Module 4 - Data Abstraction; Mutability

# Overview and Objectives

## In this module we learn about data abstraction, which allows us to abstract how objects are implemented to how they behave. We will learn how to specify and implement data abstractions. We will also explore and learn how to use mutabable and immutable data types.

## 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.

## MODULE LEVEL OBJECTIVES (MLO)

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

* explain and demonstrate how to create, implement, and use data type (CLO 1, CL02, CLO4)
* Distinguish and transform between mutable and immutable types (CLO 3)

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

# Learning Materials [~100 pages, ~5.5 hours]

## TEXTBOOK READINGS

* Barbara Liskov with John Guttag. Program Development in Java. Addison Wesley, 2001, ISBN 0-201-65768-6.
  + Chapter 5: 5.1–5.4, 5.8
* Joshua Bloch. Effective Java. Third Edition. Addison-Wesley Professional, 2017, ISBN 978-0-13-468599-1.
  + Item 17

# Learning Unit 1 – Data Abstraction (MLO 1) [~2.5 hour]

* This module discusses the most important abstraction mechanism, **data abstraction**.
* Data abstraction allows us to abstract from the **details of how data objects are implemented** to **how the objects behave**.
  + This focus on the behavior of objects forms the basis of object-oriented programming.
  + allows us to extend the programming language with new data types.
    - E.g., Polynomials in a symbolic manipulation system, and matrices in defining a package of numeric functions.
  + **Data Abstraction = Objects + Operations**

## Specifications for Data Abstraction (Liskov 5.1)

* specification define the behavior the the new type (or class in Java)
  + the specification consists of explaining what the operations do.
* Each class defines a type by defining a name for the type, a set of constructors, and a set of instance methods or methods.

visibility class dname {  
//OVERVIEW: A brief description of the behavior of the type′s objects goes here.  
  
  
//constructors  
//specs for constructors go here  
  
  
//methods  
//specs for methods go here  
}

* The specification has three parts.
  + The overview gives a brief description of the data abstraction, including a way of viewing the abstract objects in terms of “well-understood” concepts.
  + The constructors part of the specification defines the constructors that initialize new objects The methods part defines the methods that allow access to the objects once they have been created.
* Similar to specifications of methods/procedures, specifications for data abstractions take the form of comments in the code.

## Using Data Abstraction

The following example shows how to use data abstractions.

public static Poly diff (Poly p) throws NullPointerException {  
 //EFFECTS: If p is null throws NullPointerException  
 // else returns the Poly obtained by differentiating p.  
 Poly q = new Poly ( );  
 for (int i = 1; i <= p.degree( ); i++)  
 q = q.add(new Poly(p.coeff(i)\*i, i - 1));  
 return q;  
 }  
  
  
 public static IntSet getElements (int[ ] a)  
 throws NullPointerException {  
 // EFFECTS: If a is null throws NullPointerException else returns a set  
 //containing an entry for each distinct element of a.  
 IntSet s = new IntSet( );  
 for (int i = 0; i < a.length; i++) s.insert(a[i]);  
 return s;  
}

* The diff method returns a new Poly that is the result of differentiating its argument Poly.
* The getElements routine returns an IntSet containing the integers in its array argument a; there are no duplicates in the returned set (since sets do not contain duplicates) even if there are duplicates among the elements of a.
* These methods are written based on the specifications of the used abstractions and can use only what is described in the specifications.
  + not able to access the implementation details of the abstract objects since
  + can use methods to access object state and to modify that state if the object is mutable
  + can use constructors to initialize new objects.

