CS3500: Object-Oriented Design
Spring 2014

Class 5
1.21.2014

Northeastern University
College *of* Computer and Information Science

Today...

- Debugging Exercise
- Abstract syntax
- How the recipe works
- The `new` rule
- Factory method pattern
- Effective Java items
- Liskov - Chapter 3: Procedural Abstraction [19]
- Liskov - Chapter 4: Exceptions [19]

Office Hours This Week

- Tuesday, January 21: 12:30-1:25pm
- Thursday, January 23: 12:30-2:30pm
- Friday, January 24: NO OFFICE HOURS

Assignments 2 & 3

- Assignment 2
  - TestMyList.java
  - Due Tuesday, January 21, 2014 (TONIGHT)
- Assignment 3
  - MyList implementation
  - Due Friday, January 24, 2014

Abstraction Barrier

Client

- Knows the behavior of the data type
- Doesn't know how the data type was implemented, but can use the data type based on the specs

Abstraction Barrier

Implementor

- Knows the behavior of the data type
- Knows how the data type was implemented

Debugging

Writing New Code

- write tests first
- write a small amount of code
  - then test it
  - repeat

Modifying Code

- write new tests
- make a small change;
  - test it;
  - repeat

Debugging Code

- Do ***not*** make random or extensive changes to the program!
- Instead, examine the code to figure out ***what*** went wrong
  - Which tests are failing?
  - Find a test that is failing - is it repeatable? (sometimes it is not)
- ***Think*** before you change anything
- Figure out ***why*** it went wrong

Debugging ICE

- Given an assignment description and implementation, test and debug the code.
- /course/cs3500sp14/ice/2014_01_21
  - BoxOfficeDescription.txt - Description of assignment
  - BoxOffice.java - Student's implementation of assignment
  - BlackBoxTestPlan.pdf - Table for writing test cases

Expressions, Abstract Syntax, and Representations

Algebraic Laws for Addition

for all x

for all y

$0 + y = y$

$(x + 1) + y = x + (y + 1)$

Algebraic Laws for Addition

for all x

for all y

$0 + y = y$

$(x + 1) + y = x + (y + 1)$

```
  3 + 2
= (2 + 1) + 2
= 2 + (2 + 1)
= 2 + 3
= (1 + 1) + 3
= 1 + (3 + 1)
= 1 + 4
= (0 + 1) + 4
= 0 + (4 + 1)
= 0 + 5
= 5
```

Specification of the Non-Negative Integers as an Immutable ADT

```
Nat

        zero:                   ->  Nat
        succ:   Nat             ->  Nat
        add:    Nat x Nat       ->  Nat


        add (zero(), y)    =   y
        add (succ(x), y)    =   add (x, succ(y))
```

Specification of the Non-Negative Integers as an Immutable ADT

```
Nat
      zero:                  ->  Nat
      succ:   Nat            ->  Nat
      add:    Nat x Nat      ->  Nat
```

```
add (zero(), y)   =   y
add (succ(x), y)   =   add (x, succ(y))

0 = zero() = z()
1 = succ(0) = succ(zero()) = s(z())
2 = succ(1) = succ(succ(zero())) = s(s(z()))
3 = succ(2) = s(2)
4 = succ(3) = s(3)
5 = succ(4) = s(4)
```

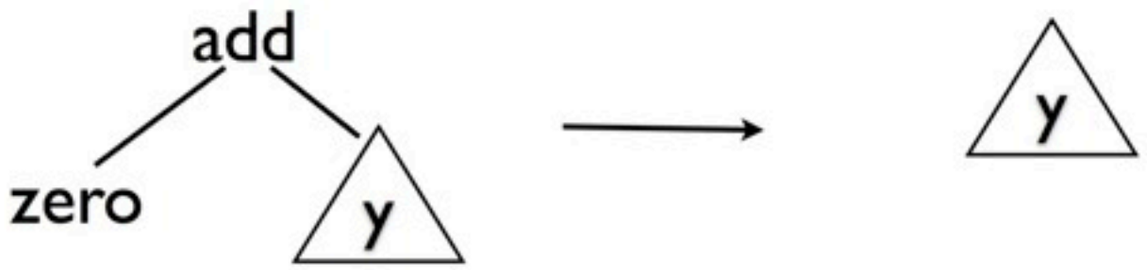
Specification of the Non-Negative Integers as an Immutable ADT

