Module 5 - Reasoning about Data Abstraction

# Overview and Objectives

## OVERVIEW

In this module we will learn how to reason OO programs. In particular, we will learn and create important concepts including abstract functions and rep invariants, which are crucial for the reasoning about progarm correctness.

## COURSE LEVEL OBJECTIVES (CLO)

Upon completion of this course, you should be able to:

1. Construct modern high quality software systems and reason about them.
2. Properly define software specifications and rep-invariants.
3. Leverage immutability to properly construct threat safe programs.
4. Explain object-oriented concepts such as information hiding, encapsulation, data and type abstraction, and polymorphism.
5. Properly use exception handling
6. Identify when it is appropriate to use inheritance and generics.

## TODO MODULE LEVEL OBJECTIVES (MLO)

Upon completion of this module’s activities, you should be able to:

1. Recognize and implement data abstraction concepts for verification including, States, Abstract Functions, and Rep Invariants (CL01, CLO 2, CL04)
2. Analyze OOP correctness with respect to given specifications by reasoning over Rep Invariants and Abstract Functions (CL1, CL2, CL4)

# Module Video (Wiley-Produced w/Dan Ramos) [3-5 minutes]

# Learning Materials [~100 pages, ~3.5 hours]

## TEXTBOOK READINGS

* Barbara Liskov with John Guttag. Program Development in Java. Addison Wesley, 2001, ISBN 0-201-65768-6.
  + Chapter 5: 5.5–5.10

# Learning Unit 1 – Intro (MLO 1,2) [0.5 hour]

* Why hide implementation from client? ​
  + Client should NOT make assumptions about the implementation​
  + Makes reimplementation (maintenance) possible​
  + Protects implementation from client​

​ - Save the client from having to change their code in case of reimplementation​

* ​To implement a data abstraction we​
  + select a **representation**, or **rep**, for its objects​
  + implement the **constructors** to initialize the rep properly​
    - implement the **methods** to use/modify the rep properly​
* **Representation = rep** = instance variables (main ones) ​
* E.g., a plausible representation for an IntSet object is a vector, where each integer in the IntSet occurs as an element of the vector​
  + We could choose to have each element of the set occur exactly once in the vector (guaranteed by insert()) or allow it to occur many times. ​
  + The latter choice makes the implementation of insert() run faster but slows down remove()​
* It is important to restrict access to the instance variables to the implementation of the methods and constructors​
  + allows for modification the implementation of an abstract type **without affecting any code** that uses the type (the potentially 1000’s of clients) ​
* Rep for IntSet: (see LiskovSet.java​) ​
* private List<Integer> els; // the rep​
* Rep for Poly (see Poly.java):​
* private int[] trms; ​  
  private int deg;​

​

# Learning Unit 2 – Abstract Functions and Rep Invariants (MLO 1, 2) [~2 hour]

## Liskov 5.5: Abstraction Functions and Representation Invariants

### Abstraction function

​- The **abstraction function (AF)** maps the rep-states to the abstract-states, i.e., internal-states to client-states​

* The mapping is needed since ​
  + the contracts are written in the context of the client, and ​
  + the code is written in the context of the representation​

​

* The **rep-invariant** describes which rep-states are "good"

### What is involved?​

* Abstract State (Client State)​

​- Representation State (Internal/concrete State)​ ​- Methods (behavior)​

* Constructors (create objects)​
* Producers (return immutable object)​
* Mutators (change state)​
* Observers (report about state)

### What is "State" ? (NOTE: Flashcard/Definition materials)

* Definition:​
  + A state is an assignment of values to instance variables (at a given point in time)​
  + Need to consider all possible values for each variable​
  + State space is the cross product of possible values for each individual variable​

​- Java example:​

* Variables: ​
  + List list;​
  + int x;​
* Possible States:​
  + list = [], x = 5​
  + list = null; x = 0​
  + list = ["cat", "dog"]; x = -8​
  + list = ["cat", 1, null]; x = 0​
  + … (All possible values of list) x (All possible values of x)​

