1 Subclassing

Subclassing is an essential feature of class-based OOP. If class \( C \) is a subclass of \( D \), then \( C \) inherits the methods of \( D \)—that is, they are part of \( C \)'s definition too. Moreover, \( C \) can extend by defining new methods that \( D \) does not have. And it can override methods, by changing their definition from the inherited definition.

In Java, a subclass also inherits the field definitions of the superclass. But in Ruby, fields are not part of a class definition: each object instance creates its own instance variables.

The root class in Ruby is \( \text{Object} \). Classes form a tree, called the class hierarchy, in which each node is a class and a node's parent is its superclass. 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), though some methods' implementation may have been changed by overriding.

A Ruby class definition specifies a superclass with `class C < D ... end` to define a new class \( C \) with superclass \( D \). Omitting "\( C \)" is equivalent to writing "\( C \text{< Object} \)", which is what our examples so far have done.

Ruby's built-in methods for reflection enable exploration of 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 has methods is_a? and instance_of?:

- Method is_a? takes a class (e.g., \( x \text{.is_a? Integer} \)) and returns true if the receiver is an instance of Integer or any (transitive) subclass of Integer—that is, if the receiver is a descendant of Integer in the class hierarchy. Java's instanceof operator is analogous to Ruby's is_a?.

- Method instance_of? is similar but returns true only if the receiver is an instance of the class exactly, not a subclass.

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

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

```ruby
class Point
  attr_accessor :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_accessor :color
  def initialize(x,y,c="black")
    super(x,y)
    @color = c
  end
end

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

- We made public getter and setter methods for @x, @y, and @color.
- We chose "black" as the default color for a ColorPoint.
- For reasons revealed below, we implement distance-from-the-origin in two ways. The distFromOrigin method accesses instance variables directly, whereas distFromOrigin2 uses 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.

Subclassing Point to define ColorPoint is good practice, because it allows us to reuse code from Point, and because 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.

1. We could make ColorPoint a subclass of Object, and copy (or retype) the code from Point. Instances of ColorPoint would then return false if sent message is_a? Point. This approach is bad practice, because any subsequent changes (including bug fixes) to Point would not affect ColorPoint, unless also programmers also copy those changes.

2. We could make ColorPoint a subclass of Object, and have it contain an instance variable, call it @pt, holding an instance of Point. Thus, ColorPoint “has a” Point. Then ColorPoint 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=, distFromOrigin, distFromOrigin2):

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

There’s a great deal of debate about whether this is better practice than our original “is a” approach. (Google “extends is evil” to read more.)

3. We could eschew subclassing altogether and use Ruby’s features for replacing and adding methods to classes. We’d need to replace Point’s initialize method, and add getter and setter methods for @color. This approach would be appropriate only if every Point object, including instances of all other subclasses of Point, should have a color.
**Point and ThreeDPoint.** Now let’s consider a different subclass of `Point`, which is for three-dimensional points:

```ruby
class ThreeDPoint < Point
  attr_accessor :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` with `super`. In addition to overriding `initialize`, we used overriding for `distFromOrigin` and `distFromOrigin2`.

Is this use of subclassing good practice? Yes and no. (Programmers have been arguing the question for decades.)

**Pro:** We get to reuse code.

**Con:** Conceptually, a `ThreeDPoint` isn’t a two-dimensional `Point`[1] so passing the former when some code expects the latter could be inappropriate.

We can’t resolve this legendary argument. But you should appreciate that subclassing has tradeoffs.

**Point and PolarPoint.** `Point` represents points with Cartesian coordinates (x and y). Let’s create a subclass that uses polar coordinates (radius r and angle theta):

```ruby
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.atan2(b,a)
    @r = Math.hypot(a,b)
  end
end
```

---

[1] Unless you think of a `ThreeDPoint` as a `Point` by taking its projection onto the plane where z equals 0.
self
end
def y= b
  a = y # avoid multiple calls to y method
  @theta = Math.atan2(b,a)
  @r = Math.hypot(a,b)
  self
end
def distFromOrigin
  @r # very simple implementation
end
# distFromOrigin2 already works!!
end

Instances of PolarPoint do not have instance variables @x and @y. (Check for yourself: no code ever assigns to those variables, so they're never created. Were we to code up this example in Java, instances of PolarPoint would have fields x and y, but would never use them.) But PolarPoint does override the x, x=, y, and y= methods of Point. So clients cannot tell the implementation is different—they can use instances of Point and PolarPoint interchangeably.

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 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, 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.

