<section-header><section-header><section-header><section-header><section-header><section-header><section-header><text></text></section-header></section-header></section-header></section-header></section-header></section-header></section-header>	<section-header><section-header><section-header><section-header><list-item><list-item><list-item><list-item><list-item><list-item><list-item></list-item></list-item></list-item></list-item></list-item></list-item></list-item></section-header></section-header></section-header></section-header>
 Ruby Arrays Lots of special syntax and many provided methods for the Array class Can hold any number of other objects, <i>indexed</i> by number Get via a[i] Set via a[i] = e Compared to arrays in many other languages More flexible and dynamic Fewer operations are errors Less efficient "The standard collection" (like lists were in ML and Racket) 	 Using Arrays See many examples, some demonstrated here Consult the documentation/tutorials If seems sensible and general, probably a method for it Arrays make good tuples, lists, stacks, queues, sets, Iterating over arrays typically done with methods taking blocks Next topic
<section-header><section-header><section-header><section-header><section-header><section-header><section-header><section-header><section-header><section-header><table-container><table-container><table-container><table-container><list-item><list-item><equation-block><table-row><table-container></table-container></table-row></equation-block></list-item></list-item></table-container></table-container></table-container></table-container></section-header></section-header></section-header></section-header></section-header></section-header></section-header></section-header></section-header></section-header>	<section-header><list-item><list-item><list-item><list-item><list-item><list-item><list-item><list-item><list-item><list-item></list-item></list-item></list-item></list-item></list-item></list-item></list-item></list-item></list-item></list-item></section-header>

Blocks everywhere

- · Rampant use of great block-taking methods in standard libraray
- Ruby has loops but very rarely used
- Can write (0..i).each {|j| e}, but often better options · Examples (consult documentation for many more)

```
a = Array.new(5) { | i | 4*(i+1) }
a.each { puts "hi" }
a.each \{|\mathbf{x}| \text{ puts } (\mathbf{x} * 2)\}
a.map { |x| x * 2 } #synonym: collect
a.any? \{|x| > 7\}
a.all? \{|x| | x > 7\}
a.inject(0) {|acc,elt| acc+elt }
a.select { |x| x > 7 } #non-synonym: filter
```

Spring 2019

CSE341: Programming Languages

More strangeness

- · Callee does not give a name to the (potential) block argument
- Instead, just calls it with yield or yield(args) - Silly example: def silly a x.silly 5 { |b| b*2 } (yield a) + (yield 42) end
 - See code for slightly less silly example
- · Can ask block given? but often just assume a block is given or that a block's presence is implied by other arguments

Spring 2019

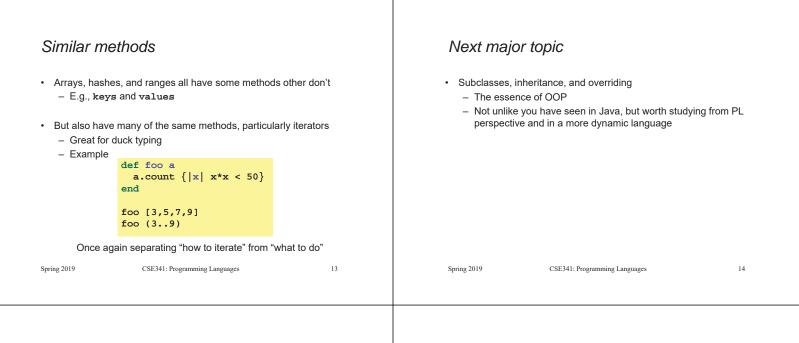
7

CSE341: Programming Languages

8

Blocks are "second-class"

