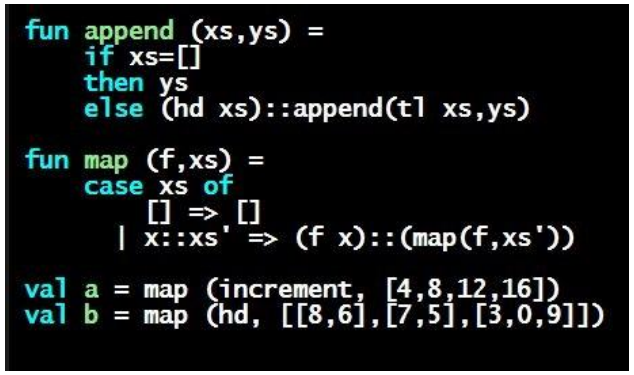

```
fun append (xs,ys) =
    if xs=[]
    then ys
    else (hd xs)::append(tl xs,ys)

fun map (f,xs) =
    case xs of
        [] => []
      | x::xs' => (f x)::(map(f,xs'))

val a = map (increment, [4,8,12,16])
val b = map (hd, [[8,6],[7,5],[3,0,9]])
```

Programming Languages

Dan Grossman
2013

Overriding and Dynamic Dispatch

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

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

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

Example: Equivalent except constructor

```
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 distFromOrigin
    @r
  end
  …
end
```

- Also need to define **`x=`** and **`y=`** (see code file)
- Key punchline: **`distFromOrigin2`**, defined in **`Point`**, "already works"

```
def distFromOrigin2
  Math.sqrt(x*x+y*y)
end
```

  - Why: calls to **`self`** are resolved in terms of the object's class