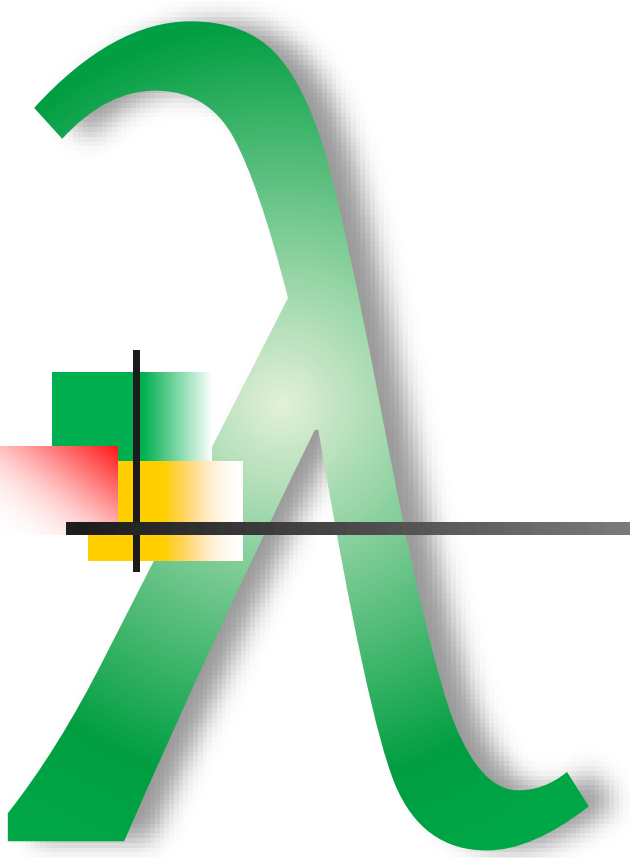
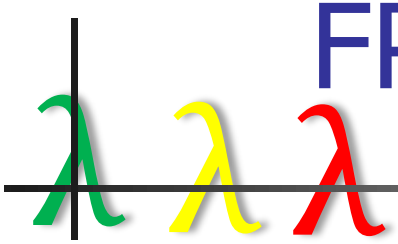


Design of Classes I

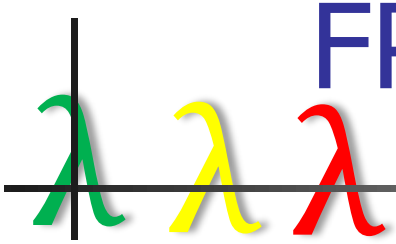
Marco T. Morazán
Seton Hall University

FP Design → OO Design?



- How do we get students started in OOP?
  - After a design-based introduction to programming
    - Design does not become obsolete!
    - Good selling point for universities
  - Simon's tennis racket
- Design-based approach
  - Design Recipe
  - Unit Testing
  - Type-oriented programming
    - Much more than structural recursion

FP Design → OO Design?

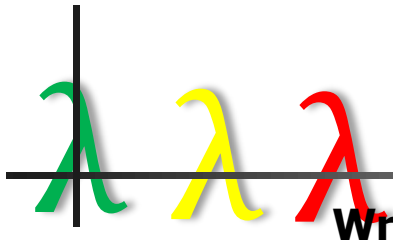


- Approach at Seton Hall University
  - Build on the design recipe
  - Build on student programming maturity
    - Full Java syntax
    - Wonderful error messages…
  - Structures → Classes
  - Union types → Interfaces
  - HOFs → Generic classes and interfaces
  - Functional abstraction → Abstract Classes

Basic Design Recipe

1. Problem Analysis
2. Data Analysis
3. Function Template Development
4. Signature and Purpose Statement
5. Function Header
6. Unit Tests
7. Function Body
8. Run Tests

Basic Design Recipe



Write a function to determine a student's year level

```
;; Credits: <= 30 is Freshman, <= 60 is Sophomore, <= 90 is a Junior, and > 90 is Senior
;; student is a structure, (make-student string R>=0 natnum),with name, gpa, & number of credits.
(define-struct student (name gpa credits))
```
→ *make-student, student?, student-name, student-gpa, student-credits*

```
;; Sample students
(define STUDENT1 (make-student "Spiderman" 3.5  16))
(define STUDENT2 (make-student "Batwoman"  3.9  43))
(define STUDENT3 (make-student "Superman"  3.7  75))
(define STUDENT4 (make-student "Ironman"   4.0  120))

;; student → string          Purpose: Determine the given student's level
(define (student-level a-student)
  (cond    [(<= (student-credits a-student) 30) "Freshman"]
           [(<= (student-credits a-student) 60) "Sophomore"]
           [(<= (student-credits a-student) 90) "Junior"]
           [else "Senior"]))
(check-expect (student-level STUDENT1) "Freshman")
(check-expect (student-level STUDENT2) "Sophomore")
(check-expect (student-level STUDENT3) "Junior")
(check-expect (student-level STUDENT4) "Senior")
```

