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

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 blocks.rb (though better
  in Ruby to use arrays as lists than define your own)

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)
  - Instances of Proc have a method call

---

Autumn 2012
### 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

### Why subclass

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

```ruby
class ColorPoint
  attr_reader :color
  attr_writer :color
  def initialize(x, y, c = "clear")
    @color = c
  end
  def distFromOrigin
    Math.sqrt(x*x + y*y)
  end
  def distFromOrigin2
    Math.sqrt(x*x + y*y)
  end
end
```

- Instead of subclassing ColorPoint, could use a Point instance variable
  - Define methods to send same message to the Point

```ruby
class ColorPoint
  attr_reader :color
  attr_writer :color
  def initialize(x, y, c = "clear")
    $pt = Point.new(x, y)
    $color = c
  end
  def x
    $pt.x
  end
end
```

### An object has a class

```ruby
p = Point.new(0,0)
cp = ColorPoint.new(0,0,"red")
p.class
p.class.superclass
cp.class
cp.class.superclass
cp.is_a? Point
cp.is_a? ColorPoint
```

- 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's `instanceof` is like Ruby’s `is_a?`

### Why subclass

- 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_reader :x, :y, :color
  attr_writer :x, :y, :color
  def initialize(x, y, c = "clear")
    ...
  end
  def x
    @pt.x
  end
end
```

- Instead of subclassing Point, could use a Point instance variable
  - Define methods to send same message to the Point
  - Often OOP programmers overuse subclassing (often composition is better)
  - But for ColorPoint, subclassing makes sense: less work and can use a ColorPoint wherever code expects a Point

```ruby
class ColorPoint
  attr_reader :color
  attr_writer :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
    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
  - 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 (study 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 blocks_inheritance.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