How to Design Class Hierarchies

Matthias Felleisen
PLT
Northeastern University, Boston

The Team

Robert Findler
Matthew Flatt
Shriram Krishnamurthi

Kathy Gray
Viera Proulx

Thesis

Teaching FP first is the ideal introduction to OOP.

A first year that teaches one semester of FP followed by another semester of OOP produces the best OO programmers.

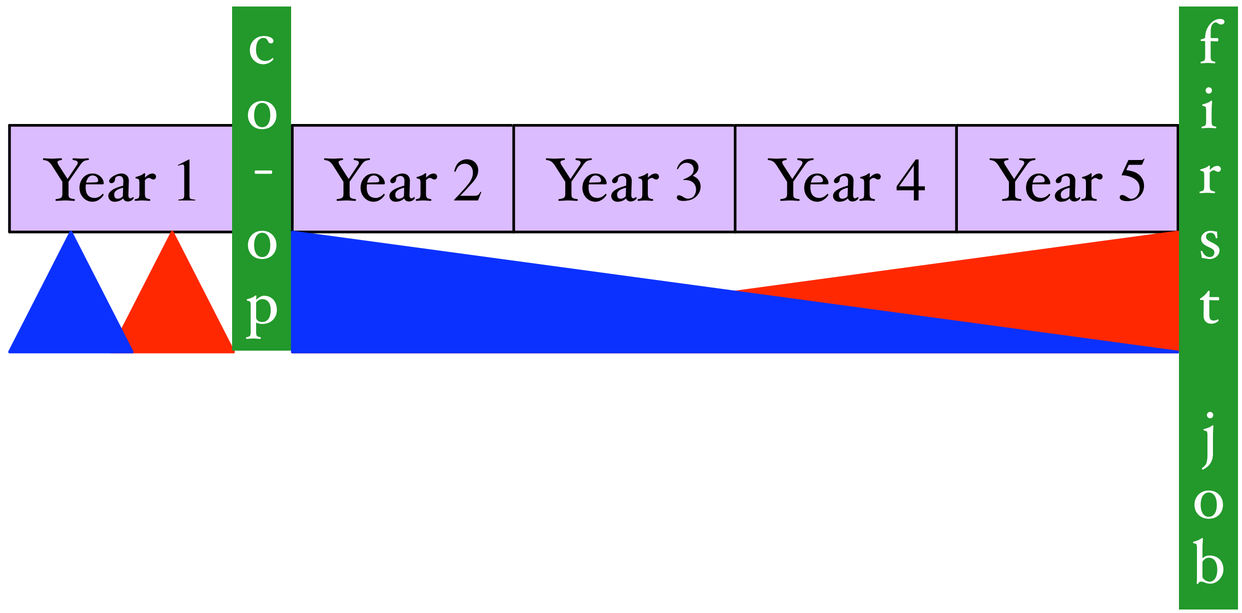
Background

How to Design Programs

- 10 years of FP in first year & high schools
- "Structure & Interpretation of the Computer Science Curriculum" [FDPE'03, JFP'04]
- building the "bridge" to the "real world"

Aimed at what's best for students in the long run

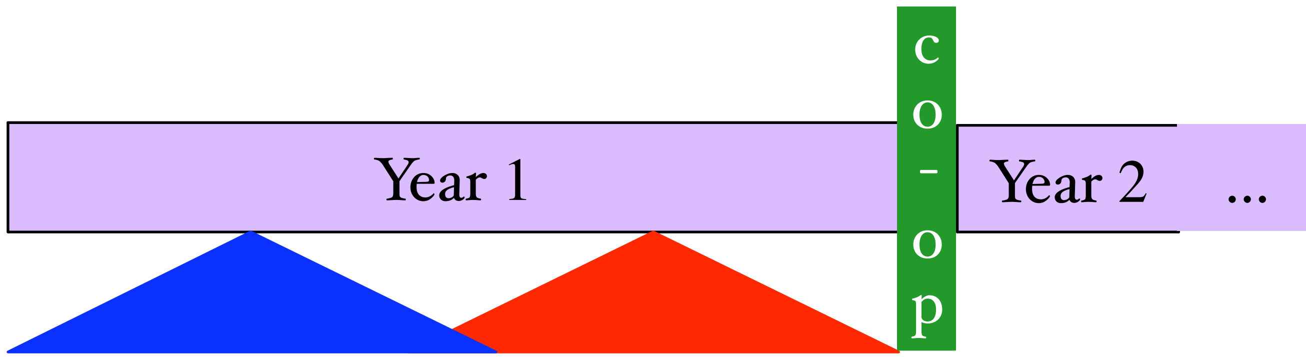
College Timeline



Principles

Preparation for Industry

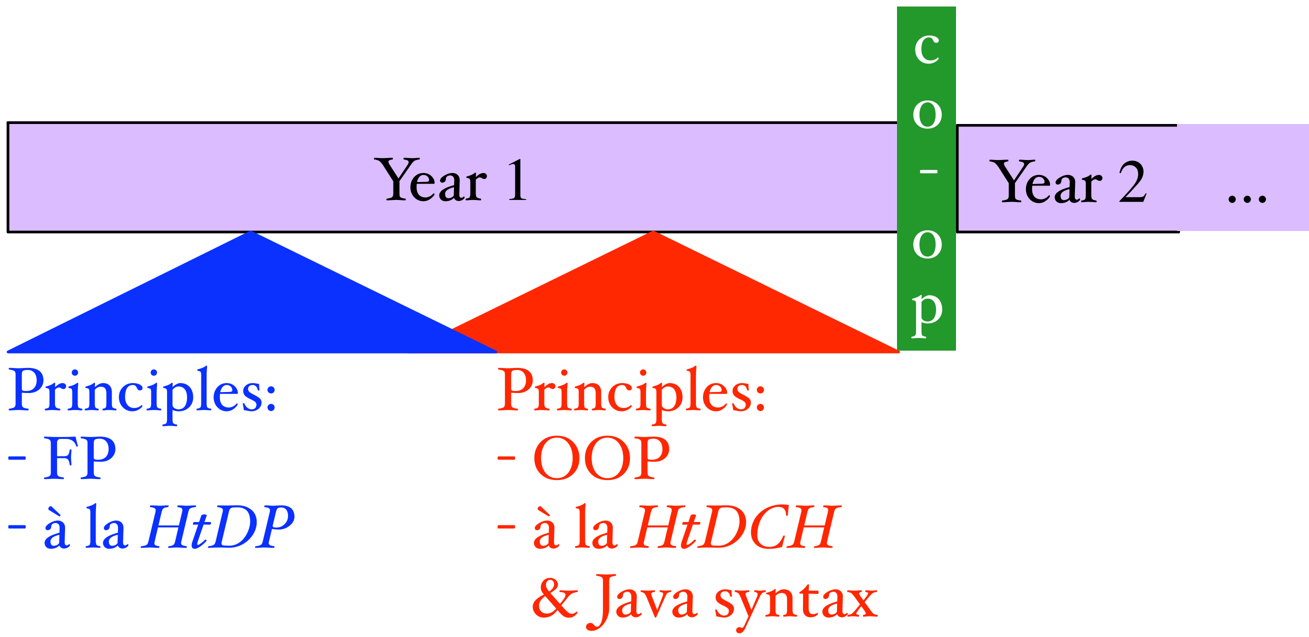
The First Year



Principles:
- computation
- programming
- easy!

Principles for OO:
- computation
- programming
- relevant

The First Year



Part I: What is OOP? — A Quiz

Or, why you should believe that FP, our wonderful “church,” and OOP, the essence of evil “state,” can have anything to do with each other
(from a 2004 conference)

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)

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.**

Alan Kay, *History of Smalltalk* (1993)

Who Said This?

Favor immutability.

(2001)

Who Said This?

Favor immutability.

Joshua Bloch, *Effective Java* (2001)

Who Said This?

Use value objects when possible.

(2001)

Who Said This?

Use value objects when possible.