## Implementing Data Abstraction (Liskov 5.3)

* 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 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
  + This allows you to modify the implementation of an abstract type without affecting any code that uses the type (the potentially 1000’s of clients)

//Rep for IntSet:  
private List<Integer> els; // the rep  
// see LiskovSet.java

//Rep for Poly:  
private int[] trms;   
private int deg;  
see Poly.java // immutable

## Instructor Screencast: TITLE

# Learning Unit 2 – Mutability (MLO 2) [~2.5 hour]

## Basic Mutability (Liskov 5.8, Bloch Item 17)

* A type is mutable if it has mutator methods (or has non-private access control); otherwise, it is immutable
* Some types should be immutable by nature, e.g., Integers, Polynomials etc. But in many cases mutable types are needed
* Mutable types enable sharing of data and are generally good for performance
* Immutable types are good for safety, reliability, and concurrency
* However, they degrade performance since they must be created and discarded frequently (i.e., garbage collection must occur frequently)

### Data Abstraction Operation Categories

* **Creators**
  + Create objects of a data abstraction (construtors, possibly other/factory)
* **Mutators**
  + Modify objects of their type
  + Typically used in **mutable** data types
* **Observers**
  + Take objects of their type as inputs and return results of other types (e.g., getters)
* **Producers**
  + Create/return objects of their type based on existing objects
  + Typically used in **immutable** data types (producers are the counterpart of mutators)

### Examples Specifications for Data Abstraction

1. Polynomials – (immutable, from Liskov)

* /\*  
   \* Liskov's Poly class, as typed in out of her text  
   \* A typical Poly is c0 + c1 x + c2 x^2 + ...   
   \* c0 : integers   
   \* deg: non-neg int   
   \* 1 variable .. (3\*x\*y not allowed)   
    
    
   \* (deg=2, terms=[5,0,3]) -> 5+3\*x^2  
   \* (deg=2, terms=[5,0,3,0,0,0,0...] -> 5+3\*x^2+0\*x^3: is not allowed in this implementation  
   \*/  
    
    
  public class Poly {  
    
   private int[] trms;  
   private int deg;  
    
   // Effects: Initializes this to be the zero polynomial  
   public Poly() {  
   trms = new int[1]; trms[0] = 0;  
   deg = 0;  
   }  
    
   // Effects: If n < 0 throws IllegalArgumentException  
   // else initializes this to be the polynomial c\*x^n  
   public Poly(int c, int n) throws IllegalArgumentException {  
   if (n < 0) {  
   throw new IllegalArgumentException("Poly(int, int) constructor");  
   }  
   if (c == 0) {  
   trms = new int[1]; trms[0] = 0;  
   deg = 0;  
   return;  
   }  
   trms = new int[n+1];  
   for (int i=0; i < n; i++) {  
   trms[i] = 0;  
   }  
   trms[n] = c;  
   deg = n;  
   }  
    
   private Poly (int n) {  
   trms = new int[n+1];  
   deg = n;  
   }  
    
   // Effects: returns the degree of this  
   public int degree() {  
   return deg;  
   }  
    
   // Effects: returns the coefficent of the term of this whose exponent is d  
   public int coeff\_book(int d){  
   if (d < 0 || d > deg) return 0 else return terms[d];  
   }  
    
   // @throws IAE if d < 0  
   // Effects: returns the coefficent of the term of this whose exponent is d  
   public int coeff(int d) {  
   if (d < 0) throw new IAE();  
   return (d > deg) ? 0 : trms[d];  
   }  
    
   // Effects: If q is null throw NullPointerException  
   // else return the Poly this - q  
   public Poly sub(Poly q) throws NullPointerException {  
   return add(q.minus());  
   }  
    
   // Effects: return the Poly -this  
   public Poly minus() {  
   Poly r = new Poly(deg);  
   for (int i=0; i <= deg; i++) {  
   r.trms[i] = -trms[i];  
   }  
   return r;  
   }  
    
   // Effects: If q is null throw NullPointerException  
   // else return the Poly this + q  
   public Poly add(Poly q) {  
   Poly la, sm;  
   if (deg > q.deg) {  
   la = this; sm = q;  
   }  
   else {  
   la = q; sm = this;  
   }  
   int newdeg = la.deg;  
   if (deg == q.deg) {  
   for (int k = deg; k > 0; k--) {  
   if (trms[k] + q.trms[k] != 0) {  
   break;  
   }  
   else {  
   newdeg--;  
   }  
   }  
   }  
   Poly r = new Poly(newdeg);  
   int i;  
   for (i = 0; i <= sm.deg && i <= newdeg; i++) {  
   r.trms[i] = sm.trms[i] + la.trms[i];  
   }  
   for (int j = i; j <= newdeg; j++) {  
   r.trms[j] = la.trms[j];  
   }  
   return r;  
   }  
    
