# Functional Objects

Matthias Felleisen
PLT
Northeastern University

# The Myth

- “Objects” represent physical objects.
- Objects encapsulate state.
- Computation means imperative state change through methods or messages.
- OO analysis is natural ... and it naturally leads to OO programing.
- In short, **OO is imperative programming done right on a large scale.**

# My Take

- Object-oriented computation is about the exchange of messages between objects. The purpose is to create objects and to send objects back and forth via messages.

- Class-based programming is about the creation of class hierarchies that specify the nature and behavior of objects during a computation.

# Snyder's Take

- Designers define new classes of objects.
- Objects have operations defined on them.
- Invocations operate on multiple types of objects.
- Class definitions share common components using inheritance.

# The Thesis

Functional  Programming
is **Good**(tm)
for Object-Oriented People.

- State
- Classes
- Sending Messages

# The Nature of the Talk

- Look (again) at some essential elements of OOP/Ls.
- Link them to FP/Ls; refresh your memory.
- Each part has a gem, more proposal than product.
- Perspective: programmer, language designer
- A small talk, squeaking about some basic little things; just good enough for breakfast Kaffee.

# Part I: State

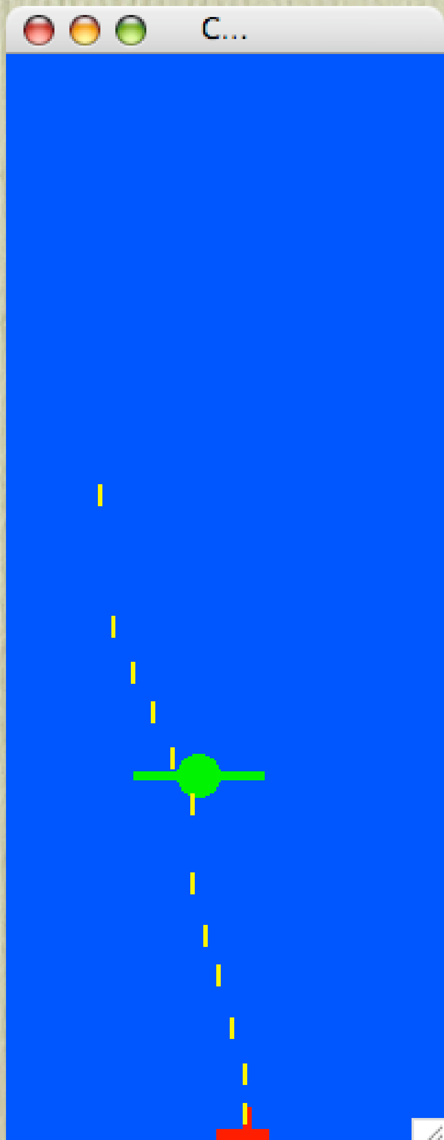
# Quiz: So, who said this?

Though [it] came from many motivations, two were central. ... [T]he small scale one was **to find a more flexible version of assignment, and then to try to eliminate it altogether.**
(1993)

**Favor immutability.**
(2001)

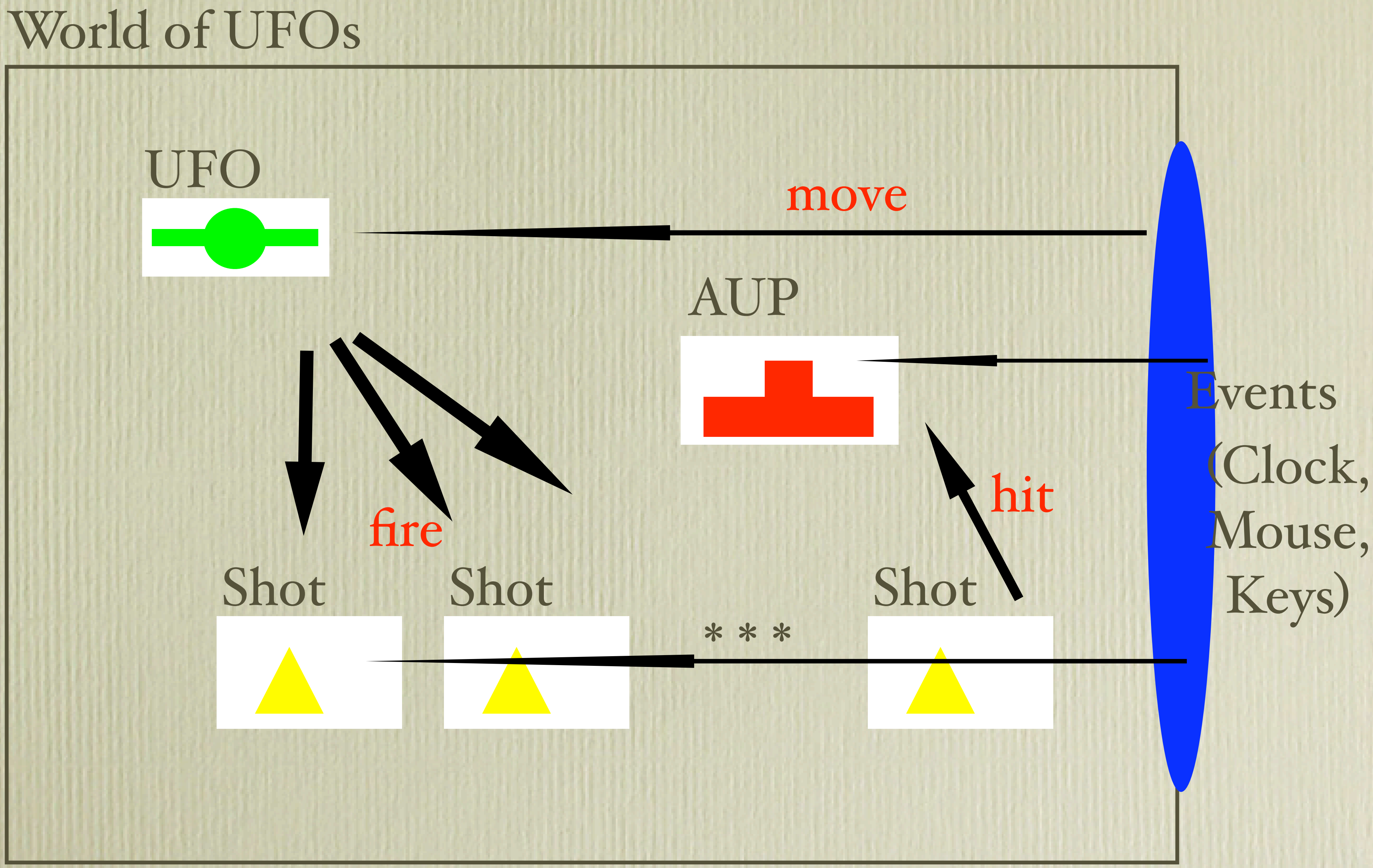
Use **value objects** when possible.
(2001)

