<section-header> CSE341: Programming Languages Lecture 20 Arrays and Such, Blocks and Procs, Inheritance and Overriding Dan Grossman Autumn 2017</section-header>	 Different end of the second second
 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
Blocks are probably Ruby's strangest feature compared to other PLs But almost just closures Normal: easy way to pass anonymous functions to methods for all the usual reasons Normal: Blocks can take 0 or more arguments Normal: Blocks use lexical scope: block body uses environment where block was defined Examples \$	<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 {|x| puts (x * 2) }
a.map {|x| x * 2 } #synonym: collect
a.any? {|x| x > 7 }
a.all? {|x| x > 7 }
a.inject(0) {|acc,elt| acc+elt }
a.select {|x| x > 7 } #non-synonym: filter
```

Autumn 2017

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
 (yield a) + (yield 42)
 end
 x.silly 5 { |b| b*2 }
 - 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

```
Autumn 2017
```

CSE341: Programming Languages

Blocks are "second-class"

```
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), ...
- But can also turn blocks into real closures
- Closures are instances of class Proc
- Called with method call
- This is Ruby, so there are several ways to make ${\tt Proc}~{\tt objects}\ensuremath{\textcircled{\sc objects}}$
 - One way: method lambda of Object takes a block and returns the corresponding Proc

Autumn 2017

CSE341: Programming Languages

Example

a = [3,5,7,9]

· Blocks are fine for applying to array elements

 $b = a.map \{ |x| x+1 \}$ i = b.count $\{ |x| x>=6 \}$

But for an array of closures, need Proc objects
 More common use is callbacks

```
c = a.map {|x| lambda {|y| x>=y}}
c[2].call 17
j = c.count {|x| x.call(5) }
```

Autumn 2017

CSE341: Programming Languages

10

12

8

Moral

Autumn 2017

- First-class ("can be passed/stored anywhere") makes closures more powerful than blocks
- · But blocks are (a little) more convenient and cover most uses
- · This helps us understand what first-class means
- Language design question: When is convenience worth making something less general and powerful?

CSE341: Programming Languages

More collections

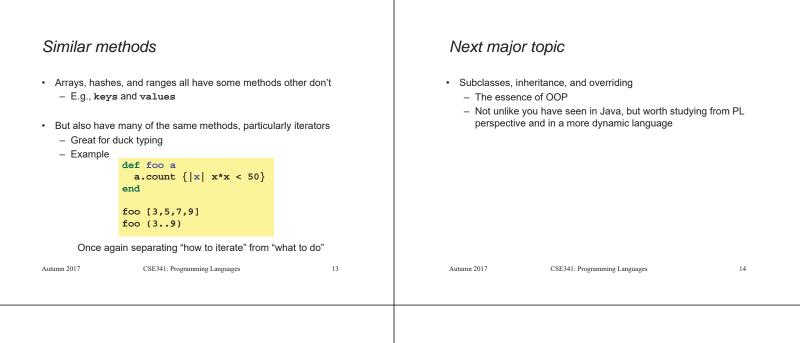
- Hashes like arrays but:
 - Keys can be anything; strings and symbols common
 - No natural ordering like numeric indices
 - Different syntax to make them
 - Like a dynamic record with anything for field names
 - Often pass a hash rather than many arguments
- · Ranges like arrays of contiguous numbers but:
 - More efficiently represented, so large ranges fine

Good style to:

- Use ranges when you can
- Use hashes when non-numeric keys better represent data

Autumn 2017

7



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

Autumn 2017

CSE341: Programming Languages

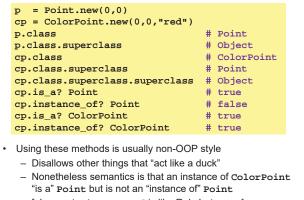
15

17

Example (to be continued)

C	lass Point	class ColorPoint < Point
	attr_accessor :x, :y	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	
e	nd	
	Autumn 2017 CSE341: Program	nming Languages 16

An object has a class



– [Java note: instanceof is like Ruby's is_a?]

Autumn 2017 CSE341: Programming Languages

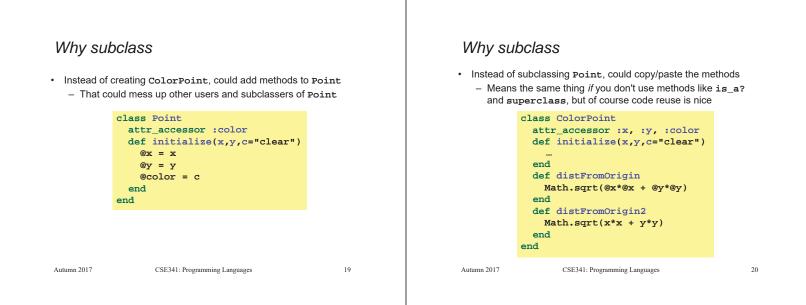
Example continued

Consider alternatives to:

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

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

Autumn 2017

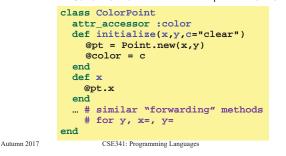


21

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



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

```
Autumn 2017
```

CSE341: Programming Languages

Example: Equivalent except constructor

<pre>class PolarPoint < Point def initialize(r,theta) @r = r</pre>	 Also need to define x= and y= (see code file)
<pre>@theta = theta end def x @r * Math.cos(@theta)</pre>	 Key punchline: distFromOrigin2, defined in Point, "already works"
end def y @r * Math.sin(@theta)	<pre>def distFromOrigin2 Math.sqrt(x*x+y*y) end</pre>
end def distFromOrigin @r end end	 Why: calls to self are resolved in terms of the object's class
Autumn 2017 CSE341: Program	nming Languages 24