Ruby 2

OPL 2/66

Outline

- Array, range and hashes
- Blocks, yield
- Subclasses

Array

- Arrays are ordered, integer-indexed collections of objects
- Array indexing starts at 0
- A new array can be created by using the literal constructor [] or Array's constructor

```
ary = [1, "two", 3.0] #=> [1, "two", 3.0]
```

```
ary = Array.new    #=> []
Array.new(3)       #=> [nil, nil, nil]
```


Array indexing

- Elements in an array can be retrieved using the Array#[] method.
- It can take
  - a single integer argument (a numeric index),
  - a pair of arguments (start and length)
  - a range
  - index starts from 0

```
arr = [1, 2, 3, 4, 5, 6]
arr[2]    #=> 3
arr[100]  #=> nil
arr[-3]   #=> 4
arr[2, 3] #=> [3, 4, 5]
arr[1..4] #=> [2, 3, 4, 5]
arr[1..-3] #=> [2, 3, 4]
```

- Negative indices start counting from the end, with -1 being the last element.

Useful methods

```
browsers = ['Chrome', 'Firefox', 'Safari', 'Opera', 'IE']
browsers.length #=> 5
browsers.count #=> 5
```

```
browsers.empty? #=> false
```

```
arr = [1, 2, 3, 4]
arr.push(5) #=> [1, 2, 3, 4, 5]
arr << 6    #=> [1, 2, 3, 4, 5, 6]
```


Range

- A Range represents an interval—a set of values with a beginning and an end.
- Ranges may be constructed using the *s*..*e* and *s*...*e* literals, or with Range::new
- Ranges constructed using .. run from the beginning to the end inclusively
- Those created using ... exclude the end value.

```
(-1..-5).to_a      #=> []
(-5..-1).to_a      #=> [-5, -4, -3, -2, -1]
('a'..'e').to_a    #=> ["a", "b", "c", "d", "e"]
('a'...'e').to_a   #=> ["a", "b", "c", "d"]
```


Hash

- A Hash is a dictionary-like collection of unique keys and their values.
- can be easily created by using its implicit form:

```
grades = { "Jane Doe" => 10, "Jim Doe" => 6 }
```

- Or using ::new method

```
grades = Hash.new
grades["Dorothy Doe"] = 9
```


Symbol

- In Ruby, symbols are **immutable names primarily used as hash keys or for referencing method names**.
- #Symbols must be valid Ruby variable names and always start with a colon (:)
  - Symbols are a good use for hash keys because they are immutable.
  - You can't change a symbol once its defined.
  - Only one copy of a symbol can exist at one time, so they do not consume a lot of memory.
  - These two reasons makes symbols as keys faster than strings as keys.

```
my_hash = {
   :one => "One",
   :a_symbol => 42,
   :boom => true
}

puts my_hash

# {:one=>"One", :a_symbol=>42, :boom=>true}
```

Block

- A block, essentially, is the same thing as a method, except it does not have a name
- blocks can only be created by the way of passing them to a method when the method is called.

```
1.  # Form 1: recommended for single line blocks
2.  [1, 2, 3].each { |num| puts num }
3.                   ^^^^^ ^^^^^^^^
4.                   block   block
5.                 arguments body
```

```ruby
5.times do
  puts "Oh, hello from inside a block!"
end
```


Block

- Blocks can be defined enclosing code in do and end, or curly braces {}.

```ruby
5.times do
  puts "Oh, hello!"
end

5.times { puts "hello!" }
```

- Block is essentially Ruby's way to perform iteration

Iterating over array

- Array has an each method, which defines what elements should be iterated over and how.
- In case of Array's each, all elements in the Array instance are yielded to the supplied block in sequence.

```ruby
arr = [1, 2, 3, 4, 5]
arr.each {|a| print a -= 10, " "}
# prints: -9 -8 -7 -6 -5
#=> [1, 2, 3, 4, 5]
```


