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 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

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

- Easy to write your own methods that use blocks

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

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 (shown in block Proc.rb)  

```ruby
def map_p proc  
if @tail.nil?  
MyList.new(yield(@head), nil)  
extelse  
MyList.new(yield(@head), @tail.map { |x| yield x})  
end  
end
```
**Subclassing**

- A class definition has a **superclass** (Object if not specified)
  ```ruby
  class ColorPoint < Point ...
  end
  ```
- 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)
cp = ColorPoint.new(0,0,"red")
p.class # Point
cp.class # ColorPoint
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

```
class ThreeDPoint < Point
  def initialize(x, y, z)
    @x = x
    end
  def distFromOrigin # distFromOrigin2 similar
    d = super
    @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
  - Inheritance avoids helper functions or code copying
  - "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

```
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 blocks_inheritance.rb)
- Key punchline:
  - distFromOrigin2, defined in Point, "already works"
  - def distFromOrigin2
    - Math.sqrt(x*x+y*y)
- Why: calls to self are resolved in terms of the object's class