Basic Design Recipe in Java Syntax

- For every data definition you write a class
- Object = data + type operations
- Must write: constructor, selectors (& mutators)
- Must write Junit Test File
- Design Recipe
    1. Problem Analysis
    2. Data Analysis
    3. Class Template Development
    4. Tests Template Development
    5. Write Tests
    6. Method Development
        - (a) Write Signature and Purpose Statement
        - (b) Write Method Header
        - (c) Write Unit Tests
        - (d) Write Method Body
    7. Run Tests

Basic Design Recipe in Java Syntax

Write a function to determine a student's year level

;; Credits: <= 30 is Freshman, <= 60 is Sophomore, <= 90 is a Junior, and > 90 is Senior

;; Student is an *object*, **new** student(string R>=0 natnum),with name, gpa, & number of credits.

- For every data definition you write a class
- Object = data + type operations
- Must write: constructor, selectors (& mutators)
- Must write Junit Test File

```
class TestStudent
{ @Test
  void testStudentMethods()
  {       Student S1=new Student("Spiderman"  3.5  16); Student S2 = new Student("Batwoman"  3.9  43);
          Student S3=new Student("Superman"   3.7  75); Student S4 = new Student("Ironman"    4.0  120);
          assertEquals(S1.getName(), "Spiderman");
          assertEquals(S1.getGpa(), 3.5, 0.01);
          assertEquals(S1.getCredits(), 16);
          assertEquals(S1.isStudent(), true);
          assertEquals(S1.getLevel(), "Freshman");
          assertEquals(S2.getLevel(), "Sophomore");
          assertEquals(S3.getLevel(), "Junior");
          assertEquals(S4.getLevel(), "Senior"); }  }
```

Basic Design Recipe in Java Syntax

Write a function to determine a student's year level

;; Credits: <= 30 is Freshman, <= 60 is Sophomore, <= 90 is a Junior, and > 90 is Senior

;; Student is an object, **new** student(string R>=0 natnum),with name, gpa, & number of credits.

- For every data definition you write a class
- Object = data + type operations
- Must write: constructor, selectors (& mutators)

```
class Student
{String name;  double gpa;  int credits;
 Student(String nm, double g, int c)
 {name = nm;  gpa = g;  credits = c; }
 String   getName()  { return(name);  }
 double  getGpa()     { return(gpa); }
 int        getCredits() { return(credits);  }
 Boolean isStudent()  { return(true); }
```

```
Purpose: Determine the given student's level
String getLevel()
{ if (credits <= 30)
    {return("Freshman");}
  else if (credits <= 60)
          {return("Sophomore");}
       else if (credits <= 90)
             {return("Junior");}
             else
             {return("Senior");} } }
```

Union Type Design

```
;; A list of number (lon) is one of:
;; empty
;; (cons number lon)

;; A list of string (los) is one of:
;; empty
;; (cons string los)
```

Parameterized Data Def
X is a type variable

```
;; A list of X (loX) is one of:
;; empty
;; (cons X loX)
```

```
;; (listof X) → number   Purpose: To compute the length of the given list
(define (len-lox a-lox)
  (if (empty? a-lox)
      0
      (+ 1 (len-lox (rest a-lox)))))
(check-expect (len-lox '())       0)
(check-expect (len-lox '(1 2 3)) 3)
```

Interface Design

- new way to define a type: data + operations
- Instances of interfaces are called *objects*

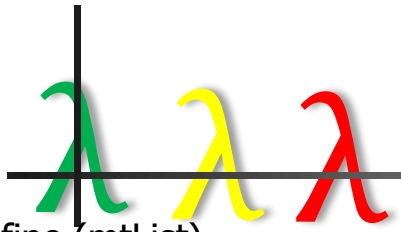
An IloX is an interface that provides the following services:

1. first: X throws error
2. rest: Ilox throws error
3. empty?: Boolean
4. cons: (X → Ilox)
5. equals: (Ilox → Boolean)
6. [Y] map: (X → Y) → I(listof Y)
7. [A] foldl: (X → A) A → A

```
;; A loX is one of:
;;  empty
;;  (cons X loX)
```