### For In class 1 discussion

// Liskov's Poly class​  
// A typical Poly is c0 + c1 x + c2 x^2 + …​  
public class Poly { ​  
private int[] trms; ​  
private int deg;​

* The state space for the abstract-state is the set of all polynomials​
* The state space for the rep-state contains all possible assignments to (deg, trms[])​

​- Example1: ​

* (2, [5, 0, 3]) is a rep-state that maps to the abstract-state 5 + 3.x2​

​ - (3, [5, 0, 3, 6]) is a rep-state that maps to the abstract-state 5 + 3.x2 + 6.x3​

* The Mapping is done using the **Abstract Function (AF)​**

### Abstraction Functions (Liskov 5.5.1)

* For each object c belonging to a type C, **AF(c)** is the state of the abstract object that c represents​

​- In **IntSet**, the AF is **many-to-one**: **many** els components map to the **same** abstract element, e.g., [1,2] and [2,1]​ are the same set ​- Since the process of abstraction involves forgetting **irrelevant information**

* it is not surprising that abstraction functions are often many-to-one​
* the order of elements in els is irrelevant (also irrelevant in the abstract element)
* Abstract state = **What the client sees​**
* Representation state = **What the implementation manipulates​**

​- The **AF** maps representation states to abstract states​

* In Java, the AF is implemented by "toString()": The AF is the specification of toString()

​- The AF should describe a "typical" abstract object. This description should be included as part of the specification. Examples:​

// Ex1: A typical IntSet is {x1, x2, ..., xn}​  
 // The abstraction function is​  
 // AF(c) = { c.els[i].intValue | 0 <= i < c.els.size }​  
  
  
// Ex2: A typical Poly is c0 + c1.x + c2.x^2 + · · ·​  
// The abstraction function is​  
// AF(c) = c0 + c1.x + c2.x^2 + · · ·​  
// where​  
// ci = c.trms[i] if 0 <= i < c.trms.size​  
// = 0 otherwise​

### Rep Invariant (Liskov 5.5.2 )

* A rep-invariant captures constraint on implementation variables​
  + All legitimate objects should satisfy the rep-invariant​
  + If the rep-invariant is not satisfied then the implementation is faulty​
  + Use JUnit to check whether they got violated​

​

* **rep-invariant = precondition for the whole class​**

​

* The rep-invariant defines the domains of the abstraction function​
* ​// The rep invariant for IntSet is:​  
  // c.els ≠ null &&​  
  // for all integers i. c.els[i] is an Integer && ​  
  // for all integers i , j. (0 <= i < j < c.els.size ⇒​  
  // c.els[i].intValue != c.els[j].intValue )​  
    
  //e.g., It is violated if els contains duplicates​

​ ​- It is fine to describe the rep invariant informally:​

// The rep invariant is:​  
// c.els != null &&​  
// all elements of c.els are Integers &&​  
// there are no duplicates in c.els​

​

* ​No trailing zeros are expected; trms[] should always have at least one element; deg must be one less than the size of trms​
* // The rep invariant for Poly is:​  
  // c.trms != null && c.trms.length >= 1 && c.deg = c.trms.length-1​  
  // && c.deg > 0 -> c.trms[deg] != 0​

​ ​

### Implementing the Abstraction Function​ and Rep Invariant 5.5.3​

* You should add the AF and rep-invariant as comments in your code​

​- You should also provide methods to implement them​ ​- In Java, AF(c) = toString()​ ​- The method that checks the rep-invariant is called **repOk()**:​

public boolean repOk( )​  
 // EFFECTS: Returns true if the rep invariant holds for this;​  
 // otherwise returns false.​