2 Duck Typing

Recall the idea of subtyping from Java: anywhere an object of type T is expected, code can instead use an object of type U, so long as U is a subtype of T. There are rules for determining subtypes—for example, if U extends T, then U is a subtype of T. But Ruby doesn’t have static types. So is there an equivalent to subtyping in Ruby? Yes, and it goes by the rather odd name “duck typing.”
Duck typing refers to the expression, “If it walks like a duck and quacks like a duck, then it’s a duck” (though a better conclusion might be “...then there is no reason to concern yourself with the possibility that it might not be a duck”). In Ruby, this refers to the idea that the class of an object (e.g., `Duck`) passed to a method is not important so long as the object can respond to all the messages it is expected to (e.g., `walk(location)` or `quack_now`).

For example, consider this method:

```ruby
def mirror_update pt
  pt.x = pt.x * -1
end
```

It is natural to view this as a method that must take an instance of a particular class `Point` (not shown here) since it uses methods `x` and `x=` defined in it. And the `x` getter must return a number since the result of `pt.x` is sent the `*` message with `-1` for multiplication. But this method is more generally useful. It is not necessary for `pt` to be an instance of `Point` provided it has methods `x` and `x=`. Moreover, the `x` and `x=` methods need not be a getter and setter for an instance variable `@x`. Even more generally, we do not need the `x` method to return a number. It just has to return some object that can respond to the `*` message with argument `-1`.

Duck typing can make code more reusable, allowing clients to make “fake ducks” and still use your code. In Ruby, duck typing basically “comes for free” as long you do not explicitly check that arguments are instances of particular classes using methods like `instance_of?` or `is_a?` (see next lecture).

Duck typing has disadvantages. The most lenient specification of how to use a method ends up describing the whole implementation of a method, in particular what messages it sends to what objects. If our specification reveals all that, then almost no variant of the implementation will be equivalent. For example, if we know `i` is a number (and ignoring clients redefining methods in the classes for numbers), then we can replace `i+i` with `i*2` or `2*i`. But if we just assume `i` can receive the `+` message with itself as an argument, then we cannot do these replacements since `i` may not have a `*` method (breaking `i*2`) or it may not be the sort of object that `2` expects as an argument to `*` (breaking `2*i`).

### 3 Dynamic dispatch

Let’s work out the semantics of object-oriented language constructs, particularly calls to methods, as carefully as we have considered the semantics of functional language constructs—particularly calls to closures. As we will see, the key distinguishing feature of OOP is what `self` is bound to in the environment when a method is called. This concept is known as dynamic dispatch (a.k.a. late binding and virtual method calls).

Before discussing `self` in particular, let’s consider more generally the semantics of resolving (i.e., “looking up”) various things like fields, variables, method names, etc. Such name resolution is a key part of a programming language’s semantics. For example, lexical scope is essential to understand programming in ML.

In Ruby, unlike Java, deciding whether some name like `f` refers to a field or a local variable is trivial because fields always start with a `@` and variables never do. There is a general principle here: rules about shadowing and name collisions are trivially avoided when there is a syntactic distinction, as there is here between fields and variables. In Java, we need a separate rule for local variables that shadow fields, though that is certainly not a big deal. Interestingly, Ruby’s choice to let you omit `()` when calling a zero-argument method creates a different potential collision: `m+2` adding 2 to the local variable `m` or is it adding 2 to the result of calling `self.m()`? It turns out local variables shadow methods, but since variables are not declared, the actual answer depends on whether the method has assigned to a variable `m` or not. This is a Ruby detail that comes up rarely, but it does point out the subtle “gotchas” that can arise when syntax is ambiguous.

To continue defining Ruby’s lookup rules, inside a block we look up the variable from the environment where the block was defined. In other words, blocks are like closures in how variables are looked up, and
the implementation of Ruby needs to associate environments with blocks just like the implementation of a functional language needs to associate environments with closures.

**Aside: method names are not first-class.** We call something *first class* if it can be computed with, stored in variables or fields, returned from functions/methods, etc. In Ruby, objects are first class and we have said before that “everything is an object,” but to be more precise “every value is an object.” Here are some things that are not objects in Ruby and therefore are not first-class in Ruby:

- **Method names:** If \( m \) and \( n \) are method names, you cannot write \( x.\langle \text{if } b \text{ then } m \text{ else } n \text{ end} \rangle \), which is trying to return the name of a method from an if-expression. Method names are not first-class. You have to write something like \( \text{if } b \text{ then } x.m \text{ else } x.n \text{ end} \).

