialize(x,y,c="clear")
    @pt = Point.new(x,y)
    @color = c
  end
  def x
    @pt.x
  end
  # similar "forwarding" methods
  end
end
```

Overriding

• `ThreeDPoint` is more interesting than `ColorPoint` because it overrides `distFromOrigin` and `distFromOrigin2` – Gets code reuse, but highly disputable if it is appropriate to say a `ThreeDPoint` "is a` Point" – Still just avoiding copy/paste

```ruby
class ColorPoint
  attr_accessor :color
  def initialize(x,y,c="clear")
    @pt = Point.new(x,y)
    @color = c
  end
  def x
    @pt.x
  end
  # similar "forwarding" methods
  end
end
```

So far…

• With examples so far, objects are not so different from closures – Multiple methods rather than just "call me" – Explicit instance variables rather than environment where function is defined – Inheritance avoids helper functions or code copying – "Simple" overriding just replaces methods

• But there is one big difference:

  **Overriding can make a method defined in the superclass call a method in the subclass**

  – The essential difference of OOP, studied carefully next lecture

```ruby
class PolarPoint < Point
  def initialize(r,theta)
    @r = r
    @theta = theta
  end
  def x
    @r * Math.cos(@theta)
  end
end
```

Example: Equivalent except constructor

• Also need to define `x=` and `y=` (see code file)

• Key punchline:
  `distFromOrigin2`, defined in `Point", already works"

```ruby
def distFromOrigin2
  Math.sqrt(x*x+y*y)
end
```

– Why: calls to `self` are resolved in terms of the object's class