# Lecture 20 Arrays and Such, Blocks and Procs, Inheritance and Overriding

#### Arrays (code you can type/paste into irb)

```ruby
a = [3, 2, 7, 9]
a[2]
a[0]
a[4]
a.size
a[-1]
a[-2]
a[1] = 6
a
a[6] = 14
a[5].size
a[3] = "hi"

b = a + [true, false]
c = [3, 2, 3] | [1, 2, 3]

triple = [false, "hi", a[0] + 4]

x = if a[1] < a[0] then 10 else 20 end
y = Array.new(x)

z = Array.new(x) { 0 }
w = Array.new(x) { |i| −i }
```

#### Blocks (code you can type/paste into irb)

```ruby
3.times {
  puts "hello"
}

[4, 6, 8].each { puts "hi" }
l = 7

[4, 6, 8].each { |x| if i > x then puts (x+1) end }

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 > 7 }
a.all? { |x| x > 7 }
a.all? # implicit are elements "true" (i.e., neither false nor nil)
a.select { |x, acc, elt| acc + elt }
```

#### Example code using yield (taking a block)

```ruby
class Foo
  def initialize(max)
    @max = max
  end

  def silly
    yield(4, 5) + yield(@max, @max)
  end

  def count base
    if base > @max
      raise "cached max"
    elsif yield base
      1 + (count(base+1) { |i| yield i })
    else
      1 + (count(base+1) { |i| yield i })
    end
  end
end

foo = Foo.new(1000)
foo.silly { |a, b| 2*a - b }
foo.count(10) { |i| (i * i) == (34 * i) }
```

#### Procs (code you can type/paste into irb)

```ruby
f = [2, 4, 6, 8, 10, 12, 14]
f[2, 4]
f.slice(2, 2)
f.slice(-2, 2)
f[2, 4] = [1, 1]
```
# elements of c are Proc objects with a call method

c[2].call 17
j = c.count {|x| x.call(5) }

##### Hashes and Ranges (cod you can type/paste into irb)

h1 = {}
h1["a"] = "Found A"
h1[false] = "Found false"
h1["a"]
h1[false]
h1[42]
h1.keys
h1.values
h1.delete("a")

h2 = {"SML"=>1, "Racket"=>2, "Ruby"=>3}

# Symbols are like strings, but cheaper. Often used with hashes.

h3 = {sml: 1, racket: 2, ruby: 3}

# each for hashes best with 2-argument block

h2.each {|k,v|
  print k; print ":"; puts v}

# ranges

(1..100).inject {|acc,elt| acc + elt}

def m a
  a.count {|x| x*x < 50}
end

# duck typing in m

m [3,5,7,9]
m (3..9)

##### Subclasses

class Point
  attr_accessor :x, :y
  def initialize(x,y)
    @x = x
    @y = y
  end
  def distFromOrigin
    d = super
    Math.sqrt(d * d + @z * @z)
  end
end

class ColorPoint < Point
  attr_accessor :color
  def initialize(x,y,c="clear") |
    # super (x,y) # keyword super calls same method in superclass
    @color = c
  end
end

# example uses with reflection

p = Point.new(0,0)
p

pp = PolarPoint.new(4,Math::PI/4)
pp

pp.distFromOrigin2

##### Subclasses with Overriding

# design question: "Is a 3D−point a 2D−point?"

# [arguably poor style here, especially in statically typed OOP languages]

class ThreeDPoint < Point
  attr_accessor :z
  def initialize(x,y,z)
    super
    @z = z
  end
  def distFromOrigin
    d = super
    Math.sqrt(d * d + @z * @z)
  end
  end

class PolarPoint < Point
  # Interesting: by not calling super constructor, no x and y instance vars
  # In Java/C#/Smalltalk would just have unused x and y fields
  def initialize(r,theta)
    @r = r
    @theta = theta
  end
  def x
    @r * Math.cos(@theta)
  end
  def y
    @r * Math.sin(@theta)
  end
  def x= a
    b = y
    @theta = Math.atan2(b,a)
    @r = Math.sqrt(a*a + b*b)
    self
  end
  def y= b
    a = x
    @theta = Math.atan2(b,a)
    @r = Math.sqrt(a*a + b*b)
    self
  end
end

def distFromOrigin # must override since inherited method does wrong thing
  @r
end

# the key example

pp = PolarPoint.new(4,Math::PI/4)
pp.x
pp.y
pp.distFromOrigin2