* Example for **Poly**:​
* public boolean repOk( ) {​  
   if (trms == null || ​  
  deg != trms.length - 1 ||​  
   trms.length == 0) return false;​  
   ​  
   if (deg == 0) return true;​  
   return trms[deg] != 0; ​  
  }​  
  ​  
  // The rep invariant for Poly is:​  
  // c.trms != null && c.trms.length >= 1 && c.deg = c.trms.length-1​  
  // && c.deg > 0 ⇒ c.trms[deg] != 0​  
  ​

​

* Example for \*IntSet\*​:

public boolean repOk( ) {​  
 if (els == null) return false;​  
 for (int i = 0; i < els.size( ); i++) {​  
 Object x = els.get(i);​  
 if (!(x instanceof Integer)) return false;​  
 for (int j = i + 1; j < els.size( ); j++)​  
 if (x.equals(els.get(j))) return false;​  
 }​  
 return true; ​  
}​  
​  
// The rep invariant is:​  
// c.els != null &&​  
// all elements of c.els are Integers &&​  
// there are no duplicates in c.els​

​

* The **repOk()** method is used in several ways​
  + Test programs (JUnit) can call it to check whether an implementation is preserving the rep invariant​
  + Constructors would call it to ensure that the newly initialized object is OK​
  + Automated repair tools can call it to check whether an automated fix is preserving the rep invariant​
  + Modifiers would call it​
    - Poly: the add(), mul(), and minus() routines would call it, but sub() need not since it doesn’t access the reps of objects directly, and coeff() need not since it doesn’t modify the rep. ​
    - IntSet: the mutators insert and remove would call it​
* The rep-invariant **does not need to hold all the time** during a function invocation, but it must hold at return time (see Poly.mul())​
* The AF is of interest only for legal reps, e.g.​
  + IntSet has an AF that is defined only if the els is non-null, and if all elements of the Vector are Integers​
    - Poly has an AF that is defined only if trms is non-null

## Instructor Screencast: TITLE

## Interactive Element: TITLE

## Instructor Screencast: TITLE

# Learning Unit 3 – Method Verification (MLO 1, 2) [~2 hour]

## Verification vs Validation​

* Verification​
  + Involves two descriptions: assume that one is correct and demonstrate that the other is correct w.r.t. the first​
  + In our context, assume that a method specification/contract is correct and demonstrate that the implementation is correct w.r.t. the spec​

​

* Validation​
  + Does the contract/spec captures the customer’s requirements? Is it desirable? Does is the API look good?​

​

## ​Reasoning about Data Abstractions (Liskov 5.7​)

* How to "informally argue the correctness" of a program?​
* How to "informally verify code"? ​

​- When verifying a standalone method, you assume that the precondition holds, then demonstrate that its specification is satisfied (postcondition is satisfied)​ ​- Verifying a Data Type is more **complex**, need to show that​

* The methods specifications are satisfied​
* The rep-invariant is not violated​

​

### Preserving the Rep Invariant 5.7.1​

* Show that the rep-invariant holds for objects returned by constructors ​

​- For methods, we can assume when they are called that the invariant holds for this; we must show that it holds when the method returns for this ​

// IntSet rep-invariant: LiskovSet.java​  
// c.els != null &&​  
 // for all integers i. c.els[i] is an Integer &&​  
 // for all integers i, j. ( 0 <= i < j < c.els.size => c.els[i].intValue != c.els[j].intValue )​

​

​- IntSet constructor satisfies this invariant because the newly created vector is empty​:

* public LiskovSet () { els = new ArrayList<Integer>(); }
* The isIn() method preserves it because we can assume that the invariant holds for this when isIn() is called and isIn() does not modify this​:
  + public boolean isIn (int x) { return els.indexOf(x) >= 0;}
* The same is true for size() and choose() and indexOf()​

​- Insert() preserves the invariant because the following conditions are met:​

* The invariant holds for this at the time of the call​
* The call to indexOf() by insert preserves the invariant​
* Insert() adds x to this only if x is not already in this (i.e., indexOf(x) returns –1); therefore, since this satisfies the invariant at the time of the call, it still satisfies the invariant after the call​
  + public void insert (int x) { if (els.indexOf(x) < 0) els.add(x);}

