CSE 341, Spring 2011, Lecture 20 Summary

Standard Disclaimer: These comments may prove useful, but certainly are not a complete summary of all the important stuff we did in class. They may make little sense if you missed class, but hopefully will help you organize and process what you have learned.

This lecture covers three topics, the last of which is essential to OO programming and will be considered more generally next time:

1. "Duck typing" — a sort of conceptual approach to dynamically typed OO programming)
2. Blocks, `Procs`, and iterators — Ruby's convenient, pervasive but somewhat idiosyncratic use of function closures
3. Subclassing, inheritance, and dynamic dispatch — the most essential aspect of OO programming

Duck Typing

In dynamically typed OO languages, one often implicitly assumes some object is an instance of a particular class. But no type system prevents an instance of a different class, and if the other class "behaves enough like" the expected class, the code you are writing may still work. For example, if we write a method like

```
def double x
  x + x
end
```

we may have "thought" the method expects an `Integer`, but it works for any class with a `+` method that can be passed `self` as its argument.

On the one hand, this flexibility can allow more code reuse and can allow callers to use "similar" objects of unexpected classes. You can write very flexible code by not explicitly testing whether an object is an instance of a certain class. Instead, you call just the methods you need. On the other hand, this means far fewer code changes produce equivalent code. For example, for integers, `x+x` and `x*2` are equivalent, but this may not be true in general.

This idea — that the "type" of a variable is based entirely on the methods used on that variable — is sometimes called "duck typing." This refers to the expression, "If it walks like a duck and quacks like a duck, it's a duck." One can make a philsophical argument: If all you know about ducks is that they walk and quack and you have something that walks and quacks, you cannot tell it is not a duck, so is there any harm in supposing that it is one?

Ruby's Closures

While Ruby has while loops and for loops not unlike Java, much Ruby code written in good style does not use them. Instead, many classes have methods that take *blocks*. These blocks are *almost* closures. For example, integers have a `times` method that takes a block and executes it the number of times you would imagine. For example,

```
x.times { puts "hi" }
```

prints `"hi"` 3 times if `x` is bound to 3 in the envioronment. To pass a block to a method, you put it in braces after the method call. The example above has no regular arguments, but a method can take any number of regular arguments and then 0 or 1 block.

Blocks are closures in the sense that they can refer to variables in scope where the block is defined. For example, after this program executes, `y` is bound to 10:

```
y = 7
[4,6,8].each { y += 1 }
```

Here [4,6,8] is an array with with 3 elements. Arrays have a method `each` that takes a block and executes it once for each element. Typically, however, we want the block to be passed each array element. We do that like this, for example to sum an array's elements and print out the running sum at each point:

```
sum = 0
[4,6,8].each { |x|
  sum += x
  puts sum
}
```

When calling a method that takes a block, you should know how many arguments will be passed to the block when it is called. For the `each` method in `Array`, the answer is 1, but as the first example showed, you can ignore some arguments by writing fewer variables between the `|` characters (or omitting them entirely).

Many collections, including arrays, have a variety of block-taking methods that look very familiar to functional programmers. For example, `inject` is just like the `fold` we studied in ML:

```
sum = [4,6,8].inject(0) { |acc,elt| acc + elt }
```

The argument to `inject` is the initial accumulator. If you omit it, `inject` will use the 0th element of the array as the initial accumulator and start with the next array element. Some other useful *iterators* (methods that take care of iterating through the elements in one way or another) are `map` and `any?`. In two lectures, we will learn how many of the iterators are actually defined in terms of `each` in a *mixin* so that they do not have to be reimplemented for each collection.

While many uses of blocks involve calling methods in the standard library, you can also define your own methods that take blocks. In fact, you can pass a block to *any* method. The method body calls the block using the `yield` keyword. For example, this code prints `"hi"` 3 times:

```
def foo x
  if x
    yield
  else
    yield
    yield
end
foo true { puts "hi" }
foo false { puts "hi" }
```

To pass arguments to a block, you put the arguments after the `yield`, e.g., `yield 7` or `yield(8,"str")`.

Blocks are not quite closures because they are not objects. We cannot store them in a field, pass them as a regular method argument, assign them to a variable, put them in an array, etc. (Notice in ML and Scheme, we could do the equivalent things with closures.) However, Ruby has "real" closures too: The class `Proc` has instances that are closures. The method `call` in `Proc` is how you apply the closure to arguments, for example `x.call` (for no arguments) or `x.call(3,4)`.