Example a = [3,5,7,9] All a method can do with a block is yield to it - Cannot return it, store it in an object (e.g., for a callback), ... Blocks are fine for applying to array elements - But can also turn blocks into real closures $b = a.map \{ |\mathbf{x}| \ \mathbf{x+1} \}$ - Closures are instances of class Proc $i = b.count \{ |x| | x > = 6 \}$ Called with method call • But for an array of closures, need **Proc** objects This is Ruby, so there are several ways to make Proc objects © $c = a.map \{ |x| lambda \{ |y| x > = y \} \}$ - One way: method lambda of Object takes a block and c[2].call 17 returns the corresponding Proc $j = c.count \{ |\mathbf{x}| \mathbf{x}.call(5) \}$ - More common use is callbacks Spring 2019 CSE341: Programming Languages Spring 2019 CSE341: Programming Languages 10 Moral More collections · Hashes like arrays but: · First-class ("can be passed/stored anywhere") makes closures - Keys can be anything; strings and symbols common more powerful than blocks - No natural ordering like numeric indices - Different syntax to make them · But blocks are (a little) more convenient and cover most uses Like a dynamic record with anything for field names - Often pass a hash rather than many arguments · This helps us understand what first-class means · Ranges like arrays of contiguous numbers but: Language design question: When is convenience worth making - More efficiently represented, so large ranges fine something less general and powerful? Good style to: - Use ranges when you can - Use hashes when non-numeric keys better represent data Spring 2019 CSE341: Programming Languages 11 Spring 2019 CSE341: Programming Languages 12



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
 - Subclassing has nothing to do with a (non-existent) type system: can still (try to) call any method on any object

Spring 2019

Spring 2019

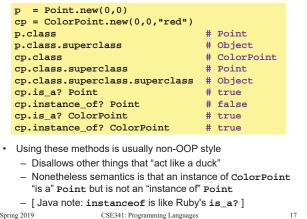
CSE341: Programming Languages

15

Example (to be continued)

class Point	class ColorPoint < Point
<pre>attr_accessor :x, :y</pre>	attr_accessor :color
<pre>def initialize(x,y)</pre>	<pre>def initialize(x,y,c)</pre>
$\mathbf{@x} = \mathbf{x}$	<pre>super(x,y)</pre>
@y = y	@color = c
end	end
def distFromOrigin	end
<pre># direct field access</pre>	
Math.sqrt(@x*@x	
+ @y*@y)	
end	
def distFromOrigin2	
# use getters	
Math.sqrt(x*x	
+ y*y)	
end	
end	
Spring 2019 CSE341: Program	amming Languages 16

An object has a class



CSE341: Programming Languages

Example continued

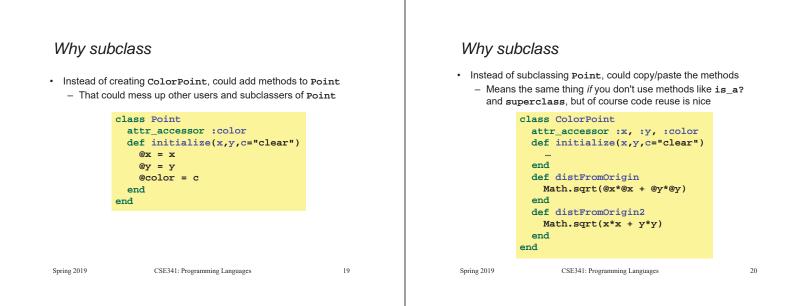
Consider alternatives to:

class ColorPoint < Point attr accessor :color def initialize(x,y,c) super(x,y) (color = cend end

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

Spring 2019

18

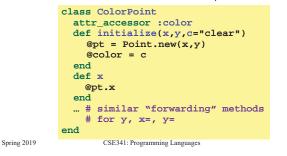


21

23

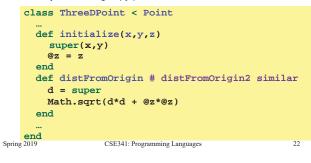
Why subclass

- 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



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



Example: Equivalent except constructor

CSE341: Programming Languages

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

```
Spring 2019
```

CSE341: Programming Languages

end def y @r * Math.sin(@theta)

@r * Math.cos(@theta)

class PolarPoint < Point</pre>

@theta = theta

@r = r

end

def x

def initialize(r,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

```
Spring 2019
```