

 **Abstraction**

 **State**

List Maps: Append to Each

```
abstract class List {
    abstract List appendAll(String s);
}

class Empty extends List {
    Empty() { }
    List appendAll(String s) { return new Empty (); }
}

class Cons extends List {
    Object first;
    List rest;
    Cons(Object first, List rest) {
        this.first = first; this.rest = rest;
    }
    List appendAll(String s) {
        return new Cons(((String)this.first).concat(s),
            this.rest.appendAll(s));
    }
}
```

List Maps: Prefix to Each

```
abstract class List {  
    ...  
    abstract List prefixAll(String s); Copy  
}  
  
class Empty extends List {  
    ...  
    List prefixAll(String s) { return new Empty (); } Copy  
}  
  
class Cons extends List {  
    ...  
    List prefixAll(String s) {  
        return new Cons(s.concat((String)this.first),  
            this.rest.prefixAll(s));  
    } Copy  
}
```

List Maps: Upcasing Each

```
abstract class List {
```

```
    ...
```

```
    abstract List upAll(); Copy
```

```
}
```

```
class Empty extends List {
```

```
    ...
```

```
    List upAll() { return new Empty (); } Copy
```

```
}
```

```
class Cons extends List {
```

```
    ...
```

```
    List upAll() {  
        return new Cons(((String)this.first).toUpperCase(),  
                        this.rest.upAll());
```

```
    }
```

```
}
```

Copy

List Maps: Trimming Each

```
abstract class List {
```

```
    ...
```

```
    abstract List trimAll(); Copy
```

```
}
```

```
class Empty extends List {
```

```
    ...
```

```
    List trimAll() { return new Empty (); } Copy
```

```
}
```

```
class Cons extends List {
```

```
    ...
```

```
    List trimAll() {  
        return new Cons(((String)this.first).trim(),  
                        this.rest.trimAll());
```

```
    }
```

```
}
```

Copy

List Maps

Every time we write a map method, we mostly repeat work:

- Declare an abstract method
- Implement the method in `Empty` to return `new Empty()`
- Implement the method in `Cons`:
 - Do something to `this.first`
 - Recursively call method of `this.rest`
 - Combine with `new Cons(...)`

Can we abstract all of this work?

Generic List Map

```
interface Xformer { Object xform(Object o); }
```

```
abstract class List {  
    abstract List map(Xformer x);  
}
```

```
class Empty extends List {  
    Empty() { }  
    List map(Xformer x) { return new Empty (); }  
}
```

```
class Cons extends List {  
    Object first; List rest;  
    Cons(Object first, List rest) {  
        this.first = first; this.rest = rest;  
    }  
    List map(Xformer x) {  
        return new Cons(x.xform(this.first),  
                        this.rest.map(x));  
    }  
}
```

Using the Generic List Map

```
class Append implements Xformer {
    String s;
    Append(String s) { this.s = s; }
    Object xform(Object o) {
        return ((String)o).concat(this.s);
    }
}
```

[Copy](#)

```
List l = new Cons("a", new Cons("b", new Empty()));
l.map(new Append("x"))
```

```
class Uppcase implements Xformer {
    Uppcase() { }
    Object xform(Object o) {
        return ((String)o).toUpperCase();
    }
}
```

[Copy](#)

```
l.map(new Upper())
```


Anonymous Classes

In full Java, *anonymous classes* make abstraction easier, just like lambda:

```
l.map(new Xformer() {  
    Object xform(Object o) {  
        return ((String)o).toUpperCase();  
    }  
}))
```

➤ **Abstraction**

➤ **State**

State

Java objects encapsulate their fields, and = assigns to a field (in **Advanced Java** and full Java)

```
class Fish {
    double weight;
    Fish(double weight) {
        this.weight = weight;
    }
    double getWeight() {
        return this.weight;
    }
    void feed(double n) {
        this.weight = this.weight + n;
    }
}
```

[Copy](#)

Note: no **return** for a **void** method

State Examples

```
Fish alice = new Fish(7);
```

```
Fish bob = new Fish(6);
```

```
alice.getWeight() → 7
```

```
bob.getWeight() → 6
```

```
alice.feed(3)
```

```
alice.getWeight() → 10
```

```
bob.getWeight() → 6
```

Objects that Contain Lists

Use the constructor to initialize state, even without arguments:

```
class Aq {  
  List fishes;  
  int count;  
  Aq() {  
    this.fishes = new Empty();  
    this.count = 0;  
  }  
  void add(Fish f) {  
    this.fishes = new Cons(f, this.fishes);  
    this.count = this.count + 1;  
  }  
  void feedAll(int n) {  
    this.fishes.map(new Feeder(n));  
  }  
}
```

[Copy](#)

Note: `begin` is implicit

Feeder

```
class Feeder implements Xformer {  
    int n;  
    Feeder(int n) { this.n = n; }  
    Object xform(Object o) {  
        ((Fish)o).feed(this.n);  
        return this; // result will be ignored, anyway  
    }  
}
```

[Copy](#)

State and Abstraction

Of course, we can put colorful fish in our aquarium:

```
class ColorFish extends Fish {  
  String color;  
  ColorFish(double weight, String color) {  
    super(weight);  
    this.color = color;  
  }  
}
```

[Copy](#)

```
Aq a = new Aq();  
a.add(new Fish(10))  
a.add(new ColorFish(11, "blue"))  
a.feedAll(3)  
a → Aq(fishes = Cons(first = ColorFish(weight = 14,  
                                     color = "blue"),  
                    rest = Cons(first = Fish(weight = 13),  
                                rest = Empty()))),  
      count = 2)
```

```
a.add("hello") → contract error
```

Arrays

Java arrays are like Scheme vectors, except that the contract for the array elements is explicit

- The type of an array of x is $x[]$
- To make a $x[]$ with n elements: `new $x[n]$`
- If x is an array, then
 - `$x[n]$` gets its n th element
 - `$x[n] = o$` sets its n th element to o

```
Fish[] v = new Fish[10];  
v[0] = new Fish(2);  
v[0].feed(4);  
v[0] → Fish(weight = 6)
```


null

What about `v[1]` through `v[9]`?

- Java includes a built-in constant `null` that can act as any object type
- Arrays are initialized to have `null` as all elements

`v[4] → null`

`v[4].feed(1) → illegal use of null`

Note that the last example is *not* a contract error

Array Contracts

If you have a `ColorFish`, you can use it as a `Fish`

```
ColorFish charlie = new ColorFish(10, "blue");  
Fish afish = charlie;
```

If you have an array of `ColorFish`, can you use it as an array of `Fish`?

```
Yes: ColorFish[] neons = new ColorFish[10];  
Fish[] fishes = neons;
```

Good:

```
fishes[0] = afish; // which is charlie  
fishes[0].getWeight() → 10  
neons[0].color → "blue"
```

Array Contracts

If you have a `ColorFish`, you can use it as a `Fish`

```
ColorFish charlie = new ColorFish(10, "blue");  
Fish afish = charlie;
```

If you have an array of `ColorFish`, can you use it as an array of `Fish`?

```
Yes: ColorFish[] neons = new ColorFish[10];  
Fish[] fishes = neons;
```

Bad:

```
fishes[0] = new Fish(10);  
neons[0].color → ???
```

Java therefore disallows the assignment dynamically

The Effect of State on Contracts

- At run-time, you can get an *illegal use of null* error
- At run-time, you can get an *illegal array assignment* error

Unlike the problem of using `ListOfObject` instead of `ListOf<X>`, these problems won't go away in future versions of Java