Kent Beck, *Test Driven Development* (2001)

Who Said This?

It is unfortunate that much of what is called "object-oriented programming" today is simply old style programming with fancier constructs. Many programs are loaded with "assignment-style" operations now done by more expensive attached procedures.

(1993)

Who Said This?

It is unfortunate that much of what is called "object-oriented programming" today is simply old style programming with fancier constructs. Many programs are loaded with "assignment-style" operations now done by more expensive attached procedures.

Alan Kay, *History of Smalltalk* (1993)

Part II: How to Design Programs

Or, how should you teach FP in the first semester

Functional Programming is Obvious

```
(define-datatype tree tree?
  [Leaf (value integer?)]
  [Node (left tree?) (right tree?)])

; tree → integer
; add up the numbers in the given tree
(define (sum t)
  (cases tree t
    [Leaf   (v)   v]
    [Node (l r)   (+ (sum l) (sum r))]))
```

EoPL (2e) Scheme

Functional Programming is Obvious

(even for the parenthetically challenged)

```
type tree =
   Leaf of int
 | Node of tree * tree

(* tree → integer
   add up the numbers in the given tree *)
let rec sum(t) =
  match t with
     Leaf(i) -> i
   | Node(l,r) -> sum(l) + sum(r)
```

OCaml

Is It Really for the Novice?

- What does a novice take away from this?
- What carries over to the industrial PLs (Java++, Perl)?
- Recursion? Types? Interactive exploration (repl)?
- How to write an interpreter?
- ... for a language you aren't allowed to use?

A Great Novice Experience

- "I learned solid principles of programming."
- "I know that this is useful, even if I didn't 'use' it."
- "Everything I learned after that is more popular, but not even remotely as convenient or powerful."
- "It changed my life."

How HtDP Produces Results

HtDP teaches

- The Design Recipe: data, data, data
- Abstraction as editing
- Organizing programs

The Design Recipe

1 Data representation
2 Purpose & contract
3 Functional examples
4 Template (inventory)
5 Body implementation
6 Test (examples)

Write a program to check whether an ant is at home.... An ant has a weight and a location in the zoo.... Home is the cartesian origin.

1. Data representation
2. Purpose & contract
3. Functional examples
4. Template (inventory)
5. Body implementation
6. Test (examples)

Write a program to check whether an ant is at home.... An ant has a weight and a location in the zoo.... Home is the cartesian origin.

1 Data representation

2 Purpose & contract

3 Functional examples

4 Template (inventory)

5 Body implementation

6 Test (examples)

“Real world”: **information**

“Computational world”: **representations**

Programmer’s job: pick a representation

⇒ determines rest of recipe

Write a program to check whether an ant is at home.... An ant has a weight and a location in the zoo.... Home is the cartesian origin.

	atomic data	intervals / enumerations	structs	unions	...
1 Data representation					
2 Purpose & contract					
3 Functional examples					
4 Template (inventory)					
5 Body implementation					
6 Test (examples)					

Write a program to check whether an ant is at home.... An ant has a weight and a location in the zoo.... Home is the cartesian origin.

1 Data representation

2 Purpose & contract

3 Functional examples

4 Template (inventory)

5 Body implementation

6 Test (examples)

```
; A posn is
;   (make-posn num num)
(define-struct posn (x y))

; An ant is
;   (make-ant num posn)
(define-struct ant (weight loc))

(make-ant 0.001 (make-posn 4 5))
(make-ant 0.007 (make-posn 3 17))
```

```
; An ant is
;   (make-ant num posn)
(define-struct ant (weight loc))

; ant-at-home? : ant -> bool
```

1 Data representation

2 Purpose & contract

3 Functional examples

4 Template (inventory)

5 Body implementation

6 Test (examples)

```
; An ant is
;   (make-ant num posn)
(define-struct ant (weight loc))

; ant-at-home? : ant -> bool
; Check whether ant a is home
```

1 Data representation

2 Purpose & contract

3 Functional examples

4 Template (inventory)

5 Body implementation

6 Test (examples)

1 Data representation

2 Purpose & contract

3 Functional examples

4 Template (inventory)

5 Body implementation

6 Test (examples)

```
; An ant is
;   (make-ant num posn)
(define-struct ant (weight loc))

; ant-at-home? : ant -> bool
; Check whether ant a is home
(define (ant-at-home? a)
  ...)
```

1 Data representation

2 Purpose & contract

3 Functional examples

4 Template (inventory)

5 Body implementation

6 Test (examples)

```
; An ant is
;   (make-ant num posn)
(define-struct ant (weight loc))

; ant-at-home? : ant -> bool
; Check whether ant a is home
(define (ant-at-home? a)
  ...)
```

```
(ant-at-home? (make-ant 0.001 (make-posn 0 0)))
"should be" true
(ant-at-home? (make-ant 0.001 (make-posn 1 1)))
"should be" false
```

1 Data representation

2 Purpose & contract

3 Functional examples

4 Template (inventory)

5 Body implementation

6 Test (examples)

```
; An ant is
;   (make-ant num posn)
(define-struct ant (weight loc))

; ant-at-home? : ant -> bool
; Check whether ant a is home
(define (ant-at-home? a)
  ... (ant-weight a)
  ... (ant-loc a) ...)
```

```
(ant-at-home? (make-ant 0.001 (make-posn 0 0)))
"should be" true
(ant-at-home? (make-ant 0.001 (make-posn 1 1)))
"should be" false
```

```
; An ant is
;   (make-ant num posn)
(define-struct ant (weight loc))
```

1 Data representation

2 Purpose & contract

3 Functional examples

4 Template (inventory)

5 Body implementation

6 Test (examples)

```
; ant-at-home? : ant -> bool
; Check whether ant a is home
(define (ant-at-home? a)
  ... (ant-weight a)
  ... (posn-at-home? (ant-loc a)) ...)
```

data reference ⇒ template reference

Add templates for referenced data, if needed, and implement body for referenced data

```
(ant-at-home? (make-ant 0.001 (make-posn 0 0)))
"should be" true
(ant-at-home? (make-ant 0.001 (make-posn 1 1)))
"should be" false
```

1 Data representation

2 Purpose & contract

3 Functional examples

4 Template (inventory)

5 Body implementation

6 Test (examples)

```
; An ant is
;   (make-ant num posn)
(define-struct ant (weight loc))

; ant-at-home? : ant -> bool
; Check whether ant a is home
(define (ant-at-home? a)
  ... (ant-weight a)
  ... (posn-at-home? (ant-loc a)) ...)


(define (posn-at-home? p)
  ... (posn-x p) ... (posn-y p) ...)
```

```
(ant-at-home? (make-ant 0.001 (make-posn 0 0)))
"should be" true
(ant-at-home? (make-ant 0.001 (make-posn 1 1)))
"should be" false
```

1 Data representation

2 Purpose & contract

3 Functional examples

4 Template (inventory)

5 Body implementation

6 Test (examples)

```
; An ant is
;   (make-ant num posn)
(define-struct ant (weight loc))

; ant-at-home? : ant -> bool
; Check whether ant a is home
;
; (define (ant-at-home? a)
;    ... (ant-weight a)
;    ... (posn-at-home? (ant-loc a)) ...)
; (define (posn-at-home? p)
;    ... (posn-x p) ... (posn-y p) ...)
;
(define (ant-at-home? a)
  (posn-at-home? (ant-loc a)))
(define (posn-at-home? p)
  (and (= (posn-x p) 0) (= (posn-y p) 0)))

(ant-at-home? (make-ant 0.001 (make-posn 0 0)))
"should be" true
(ant-at-home? (make-ant 0.001 (make-posn 1 1)))
"should be" false
```