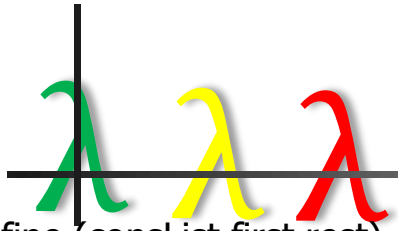
parameterized signatures

Interface Design



```
(define (mtList)
  (local [
          (define (cons an-x) (consList an-x service-manager))
           (define (equals L) (L 'empty?))
          ;; [Y]: (X → Y) → I(listof Y)
          (define (map f) service-manager)
          ;; [A]: (X → A) A → A
          (define (foldl f acc) acc)
          (define (service-manager m)
            (cond       [(eq? m 'first)   (error "first applied to the empty list")]
                        [(eq? m 'rest)    (error "rest applied to the empty list")]
                        [(eq? m 'empty?)  #true]
                        [(eq? m 'cons)    cons]
                        [(eq? m 'equals)  equals]
                        [(eq? m 'map)     map]
                        [(eq? m 'foldl)   foldl]
                        [else (error (format "Unknown list service requested: ~s" m))]))]
    service-manager))
```

Interface Design

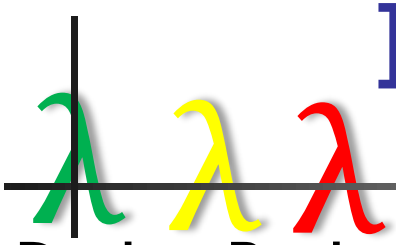


```
(define (consList first rest)
  (local [
          (define (cons an-x) (consList an-x service-manager))
          (define (equals L) (and (equal? first (L 'first))  ((rest 'equals) (L 'rest))))
          ;; [Y]: (X → Y) → I(listof Y)
          (define (map f) (consList (f first) ((rest 'map) f)))
          ;; [A]: (X → A) A → A
          (define (foldl f acc) ((rest 'foldl) f (f first acc)))
          (define (service-manager m)
            (cond       [(eq? m 'first)     first]
                        [(eq? m 'rest)      rest]
                        [(eq? m 'empty?) #false]
                        [(eq? m 'cons)     cons]
                        [(eq? m 'equals)  equals]
                        [(eq? m 'map)      map]
                        [(eq? m 'foldl)    foldl]
                        [else (error (format "Unknown list service requested: ~s" m))]))]
    service-manager))
```

Interface Design

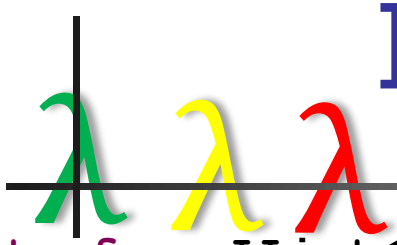
```
;; Sample instances of Ilox
(define L0 (mtList))
(define L1 (consList 1 (consList 2 (consList 3 (mtList)))))
(define L2 ((L1 'map) add1))
(define L3 (consList 4 (consList 5 (consList 6 (mtList)))))

(check-expect (L1 'first)  1)
(check-expect (((L1 'rest) 'equals)  (consList 2 (consList 3 (mtList)))) true)
(check-expect (L0 'empty?) true)
(check-expect (L3 'empty?) false)
(check-expect ((L0 'equals) (mtList)) true)
(check-expect ((L1 'equals) L2) false)
(check-expect ((L1 'equals) L4) true)
(check-expect ((L2 'equals) (consList 2 (consList 3 (consList 4 (mtList))))) true)
(check-expect ((((L1 'map) add1) 'equals) (consList 2 (consList 3 (consList 4 (mtList))))  true)
(check-expect ((L2 'foldl) (λ (x acc) (+ x acc)) 0)  9)
```

Interface Design in OO



- Design Recipe
  1. Problem Analysis
  2. Data Analysis Reveals Need for a Union Type
  3. Design Interface
  4. Develop Unit Tests for Interface
  5. Implement the Interface for each Subtype Using a Class
  6. Method Development for each Class
     - (a) Write Purpose Statement
     - (b) Write Method Header
     - (c) Write Unit Tests
     - (d) Write Method Body
  7. Run Tests

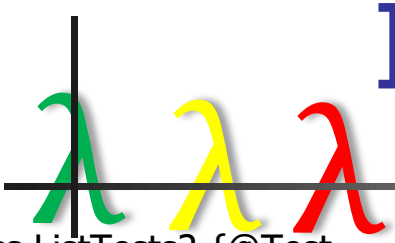
Interface Design in OO



```
interface LList<X>
{// Purpose: add the given X to the front of this list
 LList<X> cons(X val);
 // Purpose: Determine if list is empty
 boolean isEmpty();
 X first() throws Exception;
 LList<X> rest() throws Exception;
 boolean equals(IList<X> l);
 // Purpose: Return LLIST<Y> from applying
 //          the given function to all
 //          elements of this list
 <Y> LList<Y> map(IFun<X, Y> f);
 <R> R foldl(IFun2<X, R> f, R res);
}
```