# The Problem

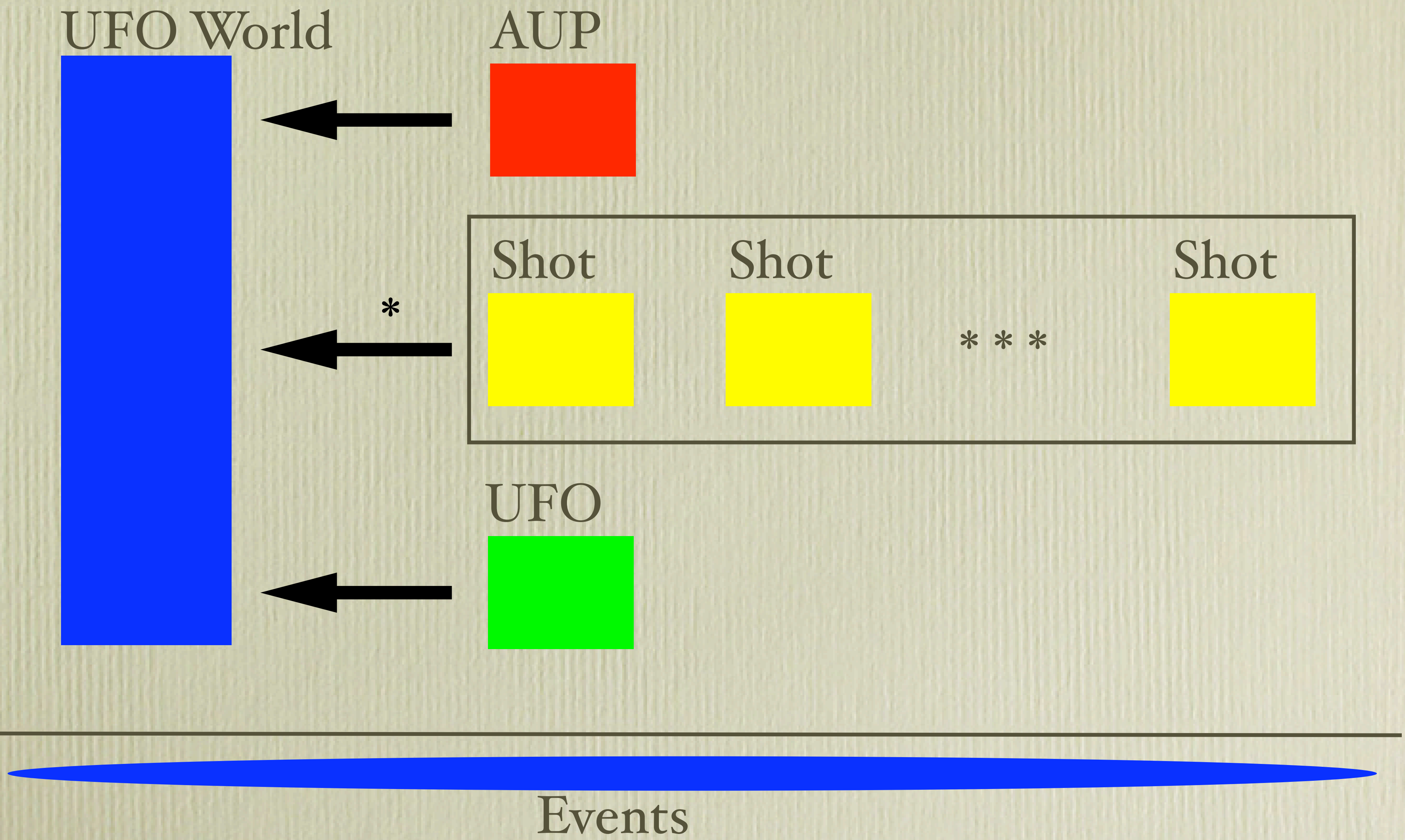



- UFO
- an anti-UFO battery
- a bunch of shots

# OO Analysis


World of UFOs
UFO
move
AUP
Events
(Clock,
Mouse,
Keys)
fire
hit
Shot
Shot
Shot
* * *

# OO Design



# OO Programming

Imperative

```
class UFO {
  int x;
  int y;
  UFO(int x, int y, ... ) {
    this.x = x; ....
  }
  ...
  void move() {
    x = x + ran(deltaX);
    y = y + deltaY;
  }
}
```

Functional

```
class UFO {
  int x;
  int y;
  UFO(int x, int y, ... ) {
    this.x = x; ....
  }
  ...
  UFO move() {
    new UFO(ran(deltaX),
                   y + deltaY);
  }
}
```

# Oh no, the old movable Point is back

- This is just the stupid movable point.
- Every OO model talk contains it.
- It won’t scale.
- Anyways, where does the new UFO go?
- And how can a clock callback use it?

# The Callback Problem (1)

```
class UFOWorld extends World {
  UFO u;
  AUP a;   . . .
  void onClockTick() {
      u.move();
      . . .
  }
  void onKeyClick(Key k) {
      a.move(k);
      . . .
  }  . . .
}
```

# The Callback Problem (1)

```
class UFOWorld  extends World {
  UFO u;
  AUP a;  . . .
  void onClockTick() {
      u.move();
      . . .
  }
  void onKeyClick(Key k) {
      a.move(k);
      . . .
  } . . .
}
```

# The Callback Problem (2)

```
class UFOWorld  extends World {
  UFO u;
  AUP a;  . . .
  void onClockTick() {
      u = u.move();
      . . .
  }
  void onKeyClick(Key k) {
      a = a.move(k);
      . . .
  } . . .
}
```

Not A Solution

# The Callback Problem (3)

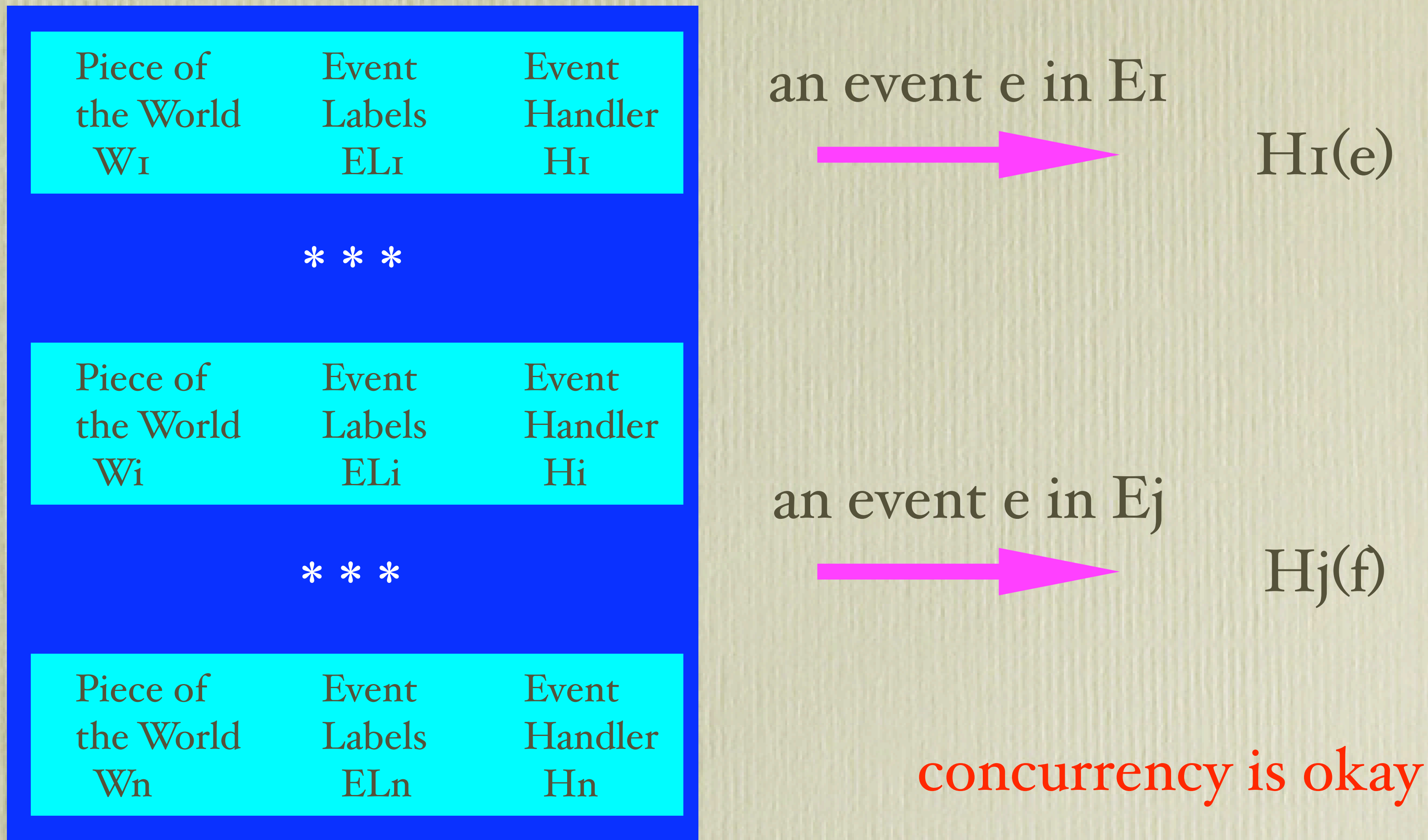
```
class UFOWorld  extends World {
  UFO u;
  AUP a;  . . .
  World onClockTick() {
     return new UFOWorld(u.move(), . . . );
     . . .
  }
  World onKeyClick(Key k) {
     return new UFOWorld(. . ., a.move(k));
     . . .
  } . . .
}
```

# The Callback Solution

```
class World {
  World theWorld;
  . . .
  abstract World onClockTick() ;
  abstract World onKeyClick(Key k);
  . . .
    eventHandler( . . .) {
      theWorld =
        . . . theWorld.onClickTick() . . .
        . . . theWorld.onKeyClick(k) . . .
    }
}
```

... and even this one assignment can disappear if you create a “world loop”.