   // Effects: If q is null throw NullPointerException  
   // else return the Poly this \* q  
   public Poly mul(Poly q) throws NullPointerException {  
    
   if ((q.deg == 0 && q.trms[0] == 0) || (deg == 0 && trms[0] == 0)) {  
   return new Poly();  
   }  
    
   Poly r = new Poly(deg + q.deg);  
   r.trms[deg + q.deg] = 0;  
   for (int i = 0; i <= deg; i++) {  
   for (int j = 0; j <= q.deg; j++) {  
   r.trms[i+j] += trms[i]\*q.trms[j];  
   }  
   }  
   return r;  
   }  
    
   public String toString() { //python:\_\_str\_\_  
   String r = "Poly:";  
    
   if (deg == 0 || trms[0] != 0) {  
   r += " " + trms[0];  
   }  
    
   for (int i = 1; i <= deg; i++) {  
   if (trms[i] < 0) {  
   r += " - " + -trms[i] + "x^" + i;  
   }  
   else if (trms[i] > 0) {  
   r += " + " + trms[i] + "x^" + i;  
   }  
   }  
   return r;  
   }  
  }

1. Sets of integer primitives (mutable, from Liskov)

* import java.util.\*;  
  /\*  
   \* Liskov's IntSet example (called LiskovSet), with minor changes  
   \* This version doesn't include AF or RI  
   \*/  
    
  public class LiskovSet {  
    
   // Overview: LiskovSets are unbounded, mutable sets of integers  
   private List<Integer> els; // the rep  
    
   // constructor  
   // EFFECTS: Intitializes this to be empty  
   public LiskovSet () { els = new ArrayList<Integer>(); }  
    
   // methods  
   // MODIFIES this  
   // EFFECTS: Adds x to the elements of this  
   public void insert (int x) {  
   if (els.indexOf(x) < 0) // could also self-use isIn()  
   els.add(x);  
   }  
    
   // MODIFIES this  
   // EFFECTS: Removes x from this  
   public void remove (int x) {  
   int index = els.indexOf(x);  
   if (index < 0) return;  
   els.set(index, els.get(els.size()-1));  
   els.remove(els.size()-1);  
   }  
    
   // EFFECTS: Returns true if x is in this else returns false  
   public boolean isIn (int x) {  
   return els.indexOf(x) >= 0; // could also use contains()  
   }  
    
   // EFFECTS: Returns the cardinality of this  
   public int size () {  
   return els.size();  
   }  
    
   // EFFECTS: If this is empty throw IllegalStateException  
   // else returns an arbitrary element of this  
   public int choose () {  
   return els.get(els.size()-1);  
   }  
  }

1. Complex numbers (immutable, from Bloch)

* import java.util.\*;  
    
  public final class Complex {  
   private final double re;  
   private final double im;  
    
   public Complex (double re, double im) {  
   this.re = re;  
   this.im = im;  
   }  
    
   // Accessors with no corresponding mutators  
   public double realPart() { return re; }  
   public double imaginaryPart() { return im; }  
    
   public Complex add (Complex c ) {  
   return new Complex (re + c.re, im + c.im);  
   }  
    
   public Complex subtract (Complex c ) {  
   return new Complex (re - c.re, im - c.im);  
   }  
    
   public Complex multiply (Complex c ) {  
   return new Complex (re \* c.re - im \* c.im,  
   re \* c.im + im \* c.re);  
   }  
    
   public Complex divide (Complex c ) {  
   double tmp = c.re \* c.re + c.im \* c.im;  
   return new Complex ((re \* c.re - im \* c.im) / tmp,  
   (re \* c.im + im \* c.re) / tmp);  
   }  
    
   @Override public boolean equals (Object o) {  
   if (o == this) return true;  
   if (!(o instanceof Complex)) return false;  
   Complex c = (Complex) o;  
    
   // See Bloch page 43 to find out why to use compare() instead of ==  
   return Double.compare(re, c.re) == 0 &&  
   Double.compare(im, c.im) == 0;  
   }  
    
