CSE341, Winter 2014, Blocks; Inheritance Summary

Disclaimer: This lecture summary is not necessarily a complete substitute for attending class, reading the associated code, etc. It is designed to be a useful resource for students who attended class and are later reviewing the material.

This lecture covers two separate topics:

1. Blocks, `Procs`, and iterators — Ruby's convenient, pervasive but somewhat strange approach to function closures
2. Subclassing, inheritance, and dynamic dispatch — the most essential aspect of OOP

Ruby's Blocks

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 environment. 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 arguments if you have no need for them by omitting the `|...|`.

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 Haskell:

```
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 a later lecture, we will learn that 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.

Using blocks in your own methods

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")`.

Using this approach, the fact that a method may expect a block is implicit; it is just that its body might use `yield`. An error will result if `yield` is used and no block was passed. The behavior when the block and the `yield` disagree on the number of arguments is somewhat flexible and not described in full detail here. A method can use the `block_given?` primitive to see if the caller provided a block. You are unlikely to use this method often: If a block is needed, it is conventional just to assume it is given and have `yield` fail if it is not. In situations where a method may or may not expect a block, often other regular arguments determine whether a block should be present. If not, then `block_given?` is appropriate.

Full Closures: The `Proc` Class

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 Haskell and Racket, 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.

A Recursive (But Unnecessary) Example

Consider a linked-list class called `MyList` that uses instance variables `@head` and `@tail` in the expected ways, using the `nil` object to represent the empty-list. (The code accompanying these notes shows such a class with several useful methods.) One would not use such a class in typical Ruby code since the `Array` class is already plenty flexible to serve for purposes where in less dynamic languages we would use arrays or tuples or lists. After all, there are methods in `Array` for adding elements at the beginning, extracting the slice that omits the first element, mapping over an array, etc. But a linked-list still makes a useful example.

Suppose we want to implement a `map` method for `MyList` that works like the common `map` function from functional programming: It builds a new list of the same length by applying a caller-provided operation to each element. Using a `Proc` object leads to a straightforward solution:

```
def map proc
    if @tail.nil?
      MyList.new(proc.call(@head), nil)
    else
      MyList.new(proc.call(@head), @tail.map(proc))
    end
end
```

As a side-note, notice that a more object-oriented approach would be for the `NilClass` (the class of `nil`) to also have a `map` method that returns `nil`. If it did (or if we added it), then our method in `MyList` could just be:

```
def map proc
    MyList.new(proc.call(@head), @tail.map(proc))
end
```

In Ruby, it is conventional to use blocks instead of `Proc` objects. One approach would be to provide callers a `map` method that took a block, used `lambda` to create a `Proc` and then used the code above as a private recursive helper method named something like `map_helper`. The reason for doing this is that we cannot pass the block we are given to a recursive call since we have no "name" for the block — we can only call `yield`.

But there is a way to implement `map` using only blocks. We show the solution followed by an explanation:

```
def map
    if @tail.nil?
      MyList.new(yield(@head), nil)
    else
      MyList.new(yield(@head), @tail.map {|x| yield x})
    end
  end
```

The key trick is `{|x| yield x}`, which passes the recursive call a new block that, when `yield` is called on it will then `yield` to the block passed to this method, which is what we wanted. This is analogous to the unnecessary function wrapping we studied in functional languages (do not write `fn x => f x` because you can write `f`), but in this case, the wrapping is necessary because there is no name like `f` for the current block. It may depend on the Ruby implementation whether or not this technique is efficient.

As with the `Proc` solution, a more object-oriented approach would add a `map` method to `NilClass`; this is orthogonal to blocks versus `Proc`.

Subclassing, Inheritance, and Dynamic Dispatch

Basic Idea and Terminology

Subclassing is an essential feature of class-based OOP. If class `C` is a subclass of `D`, then every instance of `C` is also an instance of `D`. The definition of `C` *inherits* 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` does not. And it can *override* methods, by changing their definition from the inherited definition. In Ruby, this is much like in Java. In Java, a subclass also inherits the field definitions of the superclass, but in Ruby fields are not part of a class definition because each object instance just creates its own instance variables.

Every class in Ruby except `Object` has one superclass. The classes form a tree where each node is a class and the parent is its superclass. The `Object` class is the root of the tree. In class-based languages, this is called the *class hierarchy*. By the definition of subclassing, a class has all the methods of all its ancestors in the tree (i.e., all nodes between it and the root, inclusive), subject to overriding.

Some Ruby Specifics

- A Ruby class definition specifies a superclass with `class C < D ... end` to define a new class `C` with superclass `D`. Omitting the `< D` implies `< Object`, which is what our examples so far have done.
- Ruby's built-in methods for reflection can help you explore the class hierarchy. Every object has a `class` method that returns the class of the object. Consistently, if confusingly at first, a class is itself an object in Ruby (after all, every value is an object). The class of a class is `Class`. This class defines a method `superclass` that returns the superclass.
- Every object also has methods `is_a?` and `instance_of?`. The method `is_a?` takes a class (e.g., `x.is_a? Integer`) and returns true if the receiver is an instance of `Integer` or any (transitive) subclass of `Integer`, i.e., if it is below `Integer` in the class hierarchy. The method `instance_of?` is similar but returns true only if the receiver is an instance of the class exactly, not a subclass. Note that in Java the primitive `instanceof` is analogous to Ruby's `is_a?`.

Using methods like `is_a?` and `instanceof` is "less object-oriented" and therefore often not preferred style. They are in conflict with duck typing.

A First Example: Point and ColorPoint

Here are definitions for simple classes that describe simple two-dimensional points and a subclass that adds a color (just represented with a string) to instances.