# Events and Pieces of the World




disjoint sets of events, worlds

# State: It doesn't have to be imperative

- reduce imperativeness, it's good for you
  (see ML and Scheme)
- explicates channels of communication
- enables more abstraction, which means less cost
- renders concurrency manageable
- conduct research on this programming style
  (feasibility, clarity, time and space efficiency)

in 17 ECOOPs and 18 OOSPLAs, only three papers on declarative methods and class hierarchies appeared

# Quiz: So, one more time, who said this?

Though [it] came from many motivations, two were central. ... [T]he small scale one was **to find a more flexible version of assignment, and then to try to eliminate it altogether.**
(1993)

**Favor immutability.**
(2001)

Use **value objects** when possible.
(2001)

# OOP: The Experts

Though OOP came from many motivations, two were central. ... [T]he small scale one was **to find a more flexible version of assignment, and then to try to eliminate it altogether.**

Alan Kay,
*History of Smalltalk* (1993)

**Favor immutability.**

Joshua Bloch,
*Effective Java* (2001)

Use **value objects** when possible.

Kent Beck,
*Test Driven Development* (2001)

# Part II: Classes

# OOP: The Experts, Again

Though OOP came from many motivations, two were central. The large scale one was to find **a better module scheme for complex systems** involving hiding of details

Alan Kay, *History of Smalltalk* (1993)

**A class is a module** with its own external interface.

Alan Snyder, *Encapsulation and Inheritance* (1986)

# The One Slide Version

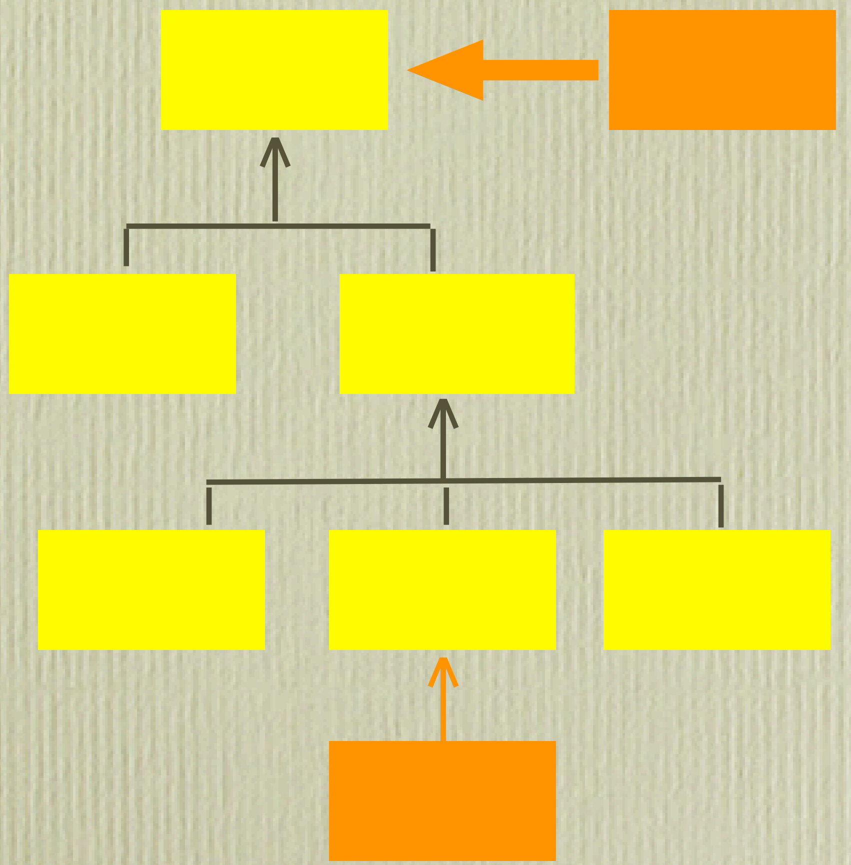
The challenge for language designers is to provide the means by which the designer of a class can express an interface to inheriting clients that reveals the minimum information needed to use the class.

Alan Snyder, *Encapsulation and Inheritance* (1985)

**You must override hashCode in every class that overrides equals.**

Joshua Bloch, *Effective Java* (2001)

# Comparative Semantics: OOP vs FP



**OO programming and computation**

$\vdash$ main( ... ) → s0→ s1 ...

# Comparative Semantics: OOP vs FP

**fun** f(x) = ... x ... g(... x ...)

**fun** g(x,y) = ... h(x) ... y ... f(y) ...

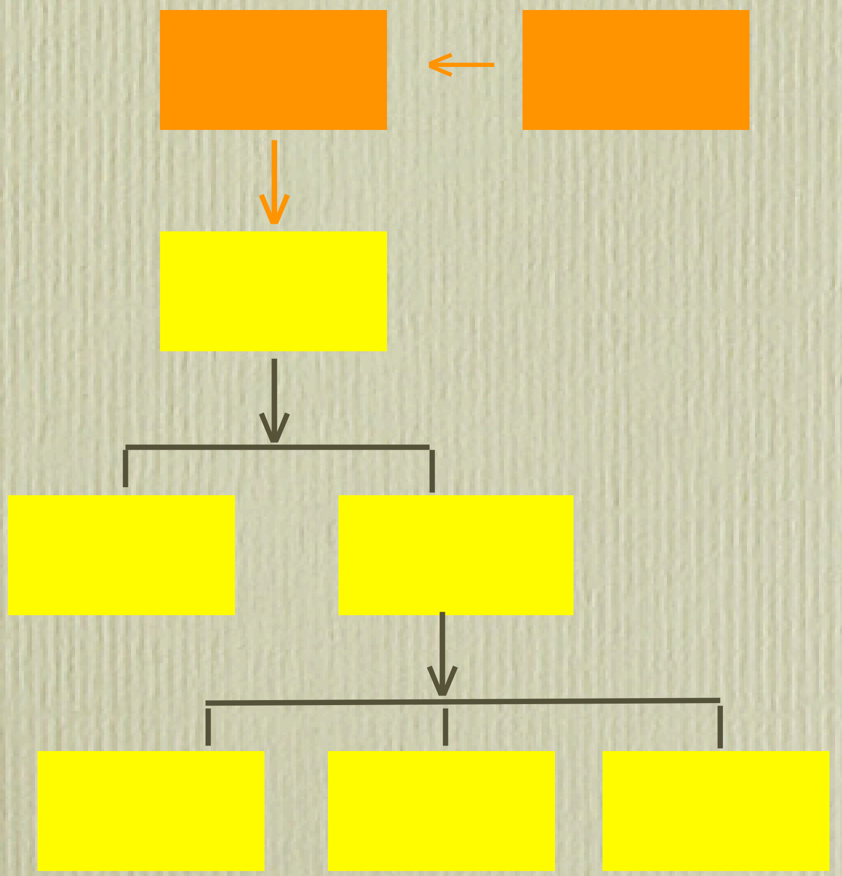
**fun** h(z,x) = fn x => ... g(z,z) ...

