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

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

```ruby
b = a + [true, false]
```

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

```ruby
# stacks
a
a.push 5
puts a

# aliasing

d = a

e = a + []
n = a[0] = 6
```

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

```ruby
# arrays make fine tuples
triple = [false, "hi", a[0] + 4]
triple[2]
```

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

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

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

```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
```

```ruby
# better: initialized with a block (coming soon)
z = Array.new(x) { 0 }
w = Array.new(x) {|i| −i }
```

```ruby
# stacks
a
a.push 5
puts a

# aliasing

d = a

e = a + []
n = a[0] = 6
```

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

### 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 }
```

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

### Subclasses

class Point
  attr_accessor :x, :y
  def initialize(x,y)
    @x = x
    @y = y
  end
  def distFromOrigin
    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)
pp = ColorPoint.new(0,0, "red")

### 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(x,y)
    @z = z
  end
  def distFromOrigin
    d = super
    Math.sqrt(d * d + @z * @z)
  end
  def distFromOrigin2
    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
    # avoids multiple calls to y method
    @theta = Math.atan2(b,a)
    @r = Math.sqrt(a*a + b*b)
    self
  end
  def y= b
    a = x
    # avoid multiple calls to y method
    @theta = Math.atan2(b,a)
    @r = Math.sqrt(a*a + b*b)
    self
  end
  def distFromOrigin
    # must override since inherited method does wrong thing
    @r
  end
  def distFromOrigin2
    # inherited distFromOrigin2 already works!!
  end
end

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