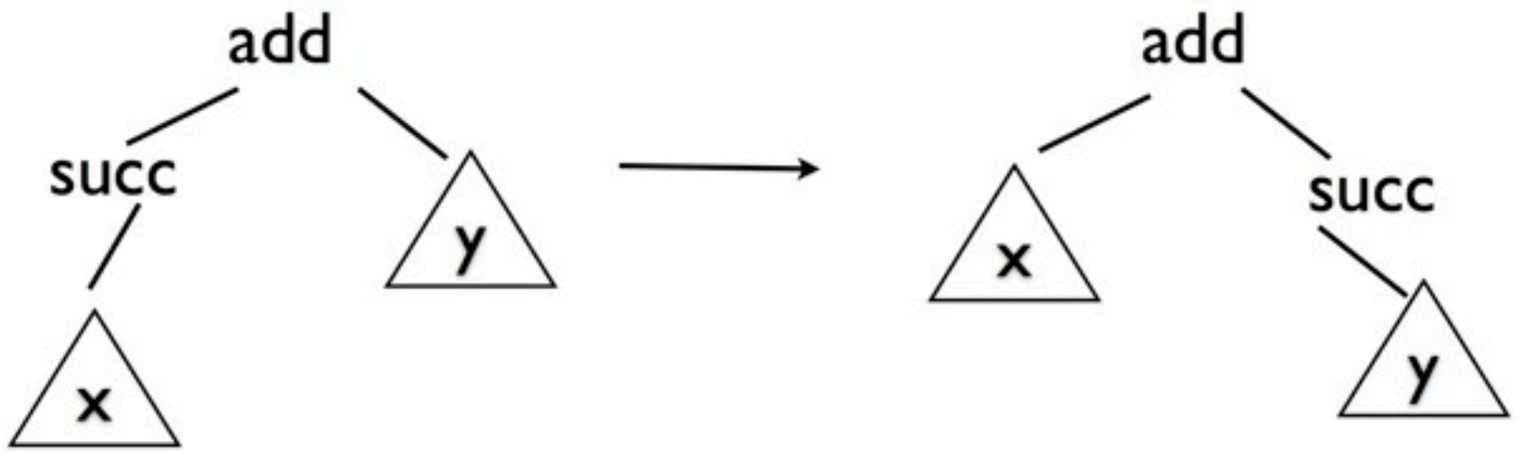
```
add (zero(), y)  =  y
add (succ(x), y)  =  add (x, succ(y))

0 = zero() = z()
1 = succ(0) = succ(zero()) = s(z())
2 = succ(1) = succ(succ(zero())) = s(s(z()))
3 = succ(2) = s(2)
4 = succ(3) = s(3)
5 = succ(4) = s(4)
```

```
    add (3, 2)
  = add (s(s(s(z()))), s(s(z())))
  = add (s(s(z())), s(s(s(z()))))
  = add (s(z()), s(s(s(s(z())))))
  = add (z(), s(s(s(s(s(z()))))))
  = s(s(s(s(s(z())))))
  = 5
```

Specification of the Non-Negative Integers as an Immutable ADT

```
add (zero(), y)   =   y
add (succ(x), y)   =   add (x, succ(y))
```