**fun** main(argv []) =
    ... h(argv[0],argv[1])f(2) ...

**FP programming and computation (naive version)**

$\vdash$ main( ... ) → s0→ s1 ...

# Comparative Semantics: OOP vs FP

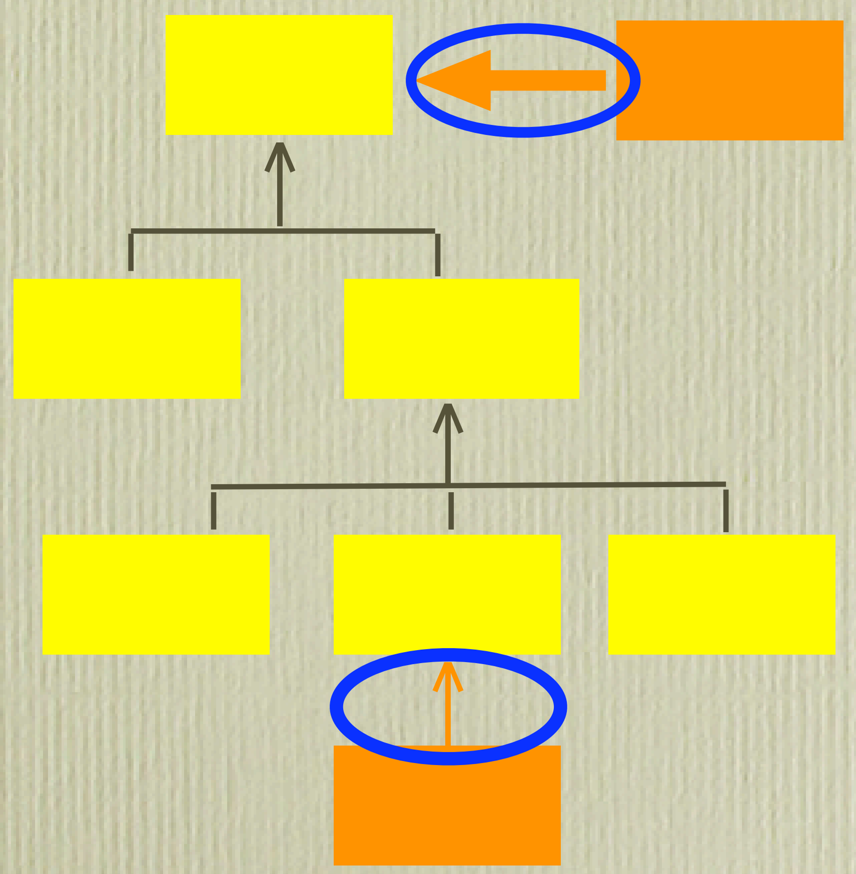


**FP programming and computation (realistic version)**

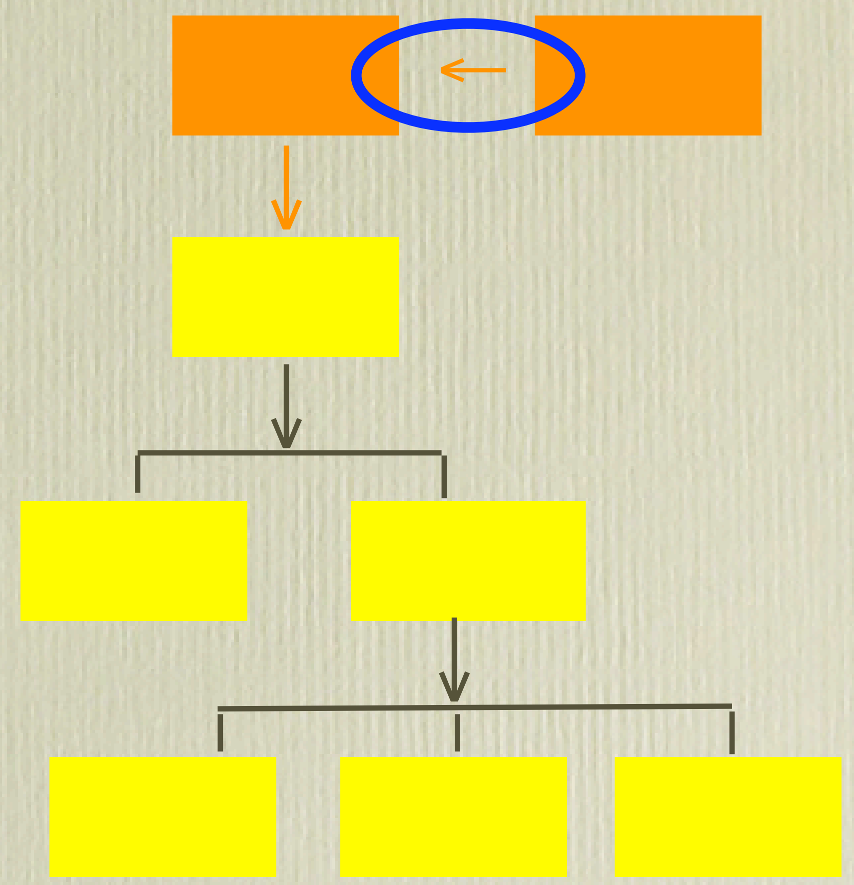
$\vdash$ main( ... ) → s0→ s1 ...

# Side by Side

OOP



FP



# FP: What's a Module

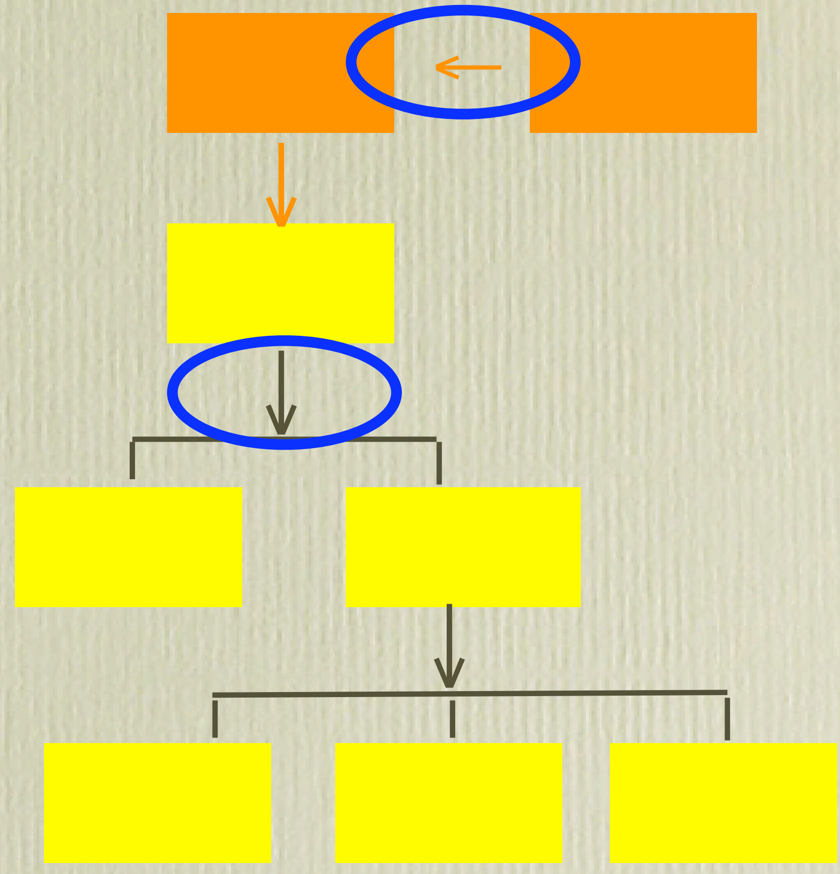
- namespaces, packages, and so on
- abstract data type (existential type, abstype)
- structure (SML module)
- functors: modules are first-class (link time) values
- applicative vs generative functors
- mutually recursive functors (units)

