This lecture

Two separate topics

- Ruby’s approach to almost-closures (blocks) and closures (Procs)
  - 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
  - Not unlike in Java, but worth studying from PL perspective and in a more dynamic language

Blocks

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

- Normal: easy way to pass anonymous functions for all the reasons we have been studying
- Normal: Blocks can take 0 or more arguments
- Strange: Can send 0 or 1 block with *any* message send
- Strange: Callee does not have a name for the block
  - Calls it with `yield`, `yield 42`, `yield (3,5)`, etc.
  - Can ask `block_given?` but rarely used in practice
    (usually assume a block is given if expected, or that a block’s presence is implied by other arguments)

Examples

- Rampant use of blocks in standard library
  - Classes define iterators; don’t write your own loops
  - Most of these examples happen to have 0 "regular" arguments

```
3.times { puts "hi" }
[4,6,8].each { puts "hi" }
[4,6,8].each { |x| puts x * 2 }
[4,6,8].map { |x| x * 2 }
[4,6,8].any? { |x| x > 7 } # block optional
[4,6,8].inject(foo) {|acc,elt| … }
```

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

Blocks are “second-class”

All a method can do with a block is `yield` to it (i.e., call it)

- Can’t return it, store it in an object (e.g., for a callback), etc.
- But can also turn blocks into real closures (next slide)

But one block can call another block via `yield`

- From example `MyList` class in `lec20.rb` (though better in Ruby to use arrays as lists than define your own)

```
def map
  if @tail.nil?
    MyList.new(@head, nil)
  else
    MyList.new(@head, @tail.map {|x| yield x})
  end
end
```

First-class closures

- Implicit block arguments and `yield` is often sufficient
- But when you want a closure you can return, store, etc.:
  - The built-in `Proc` class
  - `lambda` method of `Object` takes a block and makes a `Proc`
    - Also can do it with "& arg", not shown here
    - Instances of `Proc` have a method `call`

```
def map_p proc
  if @tail.nil?
    MyList.new(proc.call(@head), nil)
  else
    MyList.new(proc.call(@head), @tail.map proc)
  end
end
```

```
def map_p proc
  if @tail.nil?
    MyList.new(proc.call(@head), nil)
  else
    MyList.new(proc.call(@head), @tail.map proc)
  end
end
xs.map_p lambda{|x| x}
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:
  - 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 pass any object to any method

An object has a class

```ruby
p = Point.new(0,0)  # Point
cp = ColorPoint.new(0,0,"red")  # ColorPoint
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 not an "instance of" Point
  - Java's instanceof is like Ruby's is_a?

Why subclass

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

```ruby
class ColorPoint < Point
  attr_reader :color
  attr_writer :color
  def initialize(x,y,c="clear")
    @x = x
    @y = y
    @color = c
  end
  def distFromOrigin
    # direct field access
    Math.sqrt(@x*@x + @y*@y)
  end
  def distFromOrigin2
    # use getters
    Math.sqrt(x*x + y*y)
  end
end
```

- Instead of subclassing Point, could use a Point instance variable
  - 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_reader :x, :y, :color
  attr_writer :x, :y, :color
  def initialize(x,y,c="clear")
    @pt = Point.new(x,y)
    @color = c
  end
  def x
    @pt.x
  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 ThreeDPoint < Point
  def initialize(x, y, z)
    super(x, y)
    @z = z
  end
  def distFromOrigin # distFromOrigin2 similar
    d = super + @z**2
    Math.sqrt(d*d + @z*@z)
  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 whatever is environment where function is defined
  - "Simple" overriding just replaces methods

- But there is a big difference (that you learned in Java):
  
  *Overriding can make a method define in the superclass call a method in the subclass*

  - The essential difference of OOP, studied carefully next lecture

**Example: Equivalent except constructor**

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

- Also need to define `x=` and `y=`
  (see lec20.rb)

- 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