To make a `Proc` out of a block, just write `lambda { ... }` where `{ ... }` is any block. Interestingly, `lambda` is not a keyword. It is just a method in class `Object` (and every class is a subclass of `Object`, so `lambda` is available everywhere) that creates a `Proc` out of a block it is passed. You can define your own methods that do this too, but we won't go into the syntax that accomplishes this.

Subclassing, Inheritance, and Dynamic Dispatch

Subclassing is an essential feature of object-oriented programming. If class `C` is a subclass of `D` than every instance of `C` is also an instance of `D`. The definition of `C` can *inherit* the methods of `D`, i.e., they are part of `C`'s definition too. Moreover, `C` can *extend* by defining new methods that `C` has and `D` doesn't. And it can *override* methods, by changing their definition from the definition in the superclass. In Ruby, this is much like in Java. In Java, a sublcass also inherits the field definitions of the superclass, but in Ruby fields are not part of a class definition.

In the code posted with this lecture, we considered an example with the class `Point` and three different subclasses:

- `ColorPoint`, which is like a `Point` except it also has a color field
- `ThreeDPoint`, which is like a `Point` except it also has a `z` coordinate, and this changes how an object's `distFromOrigin` is computed
- `PolarPoint`, which has all the functionality of a `Point` but its constructor takes an `r` and `theta` instead of an `x` and `y` — it also uses different fields to represent points differently

Each of these subclasses raised interesting design issues, and `PolarPoint` reveals the biggest semantic difference between functional and object-oriented programming.

For `ColorPoint`, why define a new class rather than just add new methods to `Point`? Either is fine depending what we want to do. Notice that if we add methods to `Point`, then all subclasses of `Point`, such as `ThreeDPoint` and `PolarPoint`, will also have the new methods.

For `ThreeDPoint`, computer scientists have been arguing for decades about whether this subclassing is good style. On the one hand, it does let us reuse quite a bit of code, such as the methods `x`, `x=`, `y`, `y=`, and more thanks to calls to `super` in the methods we override. On the other hand, one could argue that a `ThreeDPoint` is not conceptually a `Point`, so passing the former when some code expects the latter could be inappropriate. Others say a `ThreeDPoint` is a `Point` because you can "think of it" as its projection onto the plane where `z` equals 0. We will not resolve this legendary argument, but you should appreciate that often subclassing is bad/confusing style even if it lets you reuse some code in a superclass.

The most interesting subclass we considered was `PolarPoint`. It overrides every method from `Point` except `distFromOrigin2`. The constructor behaves differently since it creates different fields (and no `@x` and `@y` fields are created since it does not call `super`). It overrides the `x`, `x=`, `y`, and `y=` methods in a way that preserves a point's behavior. Recall that the `attr_reader` and `attr_writer` lines in the definition of `Point` are just a shorthand for defining getter and setter methods. So a subclass is free to override these methods with other definitions. Notice this is much more flexible than a `Point` class in Java with public fields `x` and `y`. We also override `distFromOrigin` to take advantage of the fact that the polar representation of points makes computing the result trivial.

The really interesting thing, though, is that we do not need to override `distFromOrigin2`. To see why, consider the definition in the superclass:

```
def distFromOrigin2
  Math.sqrt(x * x + y * y) # uses getter methods
end
```

Unlike the definition of `distFromOrigin`, this method uses other method calls for the arguments to the multiplications. Recall this is just syntactic sugar for:

```
def distFromOrigin2
  Math.sqrt(self.x() * self.x() + self.y() * self.y()) # uses getter methods
end
```

In the superclass, this can seem like an unnecessary complication since `self.x()` is just a method that returns `@x` and methods of `Point` can access `@x` directly, as `distFromOrigin` does.

However, as you learned in your 100-level Java courses, overriding methods `x` and `y` in a subclass of `Point` changes how `distFromOrigin2` behaves in instances of the subclass. Given a `PolarPoint` instance, its `distFromOrigin2` method is defined with the code above, but when called, `self.x` and `self.y` will call the methods defined in `PolarPoint`, not the methods defined in `Point`.

This semantics goes by many names, including *dynamic dispatch*, *late binding*, and *virtual method calls*. There is nothing quite like it in functional programming, since the way `self` is treated in the environment is special, as the next lecture considers in more detail.