1. cons: (X → Ilox)
2. empty?: Boolean
3. first: X throws error
4. rest: Ilox throws error
5. equals: (Ilox → Boolean)
6. map: (X → Y) → I(listof Y)
7. foldl: (X → A) A → A

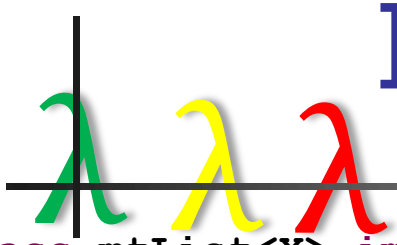
Interface Design in OO



```
class ListTests2 {@Test
 void test()
 {IList<Integer> N = new MTLIST2<Integer>();   IList<Integer> N1 = N.cons(4).cons(6).cons(3);
  IList<Integer> N2 = N.cons(4).cons(6).cons(3);  IList<Integer> N3 = N.cons(3).cons(6).cons(4);
  IList<String> E = new MTLIST2<String>();        IList<String> L0 = E.cons("pal!").cons("there ").cons("Hi ");
try {          assertEquals(N1.first(), (Integer) 3);
               assertEquals(N1.rest().equals(N.cons(4).cons(6)), true);
               assertEquals(N.isEmpty(), true);
               assertEquals(N1.isEmpty(), false);
               assertEquals(N.cons(10).equals((new mtList<Integer>()).cons(10)), true);
               assertEquals(N1.cons(0).equals((new mtList<Integer>()).cons(4).cons(6).cons(3).cons(0)), true);
               assertEquals(N.equals(N1), false);
               assertEquals(N1.equals(N2), true);
               assertEquals(E.map(s -> s.length()).equals(N), true);
               assertEquals(L0.map(s -> s.length()).equals(N1), true);
               assertEquals(N.foldl((n, r) -> r.cons(n), N).equals(N), true);
               assertEquals(N2.foldl((n, r) -> r.cons(n), N).equals(N3), true); }}}
```

- **Introduce students to (untyped) λ-expressions**
- **Introduce students to dot composition**

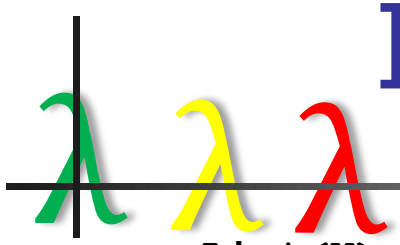
Interface Design in OO



```
class mtList<X> implements LList<X>
{       mtList() {}
        LList<X> cons(X val)
        {return((new consList<X>(val, this)));  }
        X first() throws Exception
        {throw new Exception("Empty list has no first.");}
        LList<X> rest() throws Exception
        {throw new Exception("Empty list has no rest.");}
        Boolean isEmpty() {return(true);}
        boolean equals(IList<X> l) {return(l.isEmpty());}
         <Y> LList<Y> map(IFun<X, Y> f)
        {return(new mtList<Y>());}
        <A> A foldl(IFun2<X, A> f, A res) {return(res);} }
```

- **this** instead of service manager
- **implements** associates class to interface
- **Exception** instead of error
- Parametrized signatures

Interface Design in OO

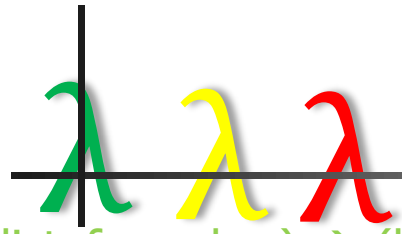


```
class consList<X> implements LList<X>
{X f;  LList<X> r;
 consList(X v, LList<X> rest) {f = v;  r = rest;}
 LList<X> cons(X val)
 {return((new consList<X>(val, this)));}
 X first() {return(f);}
 LList<X> rest(){return(r);}
 Boolean isEmpty() {return(false);}
 boolean equals(IList<X> l)
 {try{return(this.first().equals(l.first()) &&
             this.rest().equals(l.rest()));}
  catch(Exception e)
  {System.out.println("Error NMTLIST equals: " + e.getMessage());
   return(false);}}
 <R> LList<R> map(IFun<X, R> f)
 {return(newconsList<R>(f.f(this.first()),this.rest().map(f)));}
 <R> R foldl(IFun2<X, R> fn, R res)
 {return(this.rest().foldl(fn, fn.f(this.first(), res)));}  }
```

Functional Abstraction

- The DR for abstraction
  1. The Comparison: Compare functions and mark differences
  2. The Abstraction: Define the abstract function with the differences as parameters
  3. Define the original functions using the abstract function: Re-write functions using the abstraction & test them
  4. Write the signature for the abstract function