How the Recipe Works

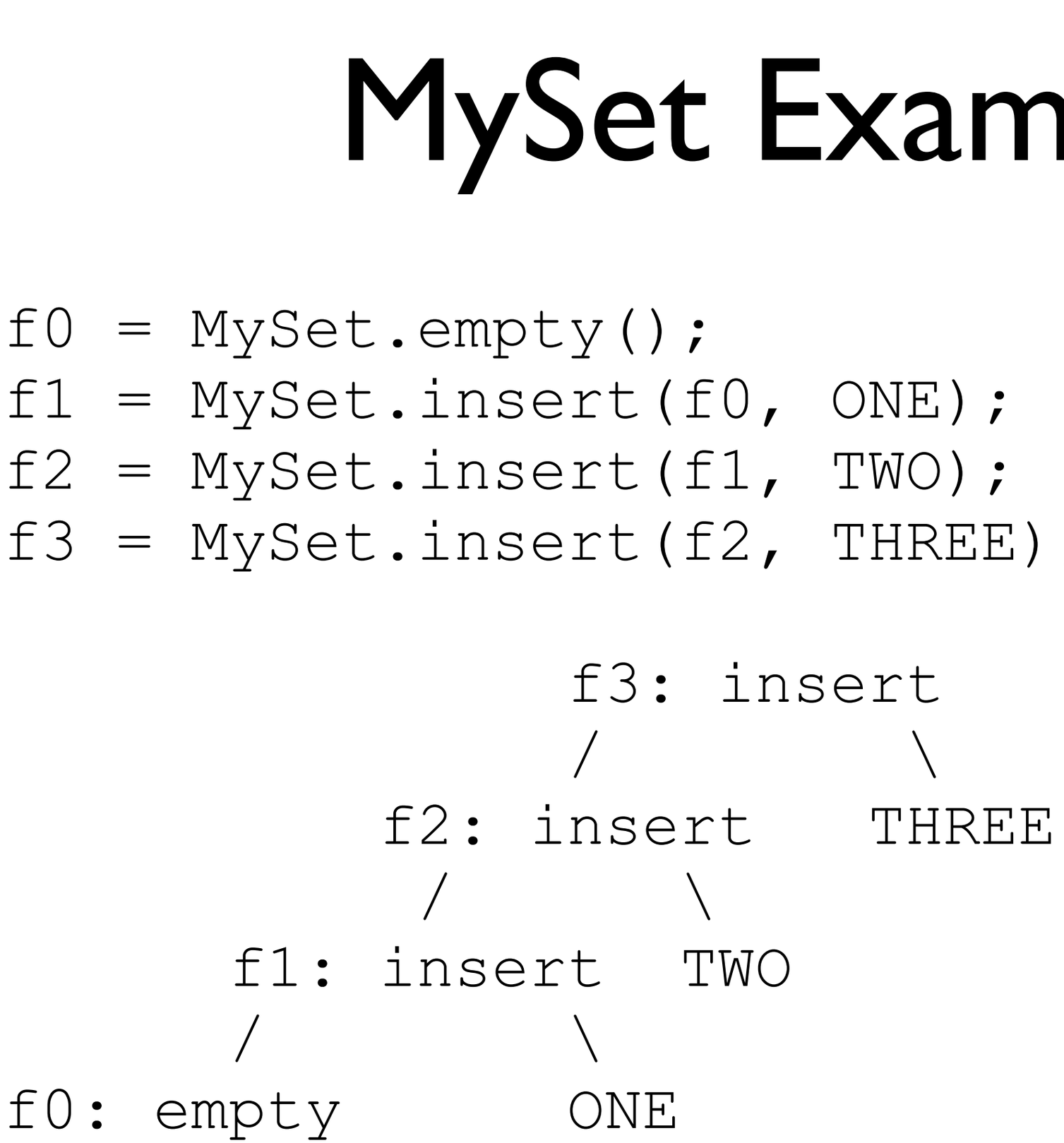
MySet Example

```
private final Long ONE = new Long(1);
private final Long TWO = new Long(2);
private final Long THREE = new Long(3);
private final Long FOUR = new Long(4);
```

```
f0 = MySet.empty();
f1 = MySet.insert(f0, ONE);
f2 = MySet.insert(f1, TWO);
f3 = MySet.insert(f2, THREE);
f4 = MySet.insert(f3, FOUR);
f5 = MySet.insert(f2, ONE);
f6 = MySet.insert(MySet.insert(f2, FOUR), THREE);
```

MySet Example

```
f0 = MySet.empty();
f1 = MySet.insert(f0, ONE);
f2 = MySet.insert(f1, TWO);
f3 = MySet.insert(f2, THREE);
```



MySet Example

```
MySet.isEmpty(MySet.empty())  =  true
MySet.isEmpty(MySet.insert(s0, k0))  =  false

f0 = MySet.empty();
f1 = MySet.insert(f0, ONE);
```

```
MySet.isEmpty(f0)
= MySet.isEmpty(MySet.empty())
= true

MySet.isEmpty(f1)
= MySet.isEmpty(MySet.insert(f0, ONE))
= false
```

```
public abstract class MySet {
    ...
    public static int size(MySet ms) {
        return ms.size();
    }
    public static boolean isEmpty(MySet ms) {
        return ms.isEmpty();
    }
    abstract int size();
    abstract boolean isEmpty();
    ...

    static class Empty extends MySet {
        ...
        int size() {
            return 0;
        }
        boolean isEmpty() {
            return true;
        }
        ...
    }
    static class Insert extends MySet {
        ...
        int size() {
            if (MySet.contains(s0, k0)) {
                return MySet.size(s0);
            }
            else {
                return 1 + MySet.size(s0);
            }
        }
        boolean isEmpty() {
            return false;
        }
        ...
    }
}
```