# FP: Encapsulation

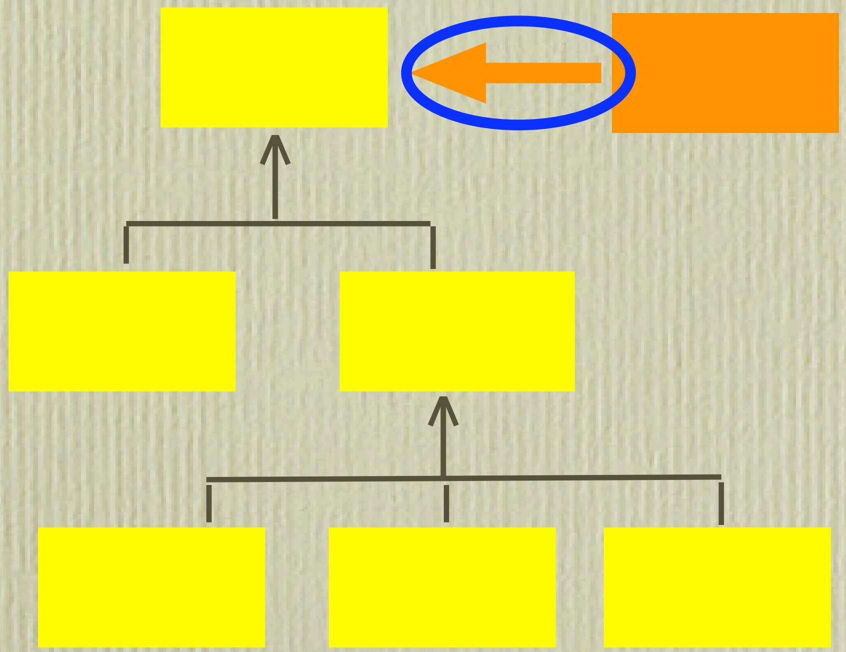
- What is information encapsulation? Are modules (1) opaque, (2) transparent, or (3) translucent?
- How do you reveal information? Type equations. Structure equations.
- How do you use revealed information? Sharing constraints.
- When do modules implement interfaces? Can clients thin the interfaces?

# FP Research

Look at POPL or LFP/ICFP proceedings and count the papers on "questions" of moduleness.



# OOP: The Client Relationship



- **private, public, protected, …**
- **static**
- **implements:** as in Java

# OOP: The Client Interface

- Gang of Four: Program to the Interface.
  Types are interfaces for fields, method signatures, and variables.
- Good: This practice passes the "rename the fields" test.
- Not so good: It doesn't pass the "rename the method" test.
- Bad: Reality is, you can always get to the class.

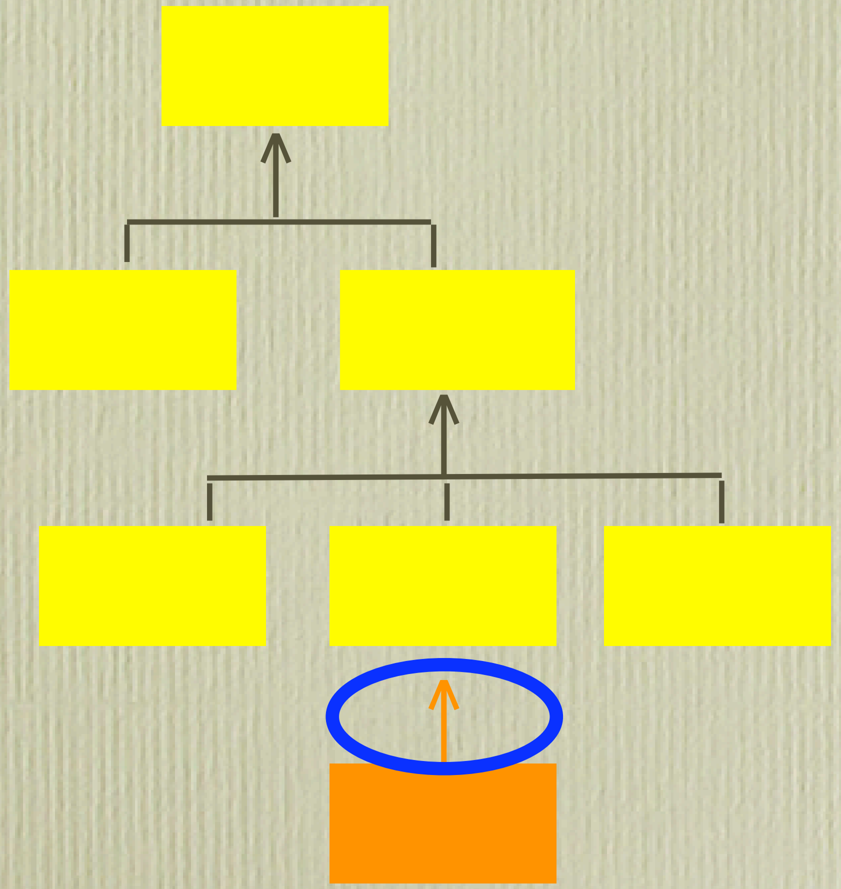
# OOP: The Client Interface

"A programming language supports encapsulation to the degree that it allows *minimal* external interfaces ... [if you can get around this] the original language is still defective."

Alan Snyder, *Encapsulation and Inheritance (1985)*

But, let's leave it at that. -- Me, now

# OOP: The Superclass Relationship



- **private, public, protected, ...**
- **static**
- **final** (good something new)
- **inner** (but only in one OOPL)

The challenge for language designers is to provide the means by which the designer of a class can express an interface to inheriting clients that reveals the minimum information needed to use the class.

# OOP: Modules from Subclasses

```
class Object {
  ...
  public boolean equals(Object o) { ... }
  public int hashCode() { ... }
```

```
class Address {
  public boolean equals(Object o) { ... }
  public int hashCode() { ... }
  ...
}
```

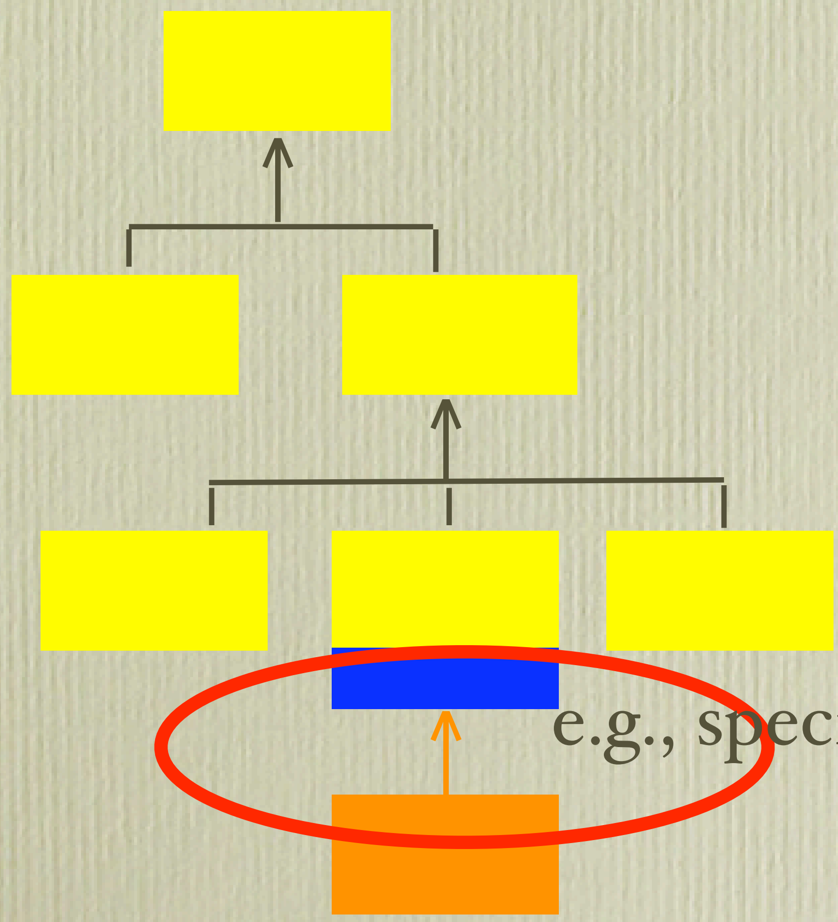
**Override hashCode in every class that overrides equals.**

**Josh Bloch, *Effective Java***

BUG!