1 Data representation

2 Purpose & contract

3 Functional examples

4 Template (inventory)

5 Body implementation

6 Test (examples)

```
; An ant is
;   (make-ant num posn)
(define-struct ant (weight loc))

; ant-at-home? : ant -> bool
; Check whether ant a is home
;
; (define (ant-at-home? a)
;    ... (ant-weight a)
;    ... (posn-at-home? (ant-loc a)) ...)
; (define (posn-at-home? p)
;    ... (posn-x p) ... (posn-y p) ...)
;
(define (ant-at-home? a)
  (posn-at-home? (ant-loc a)))
(define (posn-at-home? p)
  (and (= (posn-x p) 0) (= (posn-y p) 0)))

(ant-at-home? (make-ant 0.001 (make-posn 0 0)))
"should be" true
(ant-at-home? (make-ant 0.001 (make-posn 1 1)))
"should be" false
```

1 Data representation

2 Purpose & contract

3 Functional examples

4 Template (inventory)

5 Body implementation

6 Test (examples)

```
; An ant is
;   (make-ant num posn)

; A posn is
;   (make-posn num num)


(define (ant-at-home? a)
  ... (ant-weight a)
  ... (posn-at-home? (ant-loc a)) ...)

(define (posn-at-home? p)
  ... (posn-x p) ... (posn-y p) ...)
```

1 Data representation

2 Purpose & contract

3 Functional examples

4 Template (inventory)

5 Body implementation

6 Test (examples)

```
type tree =
    Leaf of int
  | Node of tree * tree
```

1 Data representation

2 Purpose & contract

3 Functional examples

4 Template (inventory)

5 Body implementation

6 Test (examples)

```
type tree =
    Leaf of int
  | Node of tree * tree
```

1 Data representation

2 Purpose & contract

3 Functional examples

4 Template (inventory)

5 Body implementation

6 Test (examples)

```
type tree =
    Leaf of int
  | Node of tree * tree

(* tree → integer
    add up numbers in t *)
let rec sum(t) =
  match t with
     Leaf(i) -> ... i ...
   | Node(l,r) -> ... sum(l)
                  ... sum(r) ...
```

1 Data representation

2 Purpose & contract

3 Functional examples

4 Template (inventory)

5 Body implementation

6 Test (examples)

```
type tree =
    Leaf of int
  | Node of tree * tree

(* tree → integer
   add up numbers in t *)
let rec sum(t) =
  match t with
     Leaf(i) -> ... i ...
   | Node(l,r) -> ... sum(l)
                  ... sum(r) ...
```

Shapes of Data and Templates

```
; An animal is either
;  - snake
;  - dillo
;  - ant

; A snake is
; (make-snake sym num sym)

; A dillo is
; (make-dillo num bool)

; An ant is
; (make-ant num posn)

; A posn is
; (make-posn num num)
```

```
(define (feed-animal a)
  (cond
    [(snake? a) ... (feed-snake a) ...]
    [(dillo? a) ... (feed-dillo a) ...]
    [(ant? a) ... (feed-ant a) ...]))

(define (feed-snake s)
  ... (snake-name s) ... (snake-weight s)
  ... (snake-food s) ...)

(define (feed-dillo d)
  ... (dillo-weight d)
  ... (dillo-alive? d) ...)

(define (feed-ant a)
  ... (ant-weight d)
  ... (feed-posn (ant-loc d)) ...)

(define (feed-posn p)
  ... (posn-x p) ... (posn-y p) ...)
```

The Design Recipe

- Scales to mutually referential datatypes and hierarchies
- Supplemented by "iterative refinement"
- Students easily process files/directories, grammars, river systems, etc. after a few weeks.

Functional Abstraction

`map` is **obvious**:

```
; increase : (list-of num) → (list-of num)
; increase each number by one
```

```
(define (increase alon)
  (cond
    [(empty? alon) empty]
    [else (cons (add1 (first alon))
                (increase (rest alon)))]))
```

⇒

```
(define (increase alon)
  (map add1 alon))
```

Functional Abstraction

`map` is **obvious**, even in OCaml:

```
(* increase : int list → int list
   increase each number by one *)
```

```
let rec increase l =
  match l with
     [] -> []
   | first::rest ->
       add1(first)::increase(rest)
```

⟹

```
let increase l =
  List.map add l
```

Functional Abstraction

Is abstraction really obvious?

- How do students create abstractions?
- How do they use this knowledge in C#?
- How do they know what abstractions exist in ML?
- How should this be of use in Java?

Abstraction as Editing

Programming is like writing — it needs editing

- Editing is called abstraction
- Recognize common patterns and abstract
- Learn to reuse abstractions

Abstraction as Editing

```
; (list-of num) -> (list-of num)
; increase each of the numbers by 1
(define (increase alon)
  (cond
    [(empty? alon) empty]
    [else (cons (add1 (first alon))
                (increase (rest alon)))]))

; tests
(equal? (increase (list 1 2 3))
        (list 2 3 4))
```

```
; (list-of posn) -> (list-of num)
; extract the X coordinate from each
(define (extract alop)
  (cond
    [(empty? alop) empty]
    [else (cons (posn-x (first alop))
                (extract (rest alop)))]))

; tests
(equal? (extract (list (make-posn 2 3)))
        (list 2))
```

Abstraction as Editing

```
; (list-of num) -> (list-of num)
; increase each of the numbers by 1
(define (increase alon)
  (cond
    [(empty? alon) empty]
    [else (cons (add1 (first alon))
                (increase (rest alon)))]))

; tests
(equal? (increase (list 1 2 3))
        (list 2 3 4))
```

```
; (list-of posn) -> (list-of num)
; extract the X coordinate from each
(define (extract alop)
  (cond
    [(empty? alop) empty]
    [else (cons (posn-x (first alop))
                (extract (rest alop)))]))

; tests
(equal? (extract (list (make-posn 2 3)))
        (list 2))
```

Abstraction as Editing

```
; (list-of num) -> (list-of num)
; increase each of the numbers by 1
(define (increase alon)
  (cond
    [(empty? alon) empty]
    [else (cons (add1 (first alon))
                (increase (rest alon)))]))

; tests
(equal? (increase (list 1 2 3))
        (list 2 3 4))
```

```
; (list-of posn) -> (list-of num)
; extract the X coordinate from each
(define (extract alop)
  (cond
    [(empty? alop) empty]
    [else (cons (posn-x (first alop))
                (extract (rest alop)))]))

; tests
(equal? (extract (list (make-posn 2 3)))
        (list 2))
```

```
; (list-of X) (X -> num) -> (list-of num)
; apply afun to each
(define (do-to-all alox afun)
  (cond
    [(empty? alon) empty]
    [else (cons (afun (first alox))
                (do-to-all (rest alox)))]))
```

Abstraction as Editing

```
; (list-of num) -> (list-of num)
; increase each of the numbers by 1
(define (increase alon)
  (do-to-all add1 alon))
```

```
; tests
(equal? (increase (list 1 2 3))
        (list 2 3 4))
```

```
; (list-of posn) -> (list-of num)
; extract the X coordinate from each
(define (extract alop)
  (do-to-all posn-x alop))
```

```
; tests
(equal? (extract (list (make-posn 2 3)))
        (list 2))
```

```
; (list-of X) (X -> num) -> (list-of num)
; apply afun to each
(define (do-to-all alox afun)
  (cond
    [(empty? alon) empty]
    [else (cons (afun (first alox))
                (do-to-all (rest alox)))]))
```

Program Organization

Eventually, even novice programs get large

- Rules for how to create programs
- Rules for how to organize them

Design recipes *empower* students;
programming rules *restrict* them

Program Organization

- One task, one function
- Keep functions small
- Keep like data together
- Name functions properly

Program Organization

```
(define-struct worm (head body))
; Worm = (make-worm Segment (Listof Segment))
; interpretation: head, followed by growth segments

; worm-move : Worm Direction -> Worm
; move the worm one step in the given direction

; worm-grow : Worm Posn -> Worm
; grow the worm by one body segment, head moves to Posn
; accumulator style

; worm-image : Worm -> Image
```

Program Organization

```
(define-struct worm (head body))
; Worm = (make-worm Segment (Listof Segment))
; interpretation: head, followed by growth segments

; worm-move : Worm Direction -> Worm
; move the worm one step in the given direction

; worm-grow : Worm Posn -> Worm
; grow the worm by one body segment, head moves to Posn
; accumulator style

; worm-image : Worm -> Image
```

Program Organization

```
(define-struct worm (head body))
; Worm = (make-worm Segment (Listof Segment))
; interpretation: head, followed by growth segments

; worm-move : Worm Direction -> Worm
; move the worm one step in the given direction

; worm-grow : Worm Posn -> Worm
; grow the worm by one body segment, head moves to Posn
; accumulator style

; worm-image : Worm -> Image
```

...and the same for worm Segments...