The `new` rule

The `new` rule

- The object creation expression new C(actual-list) creates a new object of class C and then calls that constructor in class C whose signature matches the arguments in actual-list." [Sestoft]
- Initialization

```
/** Book class */
public class Book{
    private String title; //name of the book
    private String author; //author of the book
    private int pubYear; //publication year of the book
    private String checkedOut; //person who has checked out the book
    private int location; //location of the book in the stacks
    /** Constructor - empty */
    public Book() {
        title ="";
        author = "";
        pubYear = 0;
        location = 0;
        checkedOut = "";
    }
    /** Constructor - title */
    public Book(String newTitle) {
        title = newTitle;
        author = "";
        pubYear = 0;
        location = 0;
        checkedOut = "";
    }
    /** Constructor with given values */
    public Book(String newTitle, String newAuthor, int newPubYear, int newLocation) {
        title = newTitle;
        author = newAuthor;
        pubYear = newPubYear;
        location = newLocation;
        checkedOut = "";
    }
}
```

Use of `new`

Design Patterns

Design Patterns: Elements of Reusable Object-Oriented Software

by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides

"It's a book of **design patterns** that describes simple and elegant solutions to specific problems in object-oriented software design. Design patterns capture solutions that have developed and evolved over time. Hence they aren't the designs people tend to generate initially. They reflect untold redesign and recoding as developers have struggled for greater reuse and flexibility in their software. Design patterns capture these solutions in a succinct and easily applied form."

Elements of Design Pattern

[Gamma et al.]

1. The **pattern name** is a handle we can use to describe a design problem, its solutions, and consequences in a word or two.
2. The **problem** describes when to apply the pattern.
3. The **solution** describes the elements that make up the design, their relationships, responsibilities, and collaborations.
4. The **consequences** are the results and tradeoffs of applying the pattern.

How Design Patterns Solve Design Problems

[Gamma et al.]

- Finding appropriate objects
- Determining object granularity
- Specifying object interfaces
- Specifying object implementations
- Putting reuse mechanisms to work
- Relating run-time and compile-time structures
- Designing for change

Factory Method

Purpose: Isolate clients from the representations of a data type

Factory Method Pattern

- The Factory Method pattern with static methods.
- The Factory Method pattern with factory objects.

StackInt

`StackInt.empty()`	`new StackInt()`
`StackInt.push (s, n)`	`new StackInt (s, n)`

StackInt

```
StackInt.empty()
```

```
new Empty()
```

```
StackInt.push (s, n)
```

```
new Push (s, n)
```

Factory Method: StackInt

- To create an empty stack, the client called `StackInt.empty()` instead of `new StackInt()`

- To create a non-empty stack, the client called `StackInt.push(s,n)` instead of `new StackInt(s,n)`

Effective Java Items
[Bloch]

- Item 8: Obey the general contract when overriding equals
- Item 9: Always override hashCode when you override equals
- Item 10: Always override toString
- Item 11: Override clone judiciously
- Item 13: Minimize the accessibility of classes and members
- Item 14: In public classes, use accessor methods, not public fields
- Item 15: Minimize mutability
- Item 38: Check parameters for validity
- Item 40: Design method signatures carefully
- Item 45: Minimize the scope of local variables
- Item 56: Adhere to generally accepted naming conventions

Item 8: Obey the general contract when overriding `equals`

[Bloch]

You want to leave the default method if:

- Each instance of the class is inherently unique.
- You don't care whether the class provides a "logical equality" test.
- A superclass has already overridden equals, and the superclass behavior is appropriate for this class.
- The class is private or package-private, and you are certain its equals method will never be invoked.

Item 8: Obey the general contract when overriding `equals`

[Bloch]

The equals method implements an equivalence relation. It is:

- Reflexive
- Symmetric
- Transitive
- Consistent
- For any non-null reference value x, x.equals(null) must return false.

Item 9: Always override `hashCode` when you override `equals`

[Bloch, JavaSE6]

- Whenever it is invoked on the same object more than once during an execution of an application, the `hashCode` method must consistently return the same integer, provided no information used in equals comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.
- If two objects are equal according to the `equals(Object)` method, then calling the `hashCode` method on each of the two objects must produce the same integer result.
- It is not required that if two objects are unequal according to the `equals(Object)` method, then calling the `hashCode` method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hash tables.

Item 10: Always override `toString`

[Bloch]

Item 11: Override `clone` judiciously

[Bloch]

Item 13: Minimize the accessibility of classes and members

[Bloch]

Item 45: Minimize the scope of local variables

[Bloch]

Item 14: In public classes, use accessor methods, not public fields

[Bloch]

Item 15: Minimize mutability

[Bloch]

Item 38: Check parameters for validity

[Bloch]

Item 40: Design method signatures carefully

[Bloch]

Item 56: Adhere to generally accepted naming conventions

[Bloch]