# OOP: What's an "Inheritance Module"

Solution 1: specialization interfaces



State and Guttag '95, Lamping 93, Hauck 93

e.g., specify simultaneous override

# OOP: Specialization Interfaces

```
class Object {
  ...
  public boolean equals(Object o) { ... }
  public int hashCode() { ... }
```

**Override hashCode in every class that overrides equals.**

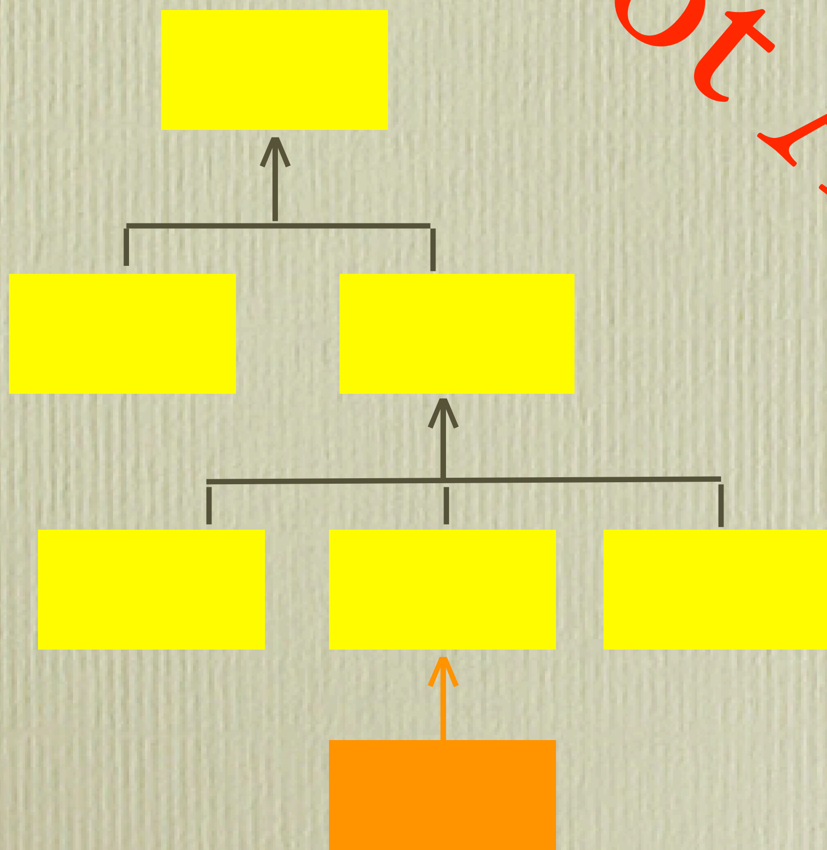
```
class Address {
  public boolean equals(Object o) { ... }
  public int hashCode() { ... }
  ...
}
```

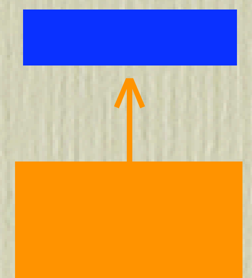
Sadly enough, nobody has implemented this solution and explored it.

# OOP: What's an "Inheritance Module"

Solution 2: mixins

Not A Solution





Mixins are class (fragments) w/o a superclass --- they describe their superclass via an interface.

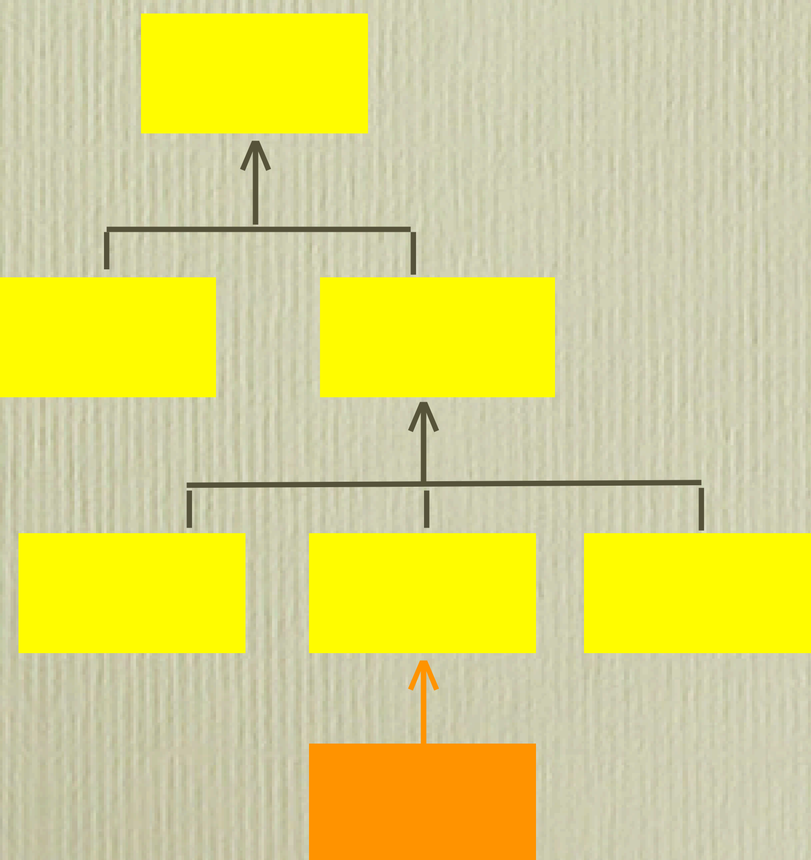
MixedJava (Flatt, Krish., Felleisen '98)
Jiazzi (Hsieh and Flatt '01)
Jam (Anaconad and Zucca '01)
Java 1.5 (Sun '04
and a few more

# Inheritance Modules: Mixins

- Mixins specify what they expect from their superclass. That's important.
- But it does not specify what a superclass expects from its subclass.
- **The relationship is inverted.**