Functional Abstraction



```
; (listof number) → (listof number)
(define (sqr-list L)
        (cond   [(empty? L) '()]
                [else (cons (sqr (first L)) (sqr-list (rest L)))]))
; (listof ir) → (listof string)
(define (names a-lir)
        (cond   [(empty? L) '()]
                [else (cons (ir-name (first L)) (names (rest L)))]))
; [X Y] map: (listof X) (X → Y) → (listof Y)
(define (map1 L f)
        (cond   [(empty? L) '()]
                [else (cons (f (first L))  (map1 (rest L) f))]))
(define (sqr-list L) (map1 L sqr))

(define (names ls) (map1 L ir-name))
```

Abstraction in Java

- The DR for abstraction
  1. The Comparison: Compare classes and mark similarities
  2. The Abstraction: Define an abstract class containing the similarities
  3. Have classes extend abstract class: Classes *inherit* similarities
  4. Refactor Code: Reimplement in terms of other methods to create similarities

```
abstract class AList<X> implements IList<X>
{
        IList<X> cons(X v)
        {return(new NMTLIST<X>(v, this));} }

class mtList<X> extends AList<X>
{ … }
class consList<X> extends AList<X>
{ … }
```

Abstraction in Java

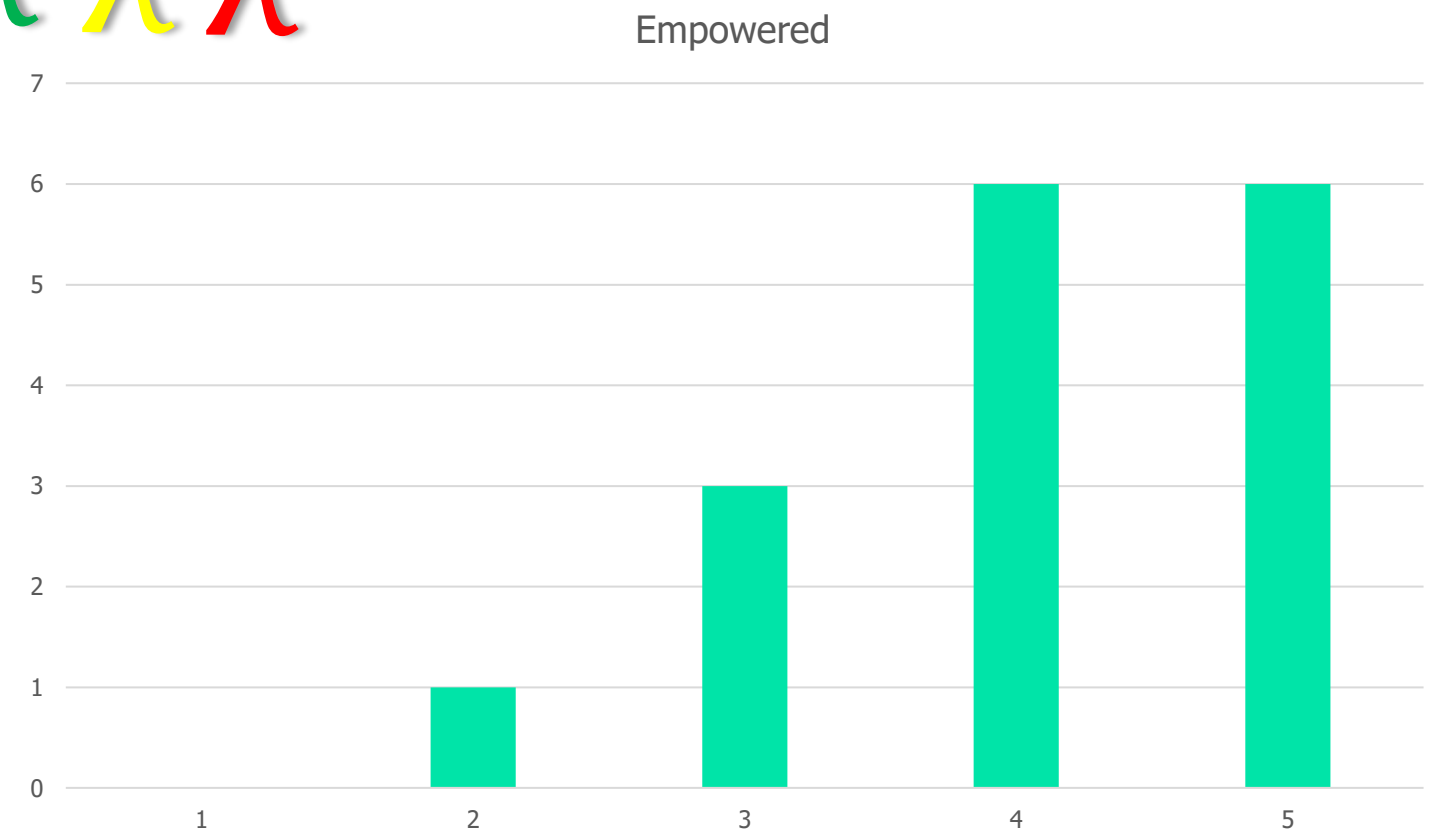
```
mtList:
boolean equals(IList<X> l) {return(l.isEmpty());}
consList:
boolean equals(IList<X> l)
{try{  return((this.first().equals(l.first())) &&
       this.rest().equals(l.rest()));}
 catch(Exception e)
 {System.out.println("Error NMTLIST equals: " + e.getMessage());
  return(false);}}
```