- **Argument lists:** Suppose you have \( f(1,2,3) + g(1,2,3) \). You cannot rewrite that as \( x = (1,2,3); f x + g x \).

- **Blocks:** You cannot, for example, store a block in a field of an object. That is exactly why class Proc exists.

The notion of what is first-class is an important in any programming-language design.

**How to treat self.** Since method names are not first-class and cannot be bound to variables, we do not look them up in the environment like variables. We do evaluate the *receiver of a message (method call)* as a first-class expression, and then we use that object’s class to determine what method to call.

In class-based object-oriented languages like Ruby and Java, the rule for evaluating a method call like \( e0.m(e1,\ldots, en) \) is:

- Evaluate \( e0, e1, \ldots, en \) to values, i.e., objects \( obj0, obj1, \ldots, objn \).

- Get the class of \( obj0 \). Every object “knows its class” at run-time. Think of the class as a special instance variable of \( obj0 \).

- Suppose \( obj0 \) has class \( A \). If \( m \) is defined in \( A \), call that method. Otherwise recur with the superclass of \( A \) to see if it defines \( m \). Raise a “message not understood” error if neither \( A \) nor any of its superclasses define \( m \). (Ruby actually resolves undefined methods by calling the receiver’s method_missing method, which is defined in Object to raise an error.)

- (Ruby’s mixins complicate the lookup rules a bit more, as we will see in a future lecture.)

- Now that we have defined what method to call, we still have to define what environment to use when evaluating its body. If the method has *formal arguments* (i.e., argument names or parameters) \( x1, x2, \ldots, xn \), then the environment for evaluating the body will map \( x1 \) to \( obj1 \), \( x2 \) to \( obj2 \), etc. But there is one more thing that is the essence of object-oriented programming and has no real analogue in functional programming: We always have \( self \) in the environment. **While evaluating the method body, \( self \) is bound to \( obj0 \), the object that is the “receiver” of the message.**

This last phrase is what is meant by the synonyms “late-binding,” “dynamic dispatch,” and “virtual method calls.” It is central to the semantics of Java and Ruby. It means that when the body of \( m \) calls a method on \( self \) (e.g., \( self.someMethod 34 \) or just \( someMethod 34 \)), we use the class of \( obj0 \) to resolve \( someMethod \), not necessarily the class where \( someMethod \) is defined.

This semantics is:
Exactly why our example from last lecture “worked” when we defined a PolarPoint class and did not have to override the version of distFromOrigin that used getter methods. The method in the superclass made calls to self.x and self.y, so when self was bound to a PolarPoint, we used the definitions of x and y in PolarPoint.

“Old news” to you because you learned it in your introductory Java courses. Now we can just give it a more precise definition because we understand that it is really about what self is bound to in the environment.

More complicated than ML function calls, though it may not seem that way to you, because you learned it first. We have to treat the notion of self differently from everything else in the language. Complicated does not necessarily mean inferior or superior; it just means the language definition is harder to describe.

4 Dynamic dispatch vs. closures

To understand how dynamic dispatch differs from the lexical scope we used for function calls, consider this simple ML code that defines two mutually recursive functions:

fun even x = if x=0 then true else odd (x-1)
and odd x = if x=0 then false else even (x-1)

This creates two closures that both have the other closure in their environment. If we later shadow the even closure with something else, e.g.,

fun even x = false

that will not change how odd behaves. When odd looks up even in the environment where odd was defined, it will get the function on the first line above. That is “good” for understanding how odd works just from looking where is defined. On the other hand, suppose we wrote a better version of even like:

fun even x = (x mod 2) = 0

Now our odd is not “benefitting from” this optimized implementation.

In OOP, we can use (abuse?) subclassing, overriding, and dynamic dispatch to change the behavior of odd by overriding even:

class A
def even x
    if x==0 then true else odd(x-1) end
end
def odd x
    if x==0 then false else even(x-1) end
end
class B < A
def even x # changes B’s odd too!
    x % 2 == 0
end
Now (B.new.odd 17) will execute faster because odd’s call to even will resolve to the method in B – all because of what self is bound to in the environment. While this is certainly convenient in the short example above, it has real drawbacks. We cannot look at one class (A) and know how calls to the code there will behave. In a subclass, what if someone overrode even and did not know that it would change the behavior of odd? Basically, any calls to methods that might be overridden need to be thought about very carefully. It is likely often better to have private methods that cannot be overridden to avoid problems. Yet overriding and dynamic dispatch is the biggest thing that distinguishes object-oriented programming from functional programming.