### Reasoning about Operations 5.7.2​: Satisfying the Specification

* In addition to preserving the rep-invariant, a method must do what it is supposed to do, i.e., satisfying its specification ​

​

* **Complicating factor**: specifications are written in terms of abstract objects, but the implementation manipulates concrete representations
  + need a way to relate the two
  + **use the abstraction function AF**

​

* Specifically​
  + Contract/specification is in JavaDoc​
  + Code is in Java​
  + Code and contracts are not the same … need to relate them​
    - Use the AF to map rep-state to abstract-state​ or go backward​
    - Use the AF to map abstract-state to rep-state​​

​- Let's argue that each operation in IntSet is implemented correctly:​

* **constructor()**: The IntSet constructor returns an object whose els component is an empty vector.
  + This is correct because the AF maps the empty vector to the empty set​

// A typical IntSet is {x1, x2, ..., xn}​  
// The abstraction function is​  
// AF(c) = { c.els[i].intValue | 0 <= i < c.els.size }​

* Let’s argue that each operation in IntSet is implemented correctly:​
  + **size()**: When size() is called, we know that ​
    - size of the els vector is the cardinality of the set because the AF maps the elements of the vector to the elements of the set
    - because the rep invariant, which can be assumed to hold when size() is called, ensures that there are no duplicates in els​
  + Therefore, returning this size is correct​
  + // A typical IntSet is {x1, x2, ..., xn}​  
    // The abstraction function is​  
    // AF(c) = { c.els[i].intValue | 0 <= i < c.els.size }​  
      
    // IntSet rep-invariant:​  
    // c.els != null &&​  
    // for all integers i. c.els[i] is an Integer && ​  
    // for all integers i , j. (0 <= i < j < c.els.size ⇒​  
    // c.els[i].intValue != c.els[j].intValue )​
  + remove(): ​
    - This method first checks whether the element to be removed is in the vector and simply returns if it is not.
    - This is correct because if the element isn’t in the vector, it isn’t in the set (see AF)​
    - Otherwise, the method removes the element from the vector, and again we get the right result because the rep invariant guarantees that there are no duplicates in els (see rep-invariant)​

​

// A typical IntSet is {x1, x2, ..., xn}​  
// The abstraction function is​  
// AF(c) = { c.els[i].intValue | 0 <= i < c.els.size }​  
  
  
// IntSet rep-invariant:​  
// c.els != null &&​  
// for all integers i. c.els[i] is an Integer && ​  
// for all integers i , j. (0 <= i < j < c.els.size ⇒​  
// c.els[i].intValue != c.els[j].intValue )​

* Given The Rep Invariant as an Assumption​
* Given Preconditions as Assumptions​
* Does the Postcondition Hold?​
  + Need to Map States Through Abstraction Function​

​

* Assumption: 1) contract is correct; 2) AF is correct​

​ **Abstract-state-before** -> **Apply contract** -> **Abstract-state-after​** **AF** **AF​** **Rep-state-before** -> Execute code -> Rep-state-after​ ​

* Everything in **bold** is correct​
* If applying AF on Rep-state-after yields Abstract-state-after, implies that the code is correct (since AF simply provides a different view of the same thing)​

​

# In Class 1 (MLO 1, 2) [.5 hours]

Consider Liskov's immutable Poly example, where an abstract Poly is defined as , and is implemented with one variable:

private Map<Integer, Integer> map;

Fill in example values that are mapped by the abstraction function.

Abstract State: Poly  
  
AF  
/|\  
|  
|  
|  
|----------------------------------------------------------  
|  
|  
|  
|  
  
  
  
Representation State: map

1. Identify representation states that should not be mapped.
2. Try to capture these states with a rule (that is, a rep-invariant).
3. Consider implementing the degree() method. What code would do the job? What more specific type of map would make the implementation simpler?