- How do we create similarities?
- Introduce students to refactoring
  - If input is of different subtype → false
  - Otherwise → foldl a function that tests first elements for equality

Abstraction in Java

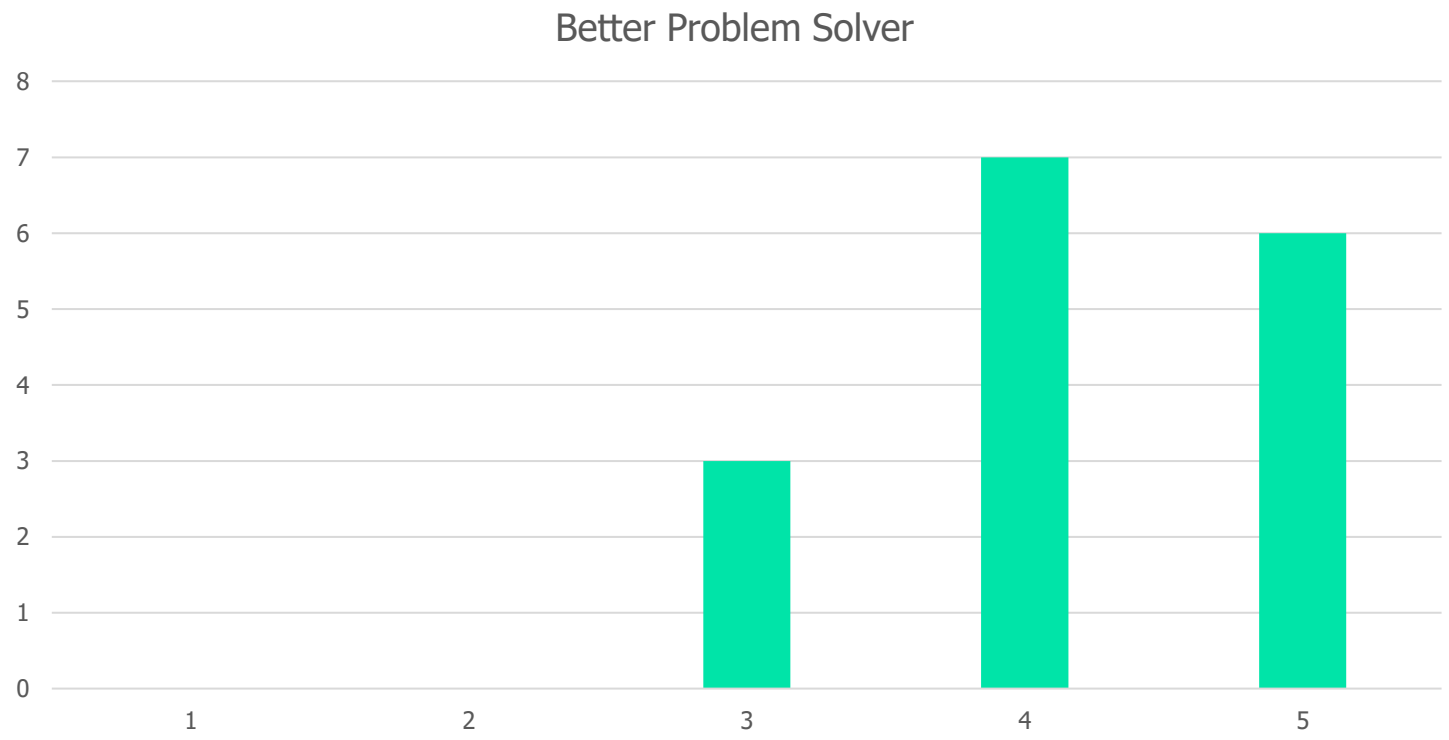
```
abstract class AList<X> implements IList<X>
{
  IList<X> cons(X v)
  {return(new NMTLIST<X>(v, this));}

  boolean equals(IList<X> l)
  {if ((this.isEmpty() && !l.isEmpty()) ||
       ((!this.isEmpty() && l.isEmpty()))
   {return(false);}
   {return(this.foldl((x, r) ->
                      {try {return(x.equals(l.first()) && r);}
                       catch(Exception e){
                        System.out.println("…" + e.getMessage());
                        return(false);}},
          true));}}}
```