...and the same for for Food...

...and then the binary functions...

First Semester Summary

FP in the first semester is about data representations:

- Design systematically to data definition
- Abstract systematically and use abstractions
- Organize programs systematically

Part III: How to Design Data in OOP

Or, how should you transition from FP to OOP

From FP to OOP

Start with data:

```
; A posn is
;   (make-posn num num)
(define-struct posn (x y))
```

```
class Posn {
  int x;
  int y;
  Posn(int x, int y) {
    this.x = x;
    this.y = y;
  }
}
```

From FP to OOP

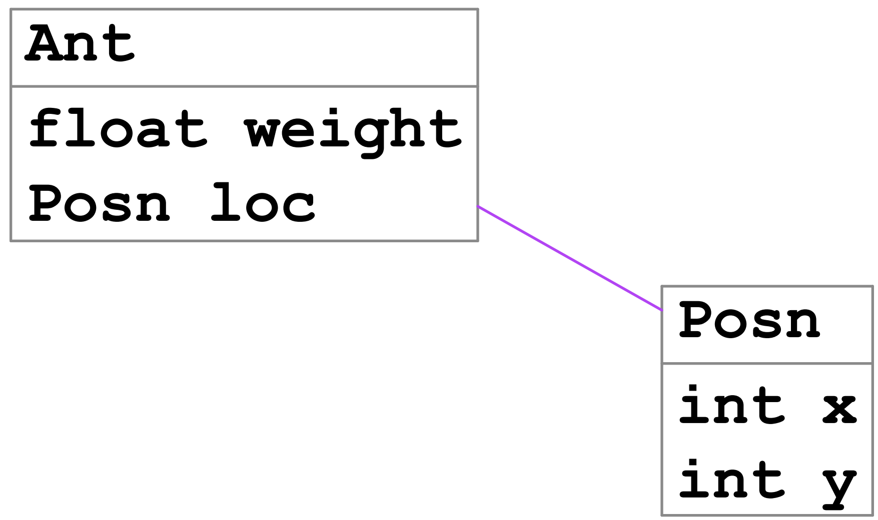
Start with data:

```
; A posn is
;   (make-posn num num)
(define-struct posn (x y))
```

Posn
int x int y

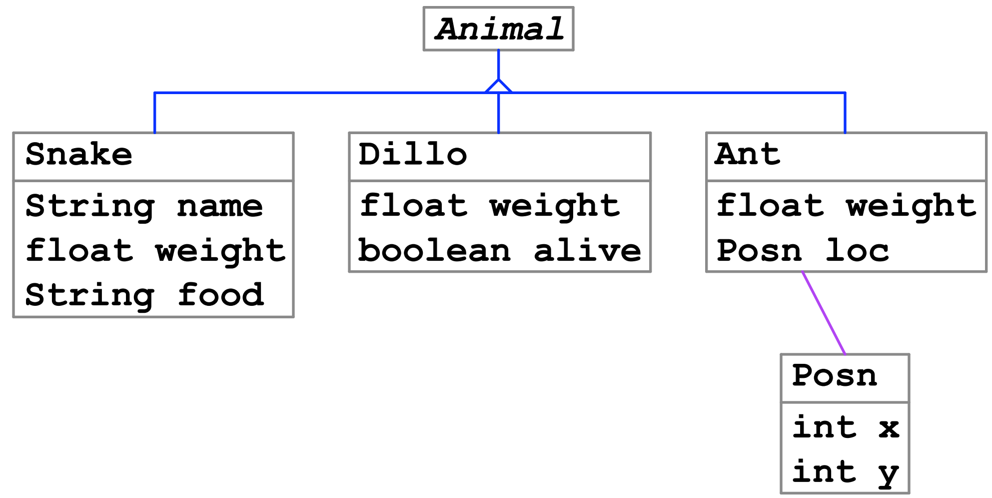
From FP to OOP

```
; An ant is
;   (make-ant num posn)

; A posn is
;   (make-posn num num)
```



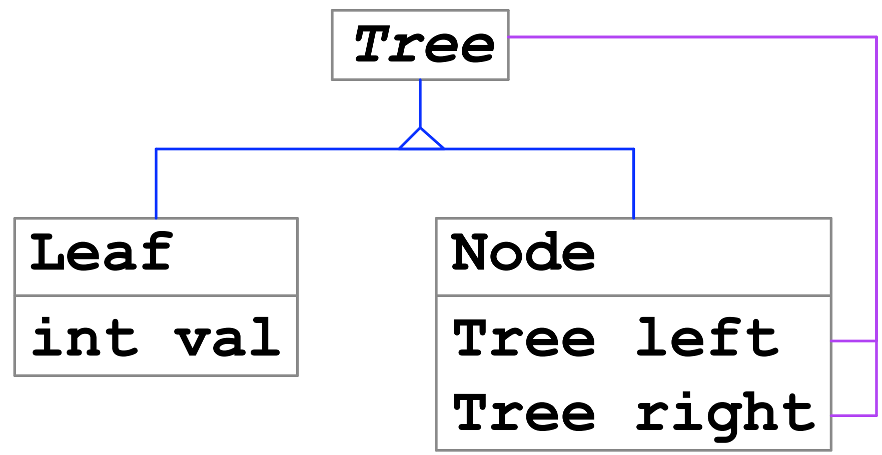
From FP to OOP

```
; An animal is either
;  - snake
;  - dillo
;  - ant

; A snake is
; (make-snake sym num sym)

; A dillo is
; (make-dillo num bool)

; An ant is
; (make-ant num posn)

; A posn is
; (make-posn num num)
```



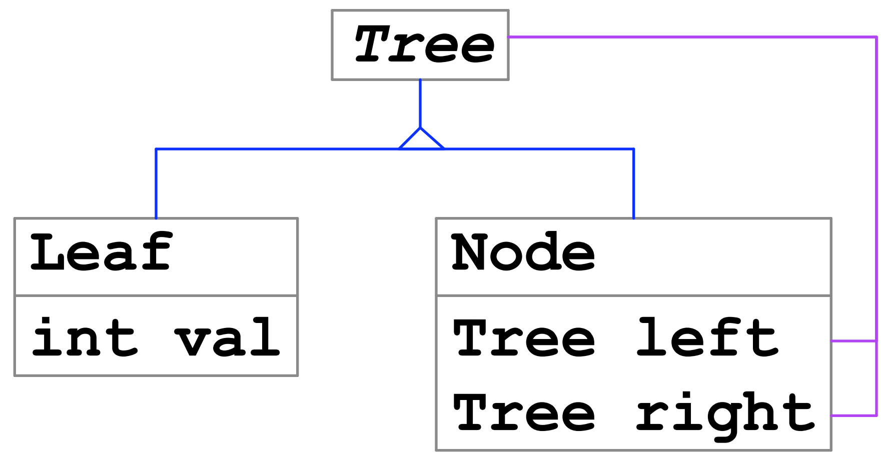
From FP to OOP

```
type tree =
    Leaf of int
  | Node of tree * tree
```



From FP to OOP

```
type tree =
    Leaf of int
  | Node of tree * tree
```



And so on (for mutually referential data definitions)...