```
class Point
  attr_reader :x, :y
  attr_writer :x, :y
  def initialize(x,y)
    @x = x
    @y = y
  end
  def distFromOrigin
    Math.sqrt(@x * @x  + @y * @y)
  end
  def distFromOrigin2
    Math.sqrt(x * x + y * y)
  end
end
class ColorPoint < Point
  attr_reader :color
  attr_writer :color
  def initialize(x,y,c="clear")
    super(x,y)
    @color = c
  end
end
```

There are many ways we could have defined these classes. Our design choices here include:

- We make the `@x`, `@y`, and `@color` instance variables mutable, with public getter and setter methods.

- The default "color" for a `ColorPoint` is `"clear"`.
- For pedagogical purposes revealed below, we implement the distance-to-the-origin two different ways. The `distFromOrigin` method accesses instance variables directly whereas `distFromOrigin2` uses the getter methods on `self`. Given the definition of `Point`, both will produce the same result.

The `initialize` method in `ColorPoint` uses the `super` keyword, which allows an overriding method to call the method of the same name in the superclass. This is not required when constructing Ruby objects, but it is often desired.

Given the existence of `Point`, defining `ColorPoint` is good style because it allows us to reuse much of our work from `Point` and it makes sense to treat any instance of `ColorPoint` as though it "is a" `Point`. But it is worth considering three alternative ways to define the `ColorPoint` class.

First, we could just define `ColorPoint` "from scratch," copying over (or retyping) the code from `Point`. In a dynamically typed language, the difference in *semantics* (as opposed to style) is small: instances of `ColorPoint` will now return false if sent the message `is_a?` with argument `Point`, but otherwise they will work the same. In languages like Java, superclasses have effects on static typing. One advantage of not subclassing `Point` is that any later changes to `Point` will not affect `ColorPoint` — in general in class-based OOP, one has to worry about how changes to a class will affect any subclasses.

Second, we could have `ColorPoint` be a subclass of `Object` but have it contain an instance variable, call it `@pt`, holding an instance of `Point`. Then it would need to define all of the methods defined in `Point` to forward the message to the object in `@pt`. Here are two examples, omitting all the other methods (`x=`, `y`, `y=`, `distFromOrigin`, `distFromOrigin2`):

```
def initialize(x,y,c="clear")
  @pt = Point.new(x,y)
  @color = c
end
def x
  @pt.x
end
```

This approach is bad style since again subclassing is shorter and we want to treat a `ColorPoint` as though it "is a" `Point`. But in general, many programmers in object-oriented languages overuse subclassing. In situations where you are making a new kind of data that includes a pre-existing kind of data *as a separate sub-part of it*, this instance-variable approach is better style.

Third, in Ruby, we can extend and modify classes with new methods. So we could simply change the `Point` class by replacing its `initialize` method and adding getter/setter methods for `@color`. This would be appropriate only if every `Point` object, including instances of all other subclasses of `Point`, should have a color.

Simple Overriding and Three-Dimensional Points

Now let's consider a different subclass of `Point`, which is for three-dimensional points:

```
class ThreeDPoint < Point
  attr_reader :z
  attr_writer :z
  def initialize(x,y,z)
    super(x,y)
    @z = z
  end
  def distFromOrigin
    d = super
    Math.sqrt(d * d + @z * @z)
  end
  def distFromOrigin2
    d = super
    Math.sqrt(d * d + z * z)
  end
end
```

Here, the code-reuse advantage is limited to inheriting methods `x`, `x=`, `y`, and `y=`, as well as using other methods in `Point` via `super`. Notice that in addition to overriding `initialize`, we used overriding for `distFromOrigin` and `distFromOrigin2`.

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. On the other hand, one could argue that a `ThreeDPoint` is *not* conceptually a (two-dimensional) `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 argument against subclassing is made stronger if we have a method in `Point` like `distance` that takes another (object that behaves like a) `Point` and computes the distance between the argument and `self`. If `ThreeDPoint` wants to override this method with one that takes another (object that behaves like a) `ThreeDPoint`, then `ThreeDPoint` instances will *not* act like `Point` instances: their `distance` method will fail when passed an instance of `Point`.

More Interesting Overriding (Dynamic Dispatch) with Polar Points

The final subclass of `Point` we will define describes objects that behave equivalently to instances of `Point` (except for the arguments to `initialize`) but use an internal representation in terms of polar coordinates (radius and angle):

```
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 x= a
    b = y # avoids multiple calls to y method
    @theta = Math.atan(b / a)
    @r = Math.sqrt(a*a + b*b)
    self
  end
  def y= b
    a = y # avoid multiple calls to y method
    @theta = Math.atan(b / a)
    @r = Math.sqrt(a*a + b*b)
    self
  end
  def distFromOrigin
    @r
  end
  # distFromOrigin2 already works!!
end
```

Notice instances of `PolarPoint` do not have instance variables `@x` and `@y`, but the class does override the `x`, `x=`, `y`, and `y=` methods so that clients cannot tell the implementation is different: they can use instances of `Point` and `PolarPoint` interchangeably. A similar example in Java would still have fields from the superclass, but would not use them. The advantage of `PolarPoint` over `Point`, which admittedly is for sake of example, is that `distFromOrigin` is simpler / more efficient.

The key point of this example is that **the subclass does not override `distFromOrigin2`, but the inherited method works correctly**. To see why, consider the definition in the superclass:

```
  def distFromOrigin2
    Math.sqrt(x * x + y * y)
  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())
  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 when you studied Java's support for class-based OOP, 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.