Student Feedback



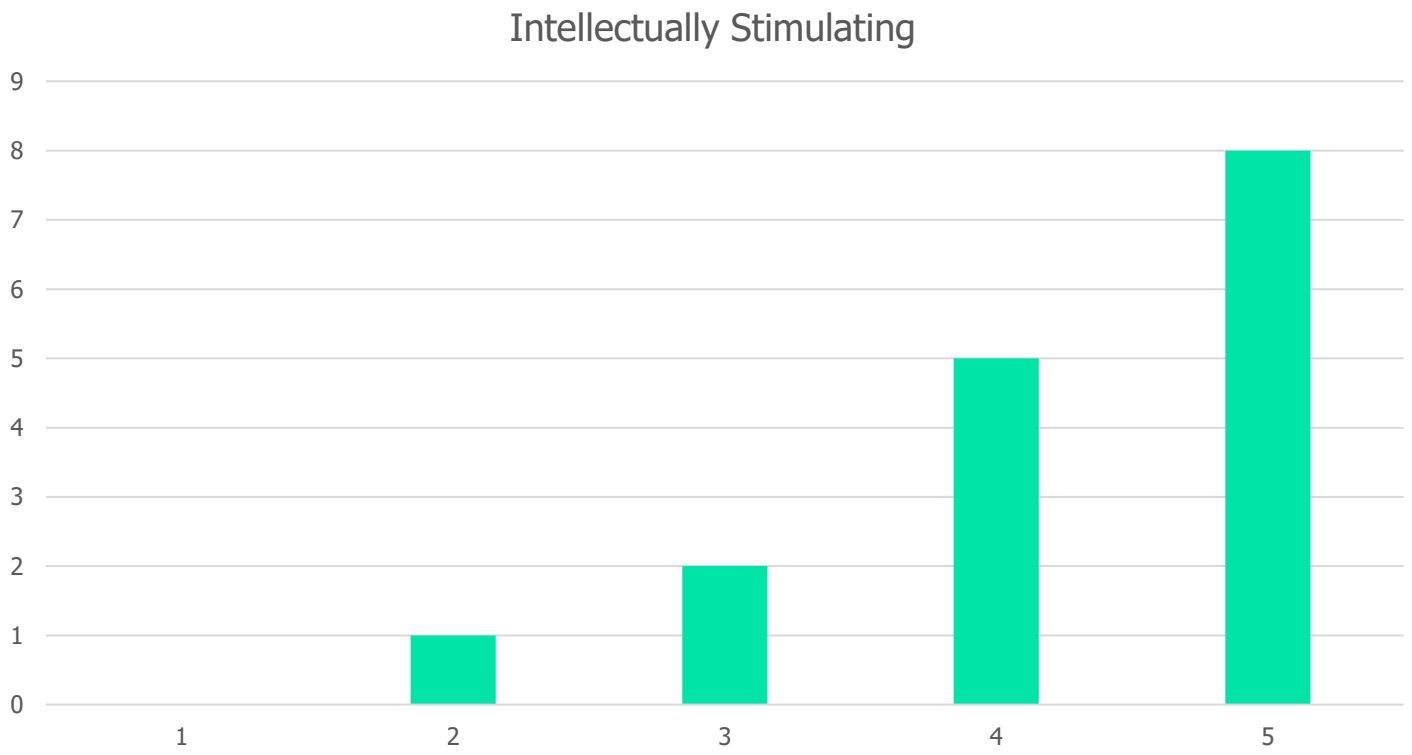
- Do you feel empowered by knowing how to design and implement OO programs?
- 1 = Not at all empowered
- 5 = Very empowered