From Functions to Methods

```
; An animal is either
;  - snake
;  - dillo
;  - ant
; ...

; animal-is-lighter? : animal num -> bool
(define (animal-is-lighter? a n)
  (cond
   [(snake? a) (snake-is-lighter? s n)]
   [(dillo? a) (dillo-is-ligheter? s n)]
   [(ant? a) (ant-is-lighter? s n)]))

; snake-is-lighter? : snake num -> bool
(define (snake-is-lighter? s n) ...)

; dillo-is-lighter? : dillo num -> bool
(define (dillo-is-lighter? d n) ...)

; ant-is-lighter? : ant num -> bool
(define (ant-is-lighter? a n) ...)
```

```
interface Animal {
  boolean isLighter(double n);
}

class Snake extends Animal {
  ...
  boolean isLighter(double n) { ... }
}

class Dillo extends Animal {
  ...
  boolean isLighter(double n) { ... }
}

class Ant extends Animal {
  ...
  boolean isLighter(double n) { ... }
}
```

From Functions to Methods

Data definition turns into class declarations

```
; An animal is either
;  - snake
;  - dillo
;  - ant
; ...

; animal-is-lighter? : animal num -> bool
(define (animal-is-lighter? a n)
  (cond
   [(snake? a) (snake-is-lighter? s n)]
   [(dillo? a) (dillo-is-ligheter? s n)]
   [(ant? a) (ant-is-lighter? s n)]))

; snake-is-lighter? : snake num -> bool
(define (snake-is-lighter? s n) ...)

; dillo-is-lighter? : dillo num -> bool
(define (dillo-is-lighter? d n) ...)

; ant-is-lighter? : ant num -> bool
(define (ant-is-lighter? a n) ...)
```

```
interface Animal {
  boolean isLighter(double n);
}

class Snake extends Animal {
  ...
  boolean isLighter(double n) { ... }
}

class Dillo extends Animal {
  ...
  boolean isLighter(double n) { ... }
}

class Ant extends Animal {
  ...
  boolean isLighter(double n) { ... }
}
```

From Functions to Methods

```
; An animal is either
;  - snake
;  - dillo
;  - ant
; ...

; animal-is-lighter? : animal num -> bool
(define (animal-is-lighter? a n)
  (cond
   [(snake? a) (snake-is-lighter? s n)]
   [(dillo? a) (dillo-is-ligheter? s n)]
   [(ant? a) (ant-is-lighter? s n)]))

; snake-is-lighter? : snake num -> bool
(define (snake-is-lighter? s n) ...)

; dillo-is-lighter? : dillo num -> bool
(define (dillo-is-lighter? d n) ...)

; ant-is-lighter? : ant num -> bool
(define (ant-is-lighter? a n) ...)
```

Variant functions turn into variant methods — all with the same contract after the implicit argument

```
interface Animal {
  boolean isLighter(double n);
}

class Snake extends Animal {
  ...
  boolean isLighter(double n) { ... }
}

class Dillo extends Animal {
  ...
  boolean isLighter(double n) { ... }
}

class Ant extends Animal {
  ...
  boolean isLighter(double n) { ... }
}
```

From Functions to Methods

```
; An animal is either
;  - snake
;  - dillo
;  - ant
; ...
```

```
; animal-is-lighter? : animal num -> bool
(define (animal-is-lighter? a n)
  (cond
   [(snake? a) (snake-is-lighter? s n)]
   [(dillo? a) (dillo-is-ligheter? s n)]
   [(ant? a) (ant-is-lighter? s n)]))
```

```
; snake-is-lighter? : snake num -> bool
(define (snake-is-lighter? s n) ...)

; dillo-is-lighter? : dillo num -> bool
(define (dillo-is-lighter? d n) ...)

; ant-is-lighter? : ant num -> bool
(define (ant-is-lighter? a n) ...)
```

Function with variant-based **`cond`** turns into just an **`abstract`** method declaration

```
interface Animal {
  boolean isLighter(double n);
}

class Snake extends Animal {
  ...
  boolean isLighter(double n) { ... }
}

class Dillo extends Animal {
  ...
  boolean isLighter(double n) { ... }
}

class Ant extends Animal {
  ...
  boolean isLighter(double n) { ... }
}
```

How to Design Classes & Interfaces

- We (ought to) know this much
- Data types via interpreter/composite patterns
- Design recipe process produces instances of those patterns — students can tackle anything
- Students see FP mechanics (conditional) vs. OOP mechanics (dynamic dispatch)

Part IV: How to Design Hierarchies

Or, why the principles of FP à la
HtDP produce class hierarchies

Method Abstraction

Duplication of code in method bodies:

```
AClass
---------------------------------------
A m(B x) { ... x ... }
A n(B x, C y) { ... x ... }
```

Method Abstraction

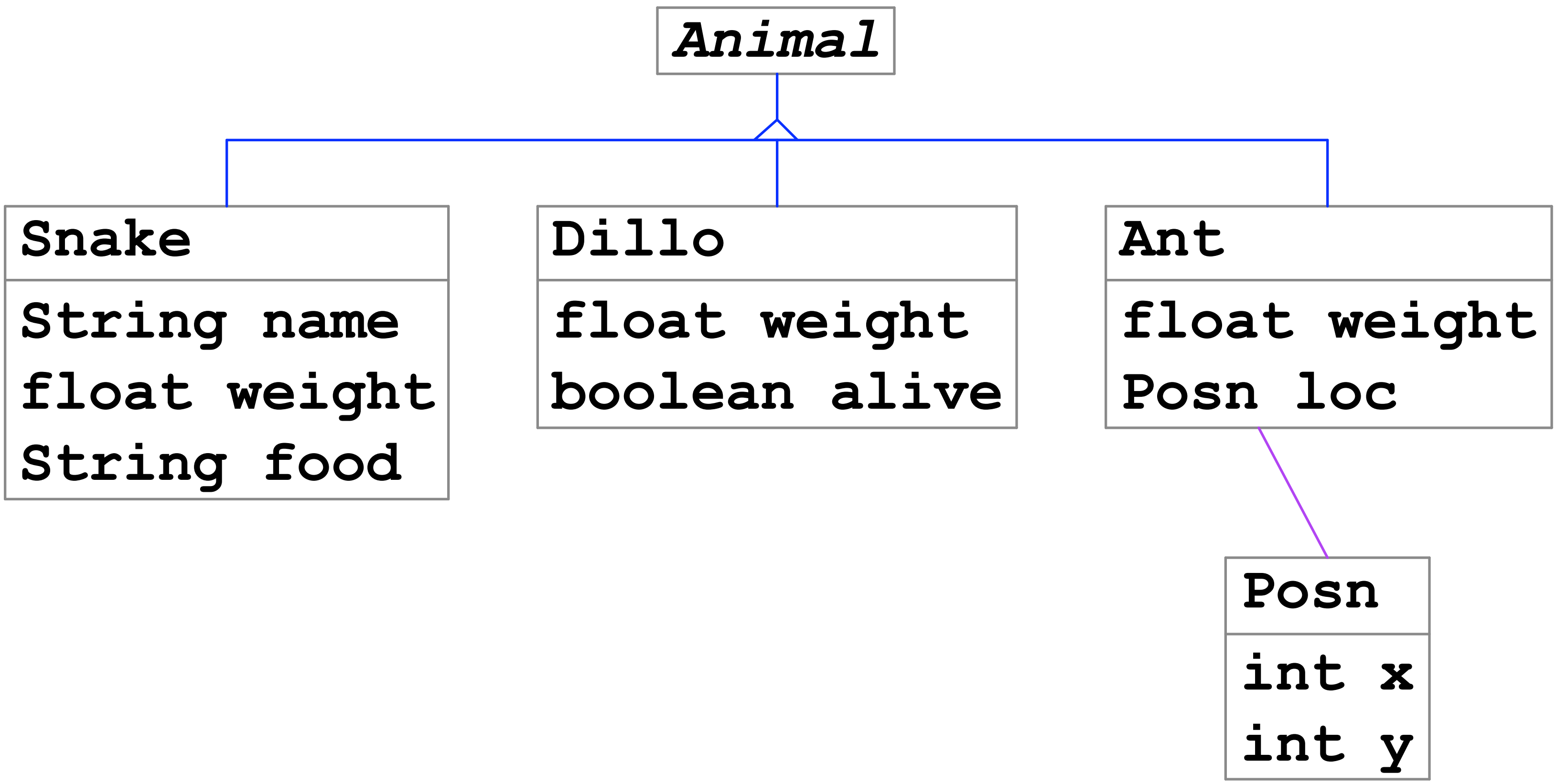
Duplication of code in method bodies:

- Handle as in FP

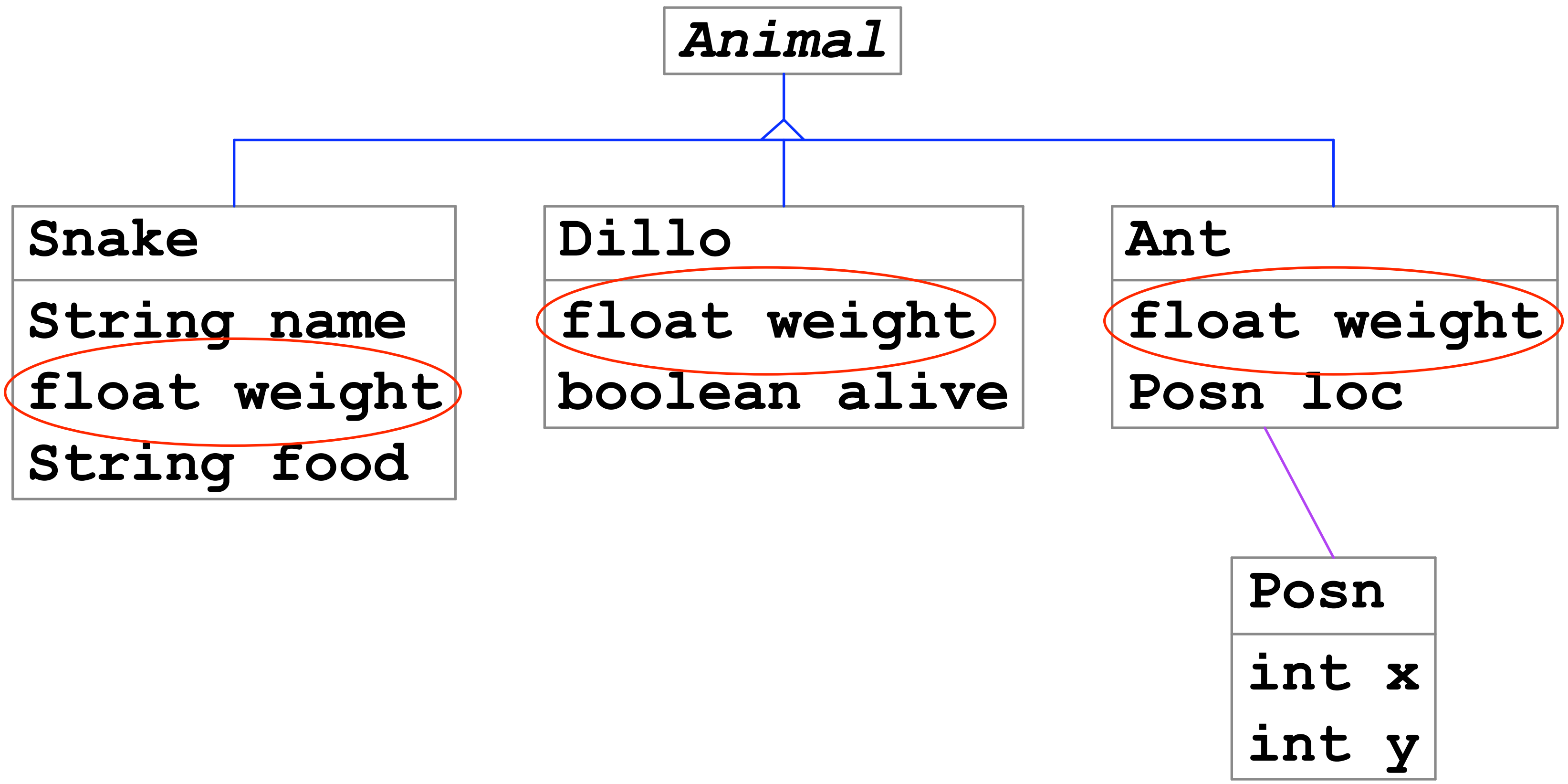
```
AClass

A q(B x, D z) { ... x ... }
A m(B x) { return q(x, new D(42)); }
A n(B x, C y) { return q(x, y.p()); }
```

Of course, it's clumsy without λ...