# In Class 2 (MLO 1, 2) [1 hours]

Consider the code:

public class Members {  
 // Members is a mutable record of organization membership  
 // AF: Collect the list as a set  
 // rep-inv1: members != null  
 // rep-inv2: members != null && no duplicates in members  
 // for simplicity, assume null can be a member...  
  
 List<Person> members; // the representation  
  
 // Post: person becomes a member  
 public void join (Person person) { members.add(person);}  
  
 // Post: person is no longer a member  
 public void leave(Person person) { members.remove(person);}

1. Analyze these 4 questions for rep-inv 1.
   1. Does join() maintain rep-inv? (Answer: YES)
   2. Does join() satisfy contract? (Answer: YES)
   3. Does leave() maintain rep-inv? (Answer: YES)
   4. Does leave() satisfy contract? (Answer: NO)
2. Repeat for rep-inv 2.
   1. Does join() maintain rep-inv? (Answer: NO)
   2. Does join() satisfy contract? (Answer: YES, but does not matter because already violate rep-inv)
   3. Does leave() maintain rep-inv? (Answer: YES)
   4. Does leave() satisfy contract? (Answer: YES)
3. Recode join() to make the verification go through. Which rep-invariant do you use? … add condition to check for duplication
4. Recode leave() to make the verification go through. Which rep-invariant do you use?

### Instructor Screencast: TITLE

### Interactive Element: TITLE

### Instructor Screencast: TITLE

Link to MP4 File

# Assignment – (MLO 1, 2) [~2 hours]

## Purpose

Understand and practice rep-invariants, contracts, tests

## Instructions

Consider following Poly example

/\*  
 \* MapPoly: Liskov's Poly class, converted to a (Tree)Map rep.  
 \* Notes:  
 \* 0) Relies on TreeMap (not just Map) for efficiently finding max nonzero coefficient  
 \* 1) Private constructor not useful, so deleted  
 \* 2) No-arg constructor implemented with 2-arg constructor  
 \* 3) Empty map represents the 0 MapPoly (very different from Poly)  
 \* 4) Uses TreeMap descending iterator to find degree(); hence no deg variable  
 \* 5) Code generally shorter.  
 \*/  
import java.util.\*;  
  
public class MapPoly {  
  
 private TreeMap<Integer, Integer> trms;  
  
 // Effects: Initializes this to be the zero polynomial  
 public MapPoly() {  
 this(0,0);  
 }  
  
 // Effects: If n < 0 throws IllegalArgumentException  
 // else initializes this to be the polynomial c\*x^n  
 public MapPoly(int c, int n) throws IllegalArgumentException {  
 if (n < 0) {  
 throw new IllegalArgumentException("MapPoly(int, int) constructor");  
 }  
 trms = new TreeMap<Integer, Integer> ();  
 if (c != 0) { trms.put(n, c); }  
 }  
  
 // Effects: returns the degree of this  
 public int degree() {  
 int result = 0;  
 if (trms.size() > 0) {  
 result = trms.descendingKeySet().iterator().next();  
 }  
 return result;  
 }  
  
 // Effects: if degree is < 0 then throws IAE;  
 // returns the coefficent of the term of this whose exponent is d (otherwise return 0)  
 public int coeff(int d) {  
 if (d < 0) throw new IllegalArgumentException("MapPoly.coeff");  
 if (trms.containsKey(d)) return trms.get(d);  
 return 0;  
 }  
  
 // Effects: If q is null throw NullPointerException  
 // else return the MapPoly this - q  
 public MapPoly sub(MapPoly q) {  
 return add(q.minus());  
 }  
  
 // Effects: return the MapPoly -this  
 public MapPoly minus() {  
 MapPoly result = new MapPoly();  
 for (Integer i : trms.keySet()) {  
 result.trms.put (i, -trms.get(i));  
 }  
 return result;  
 }  
  
