CSE 341:
Programming Languages

Hal Perkins
Spring 2011
Lecture 20—Duck Typing, Blocks & Processes & Iterators, Inheritance & Overriding
Today

Three separate topics (mostly the last one?)

1. “Duck Typing”

2. Blocks and iterators (closures in Ruby)

3. Subclassing (inheritance, overriding, dynamic-dispatch, some design issues)

Textbook and/or Section: Essential stuff for upcoming homework

• Much more on Array and Hash

• Exploratory programming

• More on blocks and iterators
Duck Typing

“If it walks like a duck and quacks like a duck, it’s a duck.”

A method might think, “I need a Foo” but really it only needs an object that has similar enough methods that it acts enough like a Foo that the method works.

Embracing duck typing: Methods that make method calls rather than assume the class of their argument.

Plus: More code reuse, very OO approach

• What messages can some object receive is all that matters

Minus: Almost nothing is equivalent

• \(x+x\) versus \(x\times2\) versus \(2\times x\)

• Callees may not want callers assuming so much
Blocks and Iterators

Many methods in Ruby “take a block,” which is a “special” thing separate from the argument list.

They are used very much like closures in functional programming; can take 0 or more arguments (see examples)

The preferred way for iterating over arrays, doing something \( n \) times, etc.

They really are closures (can access local variables where they were defined).

Useful on homework: each, possibly times, inject

Useful in Ruby: many, many more
Blocks vs. Procs

These block arguments can be used only by the “immediate” callee via the yield keyword.

If you really want a “first-class object” you can pass around, store in fields, etc., convert the block to an instance of Proc.

- `lambda { |x,y,z| e }`

- Instances of Proc have a method `call`

- This `really` is exactly a closure.

Actually, there is a way for the caller to pass a block and the callee convert it to a Proc.

- Look it up if you’re curious.

- This is what `lambda` does
  (just a method in `Object` that returns the Proc it creates)
Subclasses

Ruby is dynamically typed, so subclassing is *not* about what type-checks.

Subclassing is about *inheriting methods* from the superclass.

- In Java, it’s about inheriting fields too, but we can just write to any field we want.

Example: ThreeDPoint inherits methods x and y.

Example: ColorPoint inherits distFromOrigin and distFromOrigin2.
Overriding

If it were just inheritance, then with dynamic typing subclassing would just be avoiding copy/paste.

It's more.

But first, “simple” overriding lets us redefine methods in the subclass.

- Often convenient to use super to use superclass definition in our definition.

This is still “just” avoiding copy-paste.

Example: distFromOrigin and initialize in ThreeDPoint.
Ruby-ish Digression

Why make a subclass when we could just add/change methods to the class itself?

- Add a color field to Point itself
- Affects all Point instances, even those already created (!)

Plus: Now a ThreeDPoint has a color field too.

Minus: Maybe that messes up another part of your program.

Fun example: Redefining Fixnum’s + to return 5.
Late-Binding

So far, this OO stuff is honestly very much like functional programming

- Fields are just like things in a closure’s environment

But this is totally different:

- When a method defined in a superclass makes a self call it resolves to the method defined in the subclass (typically via overriding)

Example: distFromOrigin2 in PolarPoint still works correctly!!!

Next lecture: Studying this very carefully.