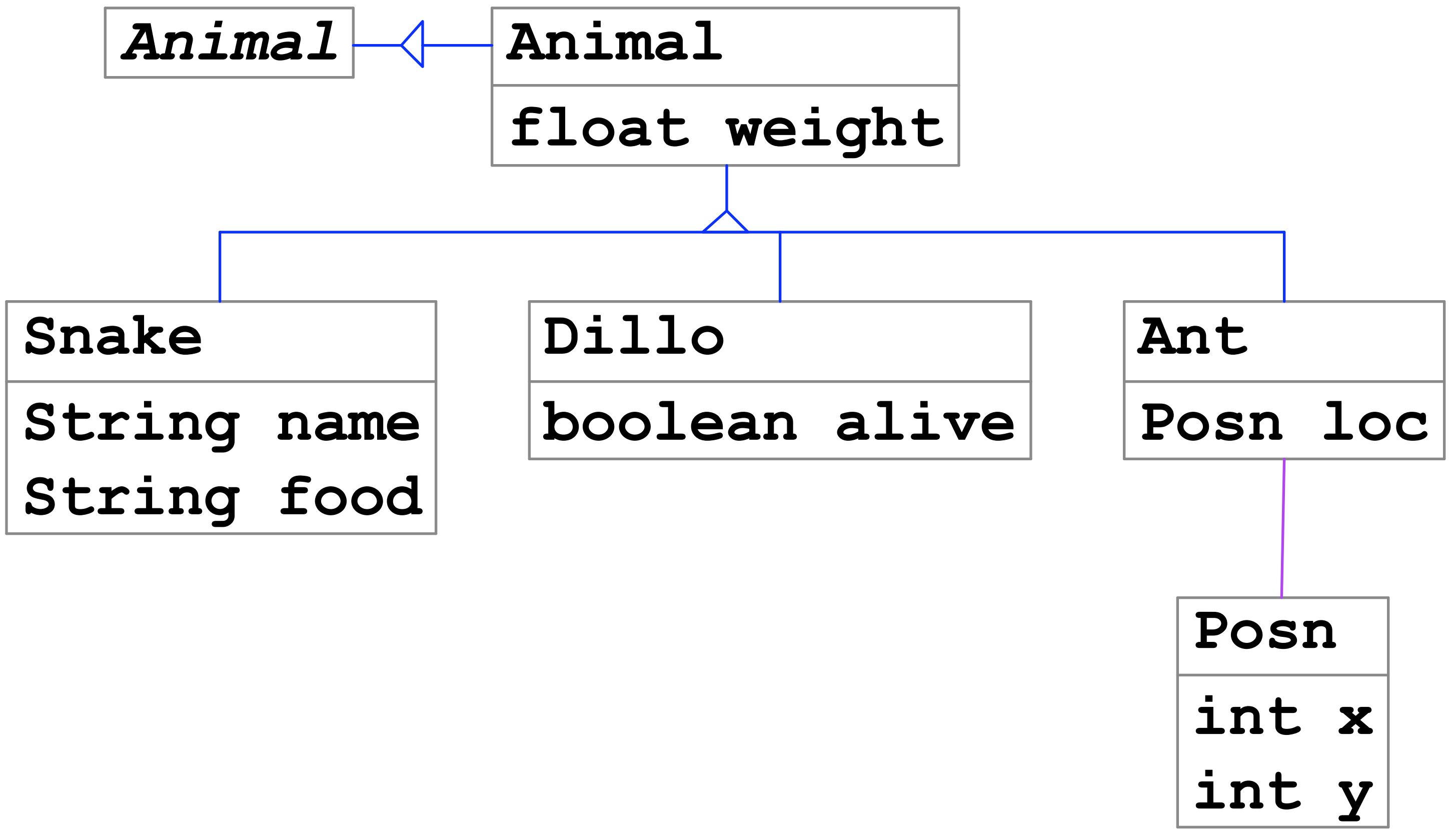
Field Abstraction



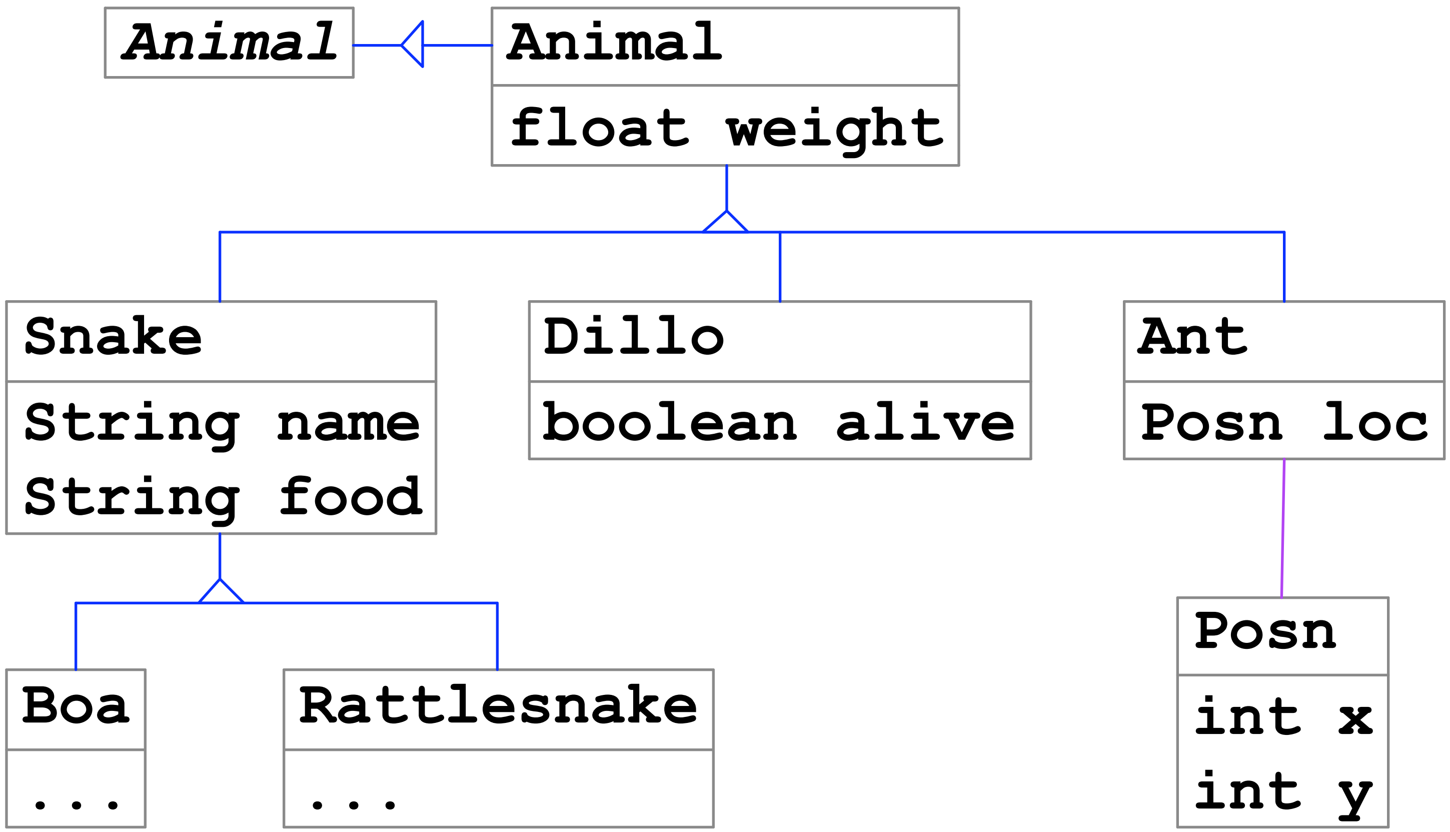
Field Abstraction



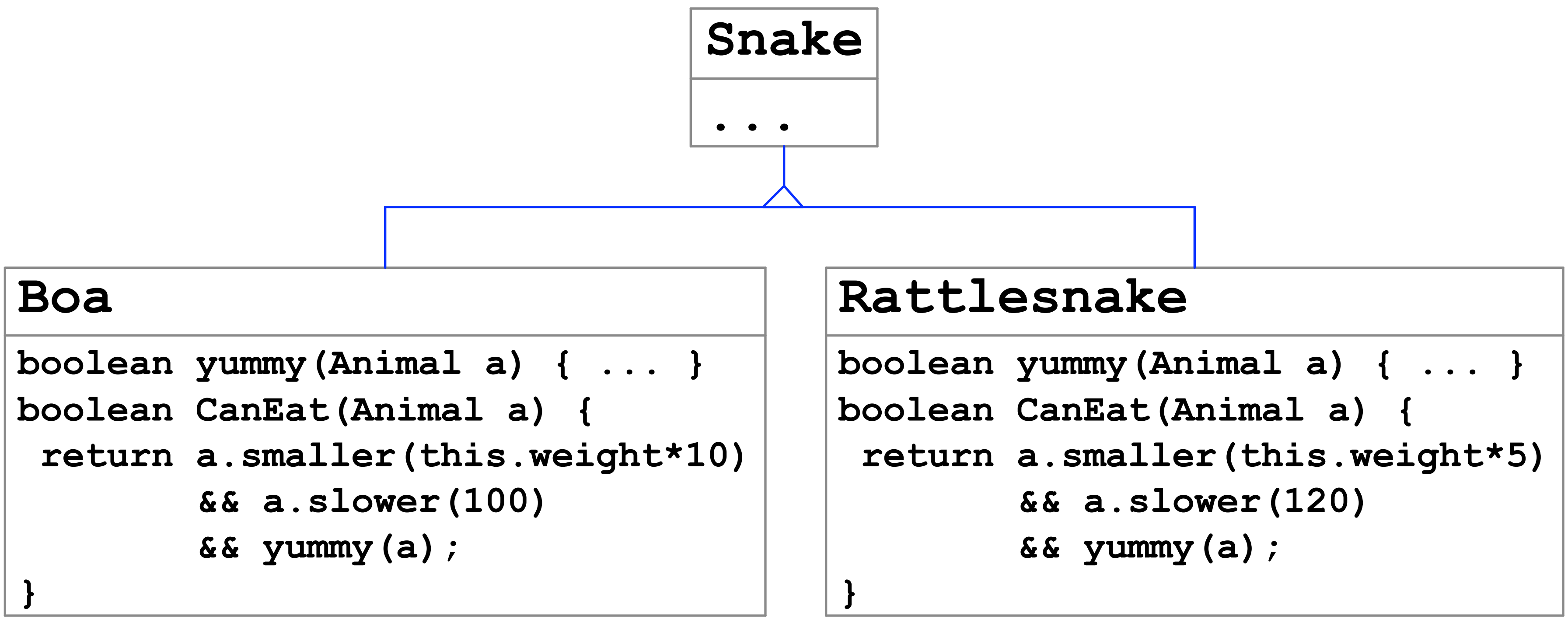
Field Abstraction



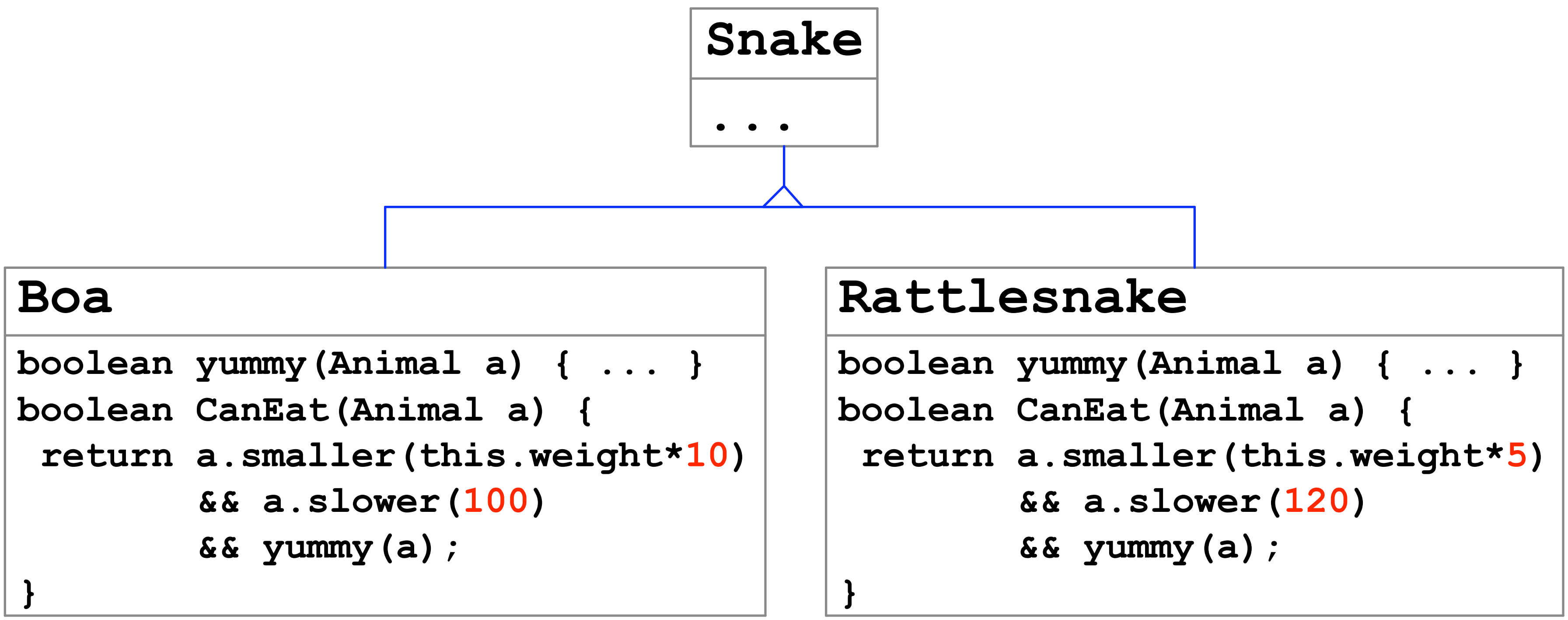
Deep Hierarchies



Deep Hierarchies



Deep Hierarchies



Deep Hierarchies

```
Snake
...
abstract boolean yummy(Animal a);
boolean CanEatWFacts(Animal a, int wf, int sd) {
 return a.smaller(this.weight*wf)
        && a.slower(sd)
        && yummy(a);
}
```

```
Boa
boolean yummy(Animal a) { ... }
boolean CanEat(Animal a) {
 return CanEatWFacts(a, 10, 100);
}
```

```
Rattlesnake
boolean yummy(Animal a) { ... }
boolean CanEat(Animal a) {
 return CanEatWFacts(a, 5, 120);
}
```

Result: the **template-hook pattern** — via reasoning, not ad hoc process

More Abstraction for Class Hierarchies

- Abstraction inside of classes/hierarchies
- Abstraction from a client-only perspective
- Abstraction over traversals of collections (`map`, `fold`, etc.)

All of these steps yield code that is pattern-based

$\Rightarrow$ theory and practice coincide!

Part V: Encapsulated State

Or, without hiding and encapsulating state, it can't be real OOP — right?

Encapsulated State

OOP proponents argue that OOP is about state:

- It is about hidden and encapsulated state
- It is about manipulating state

They're right!

State: What's Wrong with this Program?

```
// represent the world of a Worm Game: Worm, Food, and the Bounding Box
class WormWorld extends World {
  Worm w;
  Food f;
  Box  b;
  WormWorld(Worm w, Food f, Box b) { ... }

  // what happens when the clock ticks
  World onTick() { return okWorld(w.nextWorld()); }

  // what happens when the player presses a key
  World onKeyEvent(String ke) { return okWorld(w.changeDirection(ke)); }

  // create a new world from the new worm, unless it ran into a wall
  // or ate itself
  World okWorld(Worm newWorm) {
    if (newWorm.canEat(this.f) && this.ate(newWorm))
      return new WormWorld(newWorm.eat(this.f),f.create(this.b),b)
    else ...
    }

   // did the worm encounter food and eat it?
   boolean ate(Worm w) { ... }
  ...
```

State: What's Wrong with this Program?

```
// represent the world of a Worm Game: Worm, Food, and the Bounding Box
class WormWorld extends World {
  private Worm w;
  private Food f;
  private Box  b;
  private WormWorld(Worm w, Food f, Box b) { ... }

  // override: what happens when the clock ticks
  public World onTick() { return okWorld(w.nextWorld()); }

  // override: what happens when the player presses a key
  public World onKeyEvent(String ke) { return okWorld(w.changeDirection(ke)); }

  // create a new world from the new worm, unless it ran into a wall
  // or ate itself
