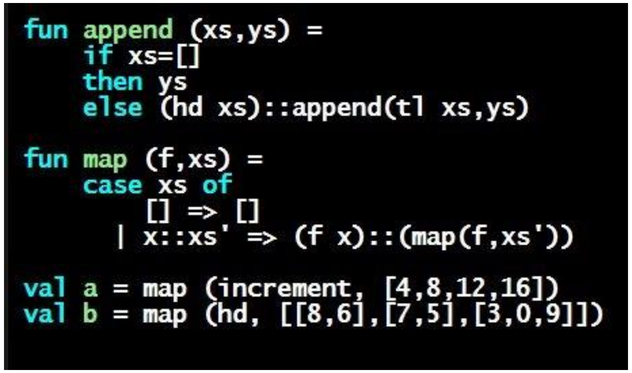

```
fun append (xs,ys) =
    if xs=[]
    then ys
    else (hd xs)::append(tl xs,ys)

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

Interfaces

Statically-Typed OOP

- Now contrast multiple inheritance and mixins with Java/C#-style interfaces

- Important distinction, but interfaces are about static typing, which Ruby does not have

- So will use Java [pseudo]code after quick introduction to static typing for class-based OOP…
  - Sound typing for OOP prevents “method missing” errors

Classes as Types

- In Java/C#/etc. each class is also a type

- Methods have types for arguments and result

```
class A {
  Object m1(Example e, String s) {…}
  Integer m2(A foo, Boolean b, Integer i) {…}
}
```

- If **C** is a (transitive) subclass of **D**, then **C** is a *subtype* of **D**
  - Type-checking allows subtype anywhere supertype allowed
  - So can pass instance of **C** to a method expecting instance of **D**

Interfaces are Types

```
interface Example {
  void   m1(int x, int y);
  Object m2(Example x, String y);
}
```

- An interface is not a class; it is only a type
  - Does not contain method *definitions*, only their *signatures* (types)
    - Unlike mixins
  - Cannot use **`new`** on an interface
    - Like mixins

Implementing Interfaces

- A class can explicitly implement any number of interfaces
  - For class to type-check, it must implement every method in the interface with the right type
    - More on allowing subtypes later!
  - Multiple interfaces no problem; just implement everything

- If class type-checks, it is a subtype of the interface

```
class A implements Example {
  public void m1(int x, int y) {…}
  public Object m2(Example e, String s) {…}
}
class B implements Example {
  public void m1(int pizza, int beer) {…}
  public Object m2(Example e, String s) {…}
}
```

Multiple interfaces

- Interfaces provide no methods or fields
  - So no questions of method/field duplication when implementing multiple interfaces, unlike multiple inheritance

- What interfaces are for:
  - "Caller can give any instance of any class implementing **`I`**"
    - So callee can call methods in **`I`** regardless of class
  - So much more flexible type system

- Interfaces have little use in a dynamically typed language
  - Dynamic typing *already* much more flexible, with trade-offs we studied