

```
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: Closure Idioms Without Closures in Java

Java

- Java 8 scheduled to have closures (like C#, Scala, Ruby, ...)
 - Write like `xs.map((x) => x.age)`
`.filter((x) => x > 21)`
`.length()`
 - Make parallelism and collections much easier
 - Encourage less mutation
- But how could we program in an ML style without help
 - Will not look like the code above
 - Was even more painful before Java had generics

One-method interfaces

```
interface Func<B,A> { B m(A x) ; }  
  
interface Pred<A> { boolean m(A x) ; }
```

- An interface is a named [polymorphic] type
- An object with one method can serve as a closure
 - Different instances can have different fields [possibly different types] like different closures can have different environments [possibly different types]
- So an interface with one method can serve as a function type

List types

Creating a generic list class works fine

- Assuming `null` for empty list here, a choice we may regret

```
class List<T> {  
    T head;  
    List<T> tail;  
    List(T x, List<T> xs) {  
        head = x;  
        tail = xs;  
    }  
    ...  
}
```

Higher-order functions

- Let's use static methods for `map`, `filter`, `length`
- Use our earlier generic interfaces for “function arguments”
- These methods are recursive
 - Less efficient in Java ☹
 - Much simpler than common previous-pointer acrobatics

```
static <A,B> List<B> map(Func<B,A> f, List<A> xs) {  
    if(xs==null) return null;  
    return new List<B>(f.m(xs.head) , map(f,xs.tail) ;  
}  
static <A> List<A> filter(Pred<A> f, List<A> xs) {  
    if(xs==null) return null;  
    if(f.m(xs.head) )  
        return new List<A>(xs.head) , filter(f,xs.tail) ;  
    return filter(f,xs.tail) ;  
}  
static <A> length(List<A> xs) { ... }
```

Why not instance methods?

A more OO approach would be instance methods:

```
class List<T> {  
    <B> List<B> map(Func<B,T> f) {...}  
    List<T> filter(Pred<T> f) {...}  
    int length() {...}  
}
```

Can work, but interacts poorly with `null` for empty list

- Cannot call a method on `null`
- So leads to extra cases in all *clients* of these methods if a list might be empty

An even more OO alternative uses a subclass of `List` for empty-lists rather than `null`

- Then instance methods work fine!

Clients

- To use **map** method to make a **List<Bar>** from a **List<Foo>**:
 - Define a class **C** that implements **Func<Bar, Foo>**
 - Use fields to hold any “private data”
 - Make an object of class **C**, passing private data to constructor
 - Pass the object to **map**
- As a convenience, can combine all 3 steps with *anonymous inner classes*
 - Mostly just syntactic sugar
 - But can directly access enclosing fields and **final** variables
 - Added to language to better support callbacks
 - Syntax an acquired taste?