# OOP: What's an "Inheritance Module"

Solution 3: classes as values **and functions**



# OOP: Classes and Functions

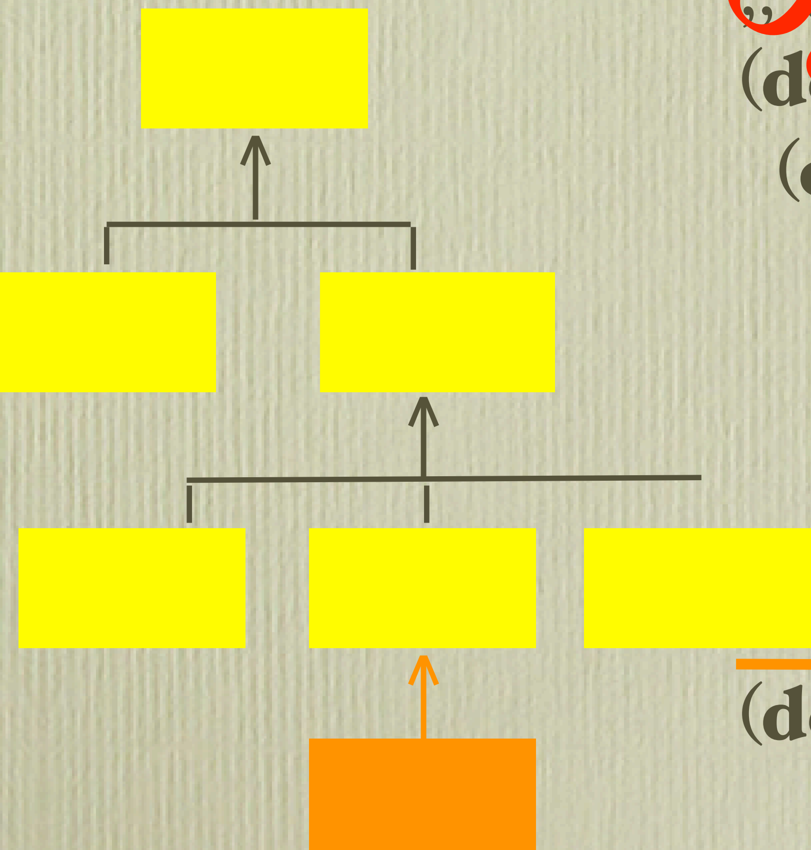
PLT Scheme, Flatt et al (1998-2004)

```
;; object% :: object<%>
(define object% (class ...))

;; (object% object% -> bool)
;;  (-> int)
;;  -> object%
(define (extend-object f g)
  (class object%
     (super-new)
     (define/override (equals o)
       (f this o))
     (define/override (hashCode)
       (g))))
```

```
;; pre-addr% :: object<%>
(define pre-addr%
    (extend-object
        (λ (this that) ...) (λ () ...)))

;; addr% :: address<%>
(define addr%
   (class pre-addr% ...))
```

# OOP: What's an "Inheritance Module"

Solution 3: classes as values **and functions**



```
;; (object% object% -> bool) (-> int) -> object%
(define (extend-object f g)
  (class object%
     (super-new)
     (define/override (equals o)
       (f this o))
     (define/override (hashCode)
       (g))))
(define pre-addr%
    (extend-object
       (λ (this that) ...) (λ () ...)))
```

PLT Scheme, Flatt et al (1998-2004)

Not Quite Right

# Inheritance Modules: First-class Classes

- First-class classes solve the problem, if we also have functions.
- But, if there are many constraints, we need an enormous number of functions to account for all possible combinations.
- Plus first-class classes come at a significant cost.
- **So, they are not a feasible solution either.**

# Inheritance Modules: An FP Approach

**class** Object
**implements** IHashable ... {
...
boolean *f*(Object that) ...
int g() ...
**export** *f* **as** equals,
*g* **as** hashCode;
}

$\models$

**interface** IHashable {
boolean equals(Object o)
int hashCode()
}

In short, separate naming from exporting as in, for example, PLT Scheme modules.

# Inheritance Modules: An FP Approach

```
class Address like Object {


  boolean f(Object that) ...
  int g() ...
  ...
  boolean h(Object that)
  ...



}
```

```
interface IHashable {
  boolean equals(Object o)
  int hashCode()
}
```

As is, Address does **not** satisfy the IHashable **interface!**

# Inheritance Modules: An FP Approach

```
class Address like Object
   implements IHashable ... {

 boolean f(Object that) ...
 int g() ...

 ...
 boolean h(Object that)
 ...

  export h as equals,
           g as hashCode;

}
```

⊨

```
interface IHashable {
 boolean equals(Object o)
 int hashCode()
}
```

- implementation inheritance, yes
- implicit subtyping, no overriding, no
- instead: **explicit export**

# Inheritance as Modules

- Inherit, don't subtype; inherit, don't override; specify **implements** separately and explicitly
- Good: satisfies the "rename variables" test
- Better: satisfies the "rename methods" test, too.
- Best: more work on ML-style modules applies.

And it's all just some basic functional-modular ideas.

# End Note: On Classes and Modules

- Clements Szyperski, Import is Not Inheritance - Why We Need Both: Modules and Classes
- Yes: Remy and Leroy, OCAML. Many **ICFP** papers.
- Yes: Findler and Flatt, Modular Object-Oriented Programming **ICFP 1998**

Good: Schaerli, Ducasse, Nierstrazs, Wuys, **ECOOP 2004**

# Part III: Sending Messages

# We already know that ...

- GoF, Design Patterns, 1994

Peter Norvig found that 16 of the 23 patterns in Design Patterns were "invisible or simpler" in Lisp.

- Thomas Kühne, A functional pattern system for object-oriented design, Darmstadt 1998

“A functional pattern system is valuable ... for object-oriented design.” [p261]

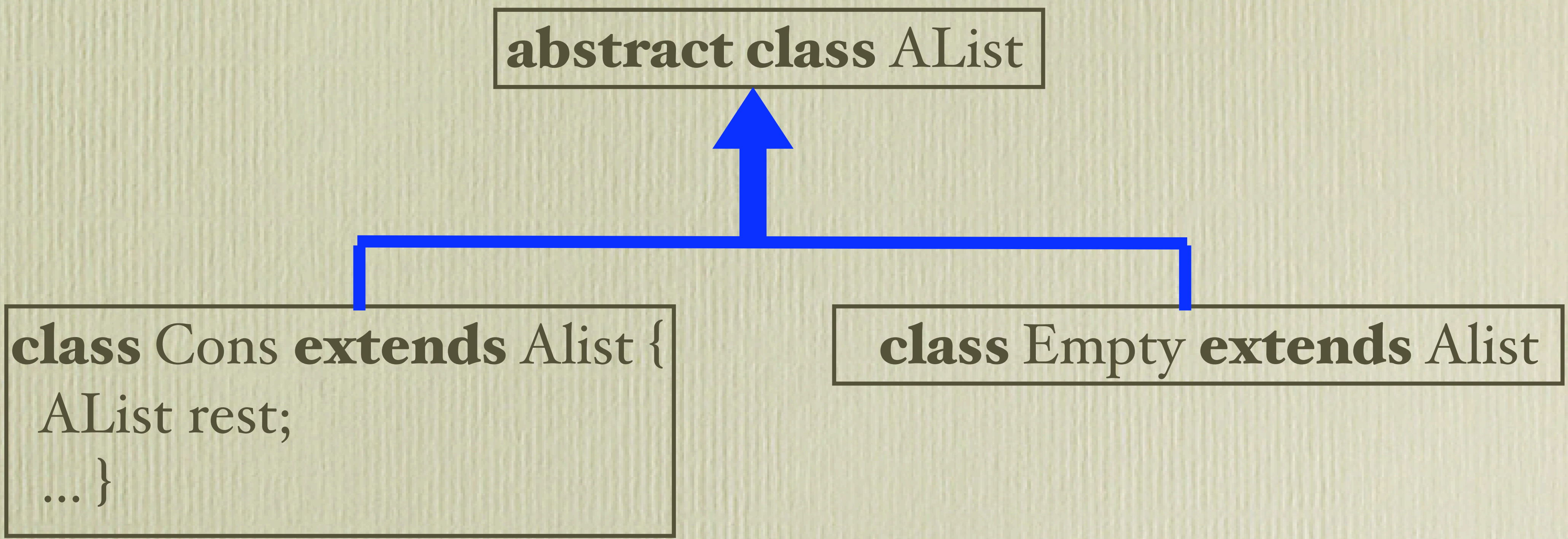
- Joshua Bloch, Effective Java, 2001

The majority of method and class advice points to functional programming.

# Implementing Unions, An Example

- GoF: Composite lets clients treat individual objects and compositions of objects uniformly.
- Kühne: Raise Nil to a first-class value
- Bloch: Replace Union with Class Hierarchies

# Let's Follow This Advice: Classes




Use a class hierarchy and "null" objects
to represent the union type
list = cons + nil

# Let's Follow This Advice: Methods

```
abstract class AList {
  int length() { return howMany(0);  }
  abstract int howMany(int a);
  ... }
```

```
class Cons extends Alist {
  AList rest;
  int howMany(int a) {
    return rest.howMany(a+1);
  }
  ... }
```

```
class Empty extends Alist {
  int howMany(int a) {
    return 0;
  }
  ... }
```

Object-oriented programming is about
sending messages to objects (invoking methods).