Student Feedback



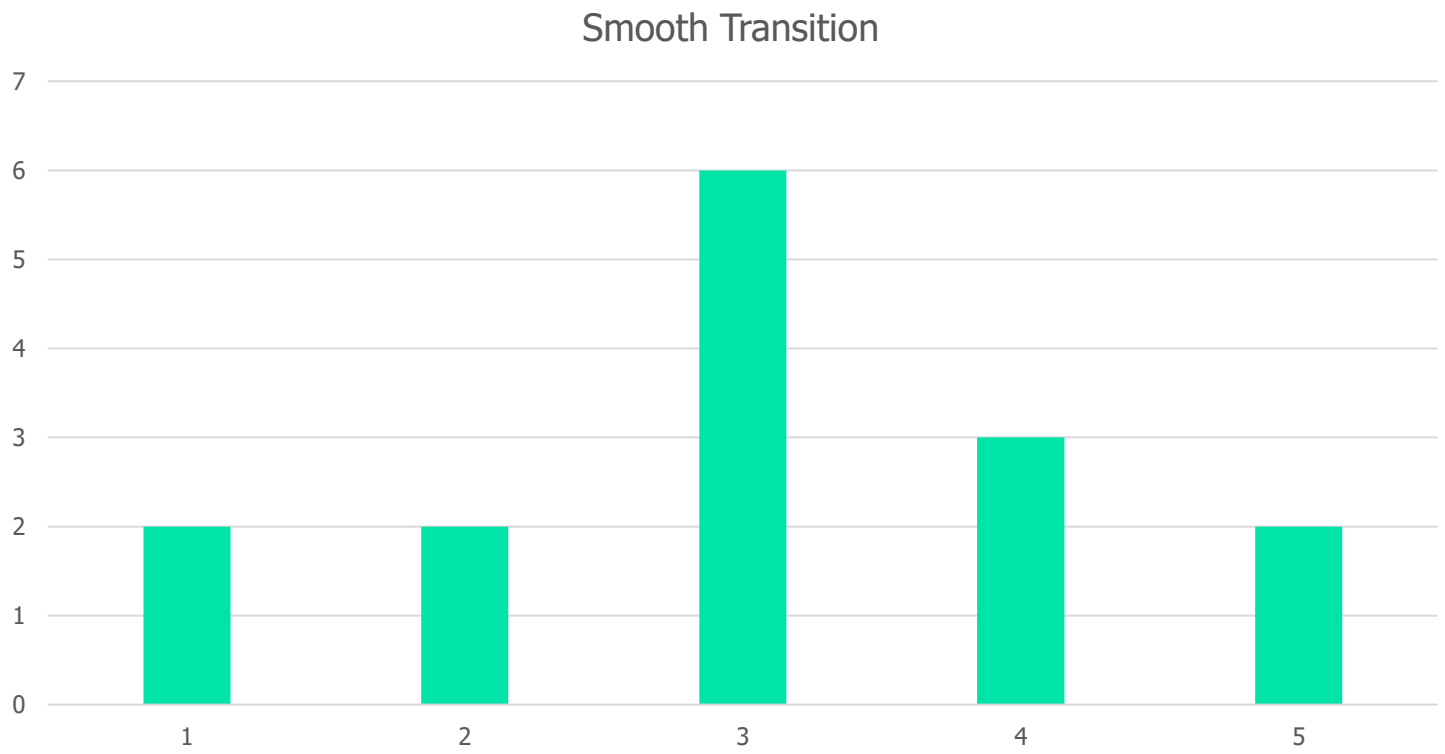
- Do you feel you are a better problem solver by knowing how to design and implement OO programs?
- 1 = Not at all
- 5 = Very much so

Student Feedback



- How intellectually stimulating is OO programming?
- 1 = Not at all stimulating
- 5 = Very stimulating

Student Feedback



- How smooth is the transition from the Racket student languages to Java?
- 1 = Not at all smooth
- 5 = Extremely smooth

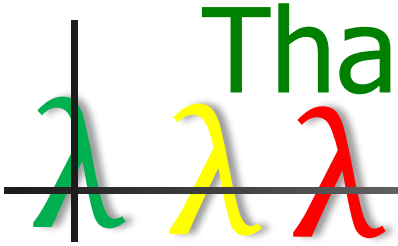
Student Feedback

- What did you like least about the course?
  - *Learning a new syntax though it gets easier with time*
  - *Lambda functions in Java. They're much more difficult to understand and implement.*
  - *Everything we learned I hated it all. I loved programming in AP CS, but I hate this.*
- What did you like most about the course
  - *Abstraction was awesome. Interfaces were cool. Creating objects were cool.*
  - *Learning to implement our knowledge into object orients design.*
  - *Solving problems with code.*

Concluding Remarks

- One can smoothly transition from a designed-based course using FP to one using OOP
  - Structures lead to classes
  - Union types lead to interfaces
  - Abstraction leads to Abstract Classes and Inheritance
- All based on type-oriented programming
- Free context for: code refactoring and data structures
  - Caveat: Other professors may complain ☹
- Students feel: empowered, better problem solvers, intellectually stimulated, and transition is mostly smoothed

Thank you for your attention!



☺ Any Questions? ☺