# lec 20 code in class

#### 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[6] = 14
a[5]
a[3] = "hi"
```

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

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

```ruby
# arrays can also have initial size chosen at run-time
# (and as we saw can grow later -- and shrink)
x = if a[1] < a[0] then 10 else 20 end
y = Array.new(x)
```

```ruby
# better: initialized with a block (coming soon)
z = Array.new(x) { () }
w = Array.new(x) { |i| -i }
```

```ruby
# stacks
a
a.push 5
a.pop
a.push 7
a.pop
a.pop
```

```ruby
# queues (from either end)
a.push 11
a.shift
a.shift
a.unshift 14
```

```ruby
# aliasing
d = a
e = a + []
d[0] = 6
d[0]
e[0]
```

```ruby
# slices
f = [2, 4, 6, 8, 10, 12, 14]
f[2, 4]
f.slice(2, 2)
f.slice(-2, 2)
f[2, 4] = [1, 1]
```

```ruby
[1, 3, 4, 12].each { |i| puts (i * i) }
```

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

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

```ruby
{4, 6, 8}.each { puts "hi" }
```

```ruby
i = 7
{4, 6, 8}.each { |x| if x > i then puts (x+1) end }
```

```ruby
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
```

```ruby
a.any? { |x| x > 7 }
a.all? { |x| x > 7 }
a.all? # implicit are elements "true" (i.e., neither false nor nil)
a.inject(0) { |acc, elt| acc + elt }
a.select { |x| x > 7 } # non-synonym: filter
```

```ruby
def t i
  (0..i).each do |j|
    print " * j
  end
  print 
end
t 9
```

#### 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 "reached max"
    elsif yield base
      1
    else
      1 + (count(base+1) { |i| yield i})
    end
  end
end
```

```ruby
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
a = [3, 5, 7, 9]
```

```ruby
# no need for Procs here
b = a.map { |x| x + 1 }
i = b.count { |x| x >= 6 }
```

```ruby
# need Procs here: want an array of functions
```

```ruby
c = a.map { |x| lambda { |y| x >= y } }
```
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.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 ":
 ranges
(1..100).inject (|acc,elt| acc + elt)
def m a
  a.count {|x| x*x < 50}
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
    Math.sqrt(@x * @x + @y * @y) # why a module method? Less OOP :-(
  end
  def distFromOrigin2
    Math.sqrt(x * x + y * y)
  end
end
class ColorPoint < Point
  attr_accessor :color
  def initialize(x,y,c="clear") # or could skip this and color starts unset
    super(x,y) # keyword super calls same method in superclass
    @color = c
  end
end

# example uses with reflection
p = Point.new(0,0)
cp = ColorPoint.new(0,0,"red")
p.class
# Point
# The key example
pp = PolarPoint.new(4,Math::PI/4)
pp.x
pp.y
pp.distFromOrigin2