  private World okWorld(Worm newWorm) {
    if (newWorm.canEat(this.f) && this.ate(newWorm))
      return new WormWorld(newWorm.eat(this.f),f.create(this.b),b)
    else ...
    }

   // did the worm encounter food and eat it?
   private boolean ate(Worm w) { ... }
  ...
```

It's missing some keywords

Encapsulation Means Privacy

- Hiding & encapsulating state means privacy
- Introduce when programs get large enough and students have a sense of invariants

Encapsulation does *not* imply a litany of assignment statements

Manipulating State: Methods produce Worlds

- Manipulating state means that some methods produce a new instance of the world
- Introduce this topic when you have an I/O library that doesn't change state

Manipulating state does *not* imply a litany of assignment statements

State and "Real" Programs

- Our students regularly design interactive programs without a single assignment statement

```
class WormWorld extends World {
  ...
  // what happens when the player presses a key
  public World onKeyEvent(String ke) { ... }
  ...
}
```

- Schemers accept the occasional assignment statement as necessary — and that's really OOP

Part VI: Experience

In the Classroom

- Development time: 3 years
- Test teaching at Northeastern: 6 semesters, 4 instructors (mostly Proulx)
- Test teaching at UChicago: 1 summer semester (Gray)
- Teacher/college workshops: 2 summers (Proulx & Gray)
- A dozen teachers in high schools, small colleges

Evaluation

Northeastern University:

- Four follow-up faculty have testified that the new crop of students are *vastly* better than the previous ones
- It works for several instructors, so it's not the "enthusiasm of the inventor" (he doesn't teach it)

High School:

- Teachers report strong AP results

Chicago:

- Too early for results

Objections

Common objection: it's FP in Java syntax

Our reply:

- See quotes at the beginning of the talk — this is True OOP™
- We emphasize what OOP is all about for real programming:
    - design of classes and hierarchies
    - abstraction
    - encapsulation
- Yes, the occasional assignment statement helps

Summary

What Doesn't Work

- *SICP*-style approaches: teaching how to implement OO in an FP
- Cartwright (at Rice) approach: teaching how to interpret an FPL in an OOPL

Students must know such principles, but it doesn't teach OOP

What Works

Take OOP seriously

- It's about the systematic design of classes
- It's about the systematic design of hierarchies
- It's about server/provider vs. client/consumer
- It's best seen as FP with grouping, privacy, and novel forms of abstraction

To produce the best OO programmers, teach FP, and then move on

HtDP and *HtDCH* demonstrate this approach with successful results

Thanks!