Map method on array

- The map method can be used to create a new array based on the original array

```ruby
arr.map {|a| 2*a}     #=> [2, 4, 6, 8, 10]
arr                   #=> [1, 2, 3, 4, 5]
arr.map! {|a| a**2}   #=> [1, 4, 9, 16, 25]
arr                   #=> [1, 4, 9, 16, 25]
```

Selecting Items from an Array

- Non-destructive Selection

```
arr = [1, 2, 3, 4, 5, 6]
arr.select {|a| a > 3}       #=> [4, 5, 6]
arr.reject {|a| a < 3}       #=> [3, 4, 5, 6]
arr.drop_while {|a| a < 4}   #=> [4, 5, 6]
arr                          #=> [1, 2, 3, 4, 5, 6]
```

- Destructive Selection

```
arr.delete_if {|a| a < 4}   #=> [4, 5, 6]
arr                         #=> [4, 5, 6]

arr = [1, 2, 3, 4, 5, 6]
arr.keep_if {|a| a < 4}   #=> [1, 2, 3]
arr                       #=> [1, 2, 3]
```


yield

- yield is a **Ruby keyword** that calls a block when you use it.
- When you use the yield keyword, **the code inside the block will run** & do its work

```
1.  def print_once
2.    yield
3.  end
4.
5.  print_once { puts "Block is being run" }
```

```
1.  def print_twice
2.    yield
3.    yield
4.  end
5.
6.  print_twice { puts "Hello" }
7.
8.  # "Hello"
9.  # "Hello"
```


yield

- You can pass any number of arguments to yield

```
1.  def one_two_three
2.    yield 1
3.    yield 2
4.    yield 3
5.  end
6.
7.  one_two_three { |number| puts number * 10 }
8.  # 10, 20, 30
```


Subclassing

- A class definition has a *superclass* (**`Object`** if not specified)

```
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/C#/C++:
  - No such thing as “inheriting fields” since all objects create instance variables by assigning to them

Example (to be continued)

```
class Point
  attr_accessor  :x, :y
  def initialize(x,y)
    @x = x
    @y = y
  end
  def distFromOrigin
    # direct field access
    Math.sqrt(@x*@x + @y*@y)
  end
  def distFromOrigin2
    # use getters
    Math.sqrt(x*x + y*y)
  end
end
```

```
class ColorPoint < Point
  attr_accessor  :color
  def initialize(x,y,c)
    super(x,y)
    @color = c
  end
end
```


An object has a class

- 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

```
p  = Point.new(0,0)
cp = ColorPoint.new(0,0,"red")
p.class                          # Point
p.class.superclass               # Object
cp.class                         # ColorPoint
cp.class.superclass              # Point
cp.class.superclass.superclass   # Object
cp.is_a? Point                   # true
cp.instance_of? Point            # false
cp.is_a? ColorPoint              # true
cp.instance_of? ColorPoint       # true
```


Example continued

- Consider alternatives to:

```
class ColorPoint < Point
  attr_accessor  :color
  def initialize(x,y,c)
    super(x,y)
    @color = c
  end
end
```

- Here subclassing is a good choice, but programmers often overuse subclassing in OOP languages

Why subclass

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

```
class Point
  attr_accessor :color
  def initialize(x,y,c="clear")
    @x = x
    @y = y
    @color = c
  end
end
```

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

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


Why subclass

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

```
class ColorPoint
  attr_accessor :color
  def initialize(x,y,c="clear")
    @pt = Point.new(x,y)
    @color = c
  end
  def x
    @pt.x
  end
  … # similar "forwarding" methods
    # for y, x=, y=
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

```
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 environment where function is defined
  - Inheritance avoids helper functions or code copying
  - "Simple" overriding just replaces methods
- But there is one big difference:
- Overriding can make a method defined 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 code file)
- Key punchline: distFromOrigin2, defined in Point, "already works"

```
def distFromOrigin2
  Math.sqrt(x*x+y*y)
end
```

  - Why: calls to self are resolved in terms of the object's class