```
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

Optional: Abstract Methods

Connections

Answered in this segment:

- What does a statically typed OOP language need to support "required overriding"?
- How is this similar to higher-order functions?
- Why does a language with multiple inheritance (e.g., C++) not need Java/C#-style interfaces?

[Explaining Java's abstract methods / C++'s pure virtual methods]

Required overriding

Often a class expects all subclasses to override some method(s)

- The purpose of the superclass is to abstract common functionality, but some non-common parts have no default

A Ruby approach:

- Do not define must-override methods in superclass
- Subclasses can add it
- Creating instance of superclass can cause method-missing errors

```
# do not use A.new
# all subclasses should define m2
class A
  def m1 v
    … self.m2 e …
  end
end
```

Static typing

- In Java/C#/C++, prior approach fails type-checking
  - No method **`m2`** defined in superclass
  - One solution: provide error-causing implementation

```
class A
  def m1 v
    … self.m2 e …
  end
  def m2 v
    raise "must be overridden"
  end
end
```

  - Better: Use static checking to prevent this error…

Abstract methods

- Java/C#/C++ let superclass give signature (type) of method subclasses should provide
  - Called *abstract methods* or *pure virtual methods*
  - Cannot creates instances of classes with such methods
    - Catches error at compile-time
    - Indicates intent to code-reader
    - Does *not* make language more powerful

```
abstract class A {
  T1 m1(T2 x) { … m2(e); … }
  abstract T3 m2(T4 x);
}
class B extends A {
  T3 m2(T4 x) { … }
}
```

Passing code to other code

- Abstract methods and dynamic dispatch: An OOP way to have subclass "pass code" to other code in superclass

```
abstract class A {
  T1 m1(T2 x) { … m2(e); … }
  abstract T3 m2(T4 x);
}
class B extends A {
  T3 m2(T4 x) { … }
}
```

- Higher-order functions: An FP way to have caller "pass code" to callee

```
fun f (g,x) = … g e …

fun h x = … f((fn y => …),…)
```

No interfaces in C++

- If you have multiple inheritance and abstract methods, you do not also need interfaces
- Replace each interface with a class with all abstract methods
- Replace each “implements interface” with another superclass

So: Expect to see interfaces only in statically typed OOP without multiple inheritance

  - Not Ruby
  - Not C++