 // Effects: If q is null throw NullPointerException  
 // else return the MapPoly this + q  
 public MapPoly add(MapPoly q) {  
  
 // find all the nonzero coefficients in either this or q  
 Set<Integer> nonZero = new HashSet<Integer>(q.trms.keySet());  
 nonZero.addAll(trms.keySet());  
  
 // Add the coefficients together; store the nonzero results  
 MapPoly result = new MapPoly();  
 for (Integer i : nonZero) {  
 int newCoeff = coeff(i) + q.coeff(i);  
 if (newCoeff != 0) result.trms.put(i, newCoeff);  
 }  
 return result;  
 }  
  
 // Effects: If q is null throw NullPointerException  
 // else return the MapPoly this \* q  
 public MapPoly mul(MapPoly q) {  
 MapPoly result = new MapPoly();  
  
 for (Integer i: trms.keySet()) {  
 for (Integer j: q.trms.keySet()) {  
 result = result.add(new MapPoly (coeff(i) \* q.coeff(j), i+j));  
 }  
 }  
 return result;  
 }  
  
 @Override public String toString() { // Note that TreeMap gets the order right  
 String r = "MapPoly:";  
  
 if (trms.size() == 0) { // special case for empty Map  
 r += " " + 0;  
 }  
  
 for (Integer i: trms.keySet()) {  
 if (coeff(i) < 0) { r += " - " + -coeff(i) + "x^" + i; }  
 else { r += " + " + coeff(i) + "x^" + i; }  
 }  
 return r;  
 }  
  
 public static void main(String[] args) {   
 System.out.println("Hello");  
 MapPoly mp = new MapPoly();  
 System.out.println("MapPoly mp = " + mp);  
 mp = mp.add(new MapPoly(3,5));  
 System.out.println("MapPoly mp = " + mp);  
 mp = mp.add(new MapPoly(-3,5));  
 System.out.println("MapPoly mp = " + mp);  
 mp = mp.add(new MapPoly(-3,5));  
 System.out.println("MapPoly mp = " + mp);  
 mp = mp.add(new MapPoly(-2,2));  
 System.out.println("MapPoly mp = " + mp);  
 System.out.println("MapPoly -mp = " + mp.minus());  
 System.out.println("MapPoly mp\*mp = " + mp.mul(mp));  
 }  
}

1. Implement repOk().
2. Introduce a fault (i.e. "bug") that breaks the rep-invariant. Try to do this with a small (conceptual) change to the code. Show that the rep-invariant is broken with a JUnit test.
3. Analyzed your bug with respect to the various contracts/methods in Poly. Are all/some/none of the contracts violated?
4. Do you think your fault is realistic? Why or why not?

Your deliverable should be a **story**. Take screenshots (e.g. of failing JUnit tests) as necessary to make your case.

### Grading Criteria

* Correctness of solution
* Clarity of story

Note: If your group had trouble with the previous assignment, feel free to appeal to your classmates to post a sample solution on Piazza.

## Deliverable

* Submit a .java file for your implementation.

## Due Date

Your assignment is due by Sunday 11:59 PM, ET.

# Module 1 Quiz (MLO 1, 2) [~.5 hour]

## Purpose

Quizzes in this course give you an opportunity to demonstrate your knowledge of the subject material.

## Instructions

Suppose C is an abstract data type that has two String fields:

class C {  
 private String s;  
 private String t;  
 ...  
}

Assuming you don’t know anything about C, which of the following might be statements in a rep invariant for C?

1. s contains only letters
2. s.length() == t.length()
3. s represents a set of characters
4. s is the reverse of t
5. s+t

**Solution**

1. No (too trivial)
2. Yes
3. No (too trivial)
4. Yes
5. No, not even an invariant (not a bool)

The quiz is 20 minutes in length. The quiz is closed-book.

## Deliverable

Use the link above to take the quiz.

## Due Date

Your quiz submission is due by Sunday 11:59 PM, ET.