# Let's Follow This Advice: Test

```
class Test {
  boolean main(int n) {
    AList last = new Empty();
    ...
    // create list with n Cons'es
    return last.howMany() == n;
  }
}
```

Compile, link, run: what happens?

# Let's Follow This Advice: Guess again

- Test.main(10) works just fine
- Test.main(100000)

[:Web/Presentations/Ecoop] matthias% java Test
Exception in thread "main" java.lang.StackOverflowError

C#, C++ [*], CLOS, Eiffel, and so on, ... **don't** run the programs when we follow the guidelines of OO programming.

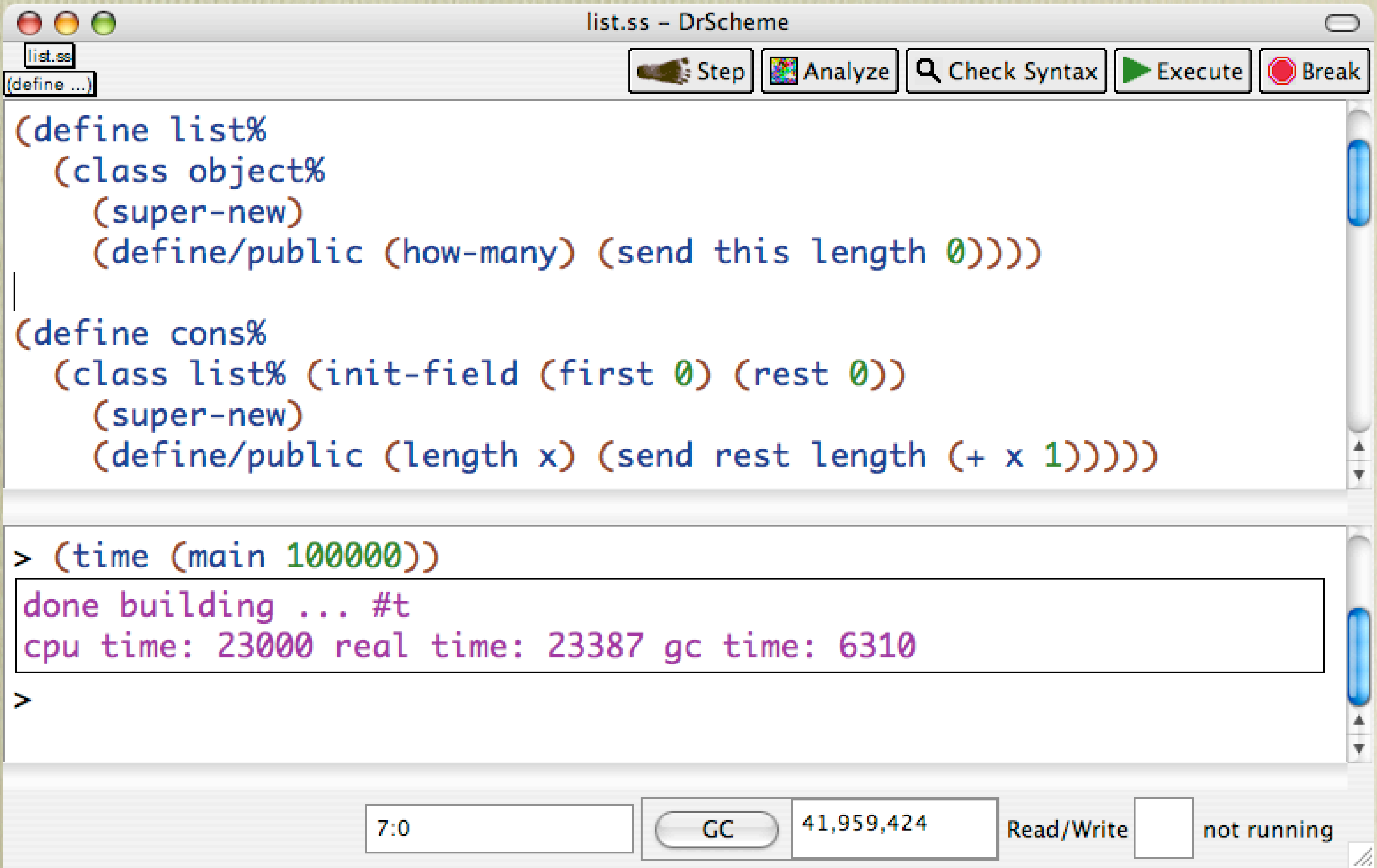
# Loops to the Rescue

```
abstract class AList {
 int howMany0() {
  int i = 0;
  for(AList l = this; !(a instanceof Empty); l = ((Cons)l).rest)
    i = i + 1;
  return i;
 } ...
}
```

We must use non-OO means to produce working code.

Object-Oriented Programming in languages that don't require tail-call optimizations makes no sense.

# Scheme's Methods Can Do It



# How Come Schemers have it Right?

- Scheme's method invocation is a procedure call.
- Scheme implementations must optimize tail-calls.
- Because Gerry and Guy were omniscient ...

# How Come Schemers have it Right?

Nah,

Guy in email to me, cc'ed Gerry on April 2, 2004:

"We decided to construct a toy implementation of an actor language so that we could play with it ...

Then came a crucial discovery. Once we got the interpreter working correctly and had played with it for a while, writing small actors programs, we were astonished to discover that that the program fragments in _apply_ that implemented function application and actor invocation were identical!"

# How Come Schemers have it Right?

Nah,

**Schemers have it right because they followed the pure OO example.**

# Part IV: More

# OOP, FP, Multiparadigm Programming

- Budd, Leda (multiparadigm)
- Remy and Leroy: OCAML
- Odersky: Pizza, GJ, Scala
- MPOOL: [caution]

# More for OOP from FP

- A Class System from Macros: Matthew Flatt, PLT Scheme
- A Contract System for Objects from FP
  Robert Findler, PLT Scheme
- Teaching OO Programming -- with Functions First  PLT for eight years now

# OOP from Scheme Macros

- PLT Scheme's classes and mixin system is more expressive than Java's.
- It's all implemented with macros, specifically, 2257 lines of (functional) macro code.
- Because this OO implementation is that small, Flatt can experiment easily with different variants of classes.

# More for OOP from FP

- A Class System from Macros: Matthew Flatt, PLT Scheme
- A Contract System for Objects from FP Robert Findler, PLT Scheme
- Teaching OO Programming -- with Functions First  PLT for eight years now

# OO Contracts from Scheme

- DrScheme is a large code base with 100's of small components that exchange higher-order functions.
- Software contracts are essential to keep these components sane.
- Findler & Felleisen ICFP 2002 shows how to cope with infinite behavior in a software contract context.
- Findler now carries over this work to OOP because objects also have infinite behavior.

# More for OOP from FP

- A Class System from Macros: Matthew Flatt, PLT Scheme
- A Contract System for Objects from FP
  Robert Findler, PLT Scheme
- Teaching OO Programming -- with Functions First  PLT for eight years now

# Teaching Good OOP Requires FP

- TeachScheme!'s design recipe approach organizes functional programs around the structure of data and collections of functions.
- It **naturally** leads to OO programming in the follow-up course.
- Experience shows time and again
  - 1 year of Java (or C++) is
  - **inferior to** 1 semester of TeachScheme! followed by 1 semester of Java

# Part V: Conclusion

# FP and OOP

- FP has benefited from OOP for a long time.
- OOP could benefit from FP.
- Go back to your roots and let's work together.

# The End

Doing encapsulation right is a commitment not just to abstraction of state, but to eliminate state oriented metaphors from programming.

Alan Kay, *Early History of Smalltalk*

Thanks to

Matthew Flatt
Robby Findler
Shriram Krishnamurthi
Dan Friedman