

```
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

*Subtyping for OOP*

# *Now...*

Use what we learned about subtyping for records and functions to understand subtyping for class-based OOP

- Like in Java/C#

Recall:

- Class names are also types
- Subclasses are also subtypes
- Substitution principle: Instance of subclass should be usable in place of instance of superclass

# *An object is...*

- Objects: mostly records holding fields and methods
  - Fields are mutable
  - Methods are immutable functions that also have access to **self**
- So *could* design a type system using types very much like record types
  - Subtypes could have extra fields and methods
  - Overriding methods could have contravariant arguments and covariant results compared to method overridden
    - Sound only because method “slots” are immutable!

# *Actual Java/C#...*

Compare/contrast to what our “theory” allows:

1. Types are class names and subtyping are explicit subclasses
  2. A subclass can add fields and methods
  3. A subclass can override a method with a covariant return type
    - (No contravariant arguments; instead makes it a non-overriding method of the same name)
- (1) Is a subset of what is sound (so also sound)
- (3) Is a subset of what is sound and a different choice (adding method instead of overriding)

# *Classes vs. Types*

- A class defines an object's behavior
  - Subclassing inherits behavior and changes it via extension and overriding
- A type describes an object's methods' argument/result types
  - A subtype is substitutable in terms of its field/method types
- These are separate concepts: try to use the terms correctly
  - Java/C# confuse them by requiring subclasses to be subtypes
  - A class name is both a class and a type
  - This confusion is convenient in practice

# *Optional: More details*

Java and C# are sound: They do not allow subtypes to do things that would lead to “method missing” or accessing a field at the wrong type

Confusing (?) Java example:

- Subclass can declare field name already declared by superclass
- Two classes can use any two types for the field name
- Instance of subclass have two fields with same name
- “Which field is in scope” depends on which class defined the method

# *Optional:* **self/this** is special

- Recall our Racket encoding of OOP-style
  - “Objects” have a list of fields and a list of functions that take **self** as an explicit extra argument
- So if **self/this** is a function argument, is it contravariant?
  - No, it is *covariant*: a method in a subclass can use fields and methods only available in the subclass: essential for OOP

```
class A {  
  int m() { return 0; }  
}  
class B extends A {  
  int x;  
  int m() { return x; }  
}
```

- Sound because calls always use the “whole object” for **self**