   @Override public int hashCode () {  
   int result = 17 + hashDouble(re);  
   result = 31 \* result + hashDouble(im);  
   return result;  
   }  
    
   private int hashDouble (double val) {  
   long longBits = Double.doubleToLongBits(val);  
   return (int) (longBits ^ (longBits >>>32));  
   }  
    
   @Override public String toString() {  
   return "(" + re + " + " + im + "i)";  
   }  
    
    
   public static void main(String[] args) {  
   Complex c1 = new Complex (2, 3);  
   Complex c2 = new Complex (4, 5);  
   System.out.println("c1 : " + c1);  
   System.out.println("c2 : " + c2);  
   System.out.println("c1 + c2 : " + c1.add(c2));  
   System.out.println("c1 - c2 : " + c1.subtract(c2));  
   System.out.println("c1 \* c2 : " + c1.multiply(c2));  
   System.out.println("c1 / c2 : " + c1.divide(c2));  
   System.out.println("c1.equals(c2): " + c1.equals(c2));  
   System.out.println("c1.hashCode(): " + c1.hashCode());  
   }  
    
  }

### Mutable/Immutable Transformation

* Consider the **mutable** Stack example from Bloch
* /\*\*  
   \* Bloch's Generic Stack example  
   \* SWE 619  
   \*/  
    
  import java.util.\*;  
    
  public class Stack <E> {  
    
   private E[] elements;  
   private int size = 0;  
   private static final int DEFAULT\_CAPACITY = 16;  
    
    
   @SuppressWarnings("unchecked") // All items in elements[] are of type E  
   public Stack() {   
   // this.elements = new E[DEFAULT\_CAPACITY]; // doesn't compile  
   this.elements = (E[]) new Object[DEFAULT\_CAPACITY];  
   }  
    
   public void push (E e) {  
   ensureCapacity();  
   elements[size++] = e;  
   }  
    
   public E pop () {  
   if (size == 0) throw new IllegalStateException("Stack.pop");  
   E result = elements[--size];  
   elements[size] = null;  
   return result;  
   }  
    
   public boolean isEmpty() {  
   return size == 0;  
   }  
    
   // public void pushAll(Iterable<E> src) { // doesn't compile  
   public void pushAll(Iterable<? extends E> src) {  
   for (E e: src) {  
   push(e);  
   }  
   }  
    
   // public void popAll(Collection<E> dst) { // doesn't compile  
   public void popAll(Collection<? super E> dst) {   
   while (!isEmpty()) {  
   dst.add(pop());  
   }  
   }  
    
   @SuppressWarnings("unchecked") // All items in elements[] are of type E  
   private void ensureCapacity() {  
   if (elements.length == size) {  
   Object oldElements[] = elements;  
   // elements = new E[2\*size + 1]; // doesn't compile  
   elements = (E[]) new Object[2\*size + 1];  
   System.arraycopy(oldElements, 0, elements, 0, size);  
   }  
   }  
    
    
   public static void main(String[] args) {  
   // Simple exercise to push/pop cmd line args  
   Stack <String> s = new Stack <String>();  
   for (String arg : args)  
   s.push(arg);  
   while (!s.isEmpty() )  
   System.out.println(s.pop().toUpperCase());  
    
   // Exercise for pushAll, popAll  
   Stack <Number> s1 = new Stack<Number>();  
   Integer i = 1; s1.push(i);  
   i = 2; s1.push(i);  
    
   Collection<Integer> integers = new HashSet<Integer> ();  
   integers.add(2);  
   integers.add(3);  
   s1.pushAll(integers);  
    
   Collection<Object> result = new ArrayList<Object> ();  
   s1.popAll(result);  
    
    
   for (Object n : result)   
   System.out.println(n);  
   }  
    
  }
* **Goal**: Transform to an immutable version. Doesn’t seem natural, but it is a design decision (recall, modifying a string returns a new string!)
* ImmutableStack s = new ImmutableStack();  
  s = s.push(“cat”); // push() returns a stack!! Why?  
  s = s.push(“dog”);

### Transform Mutator and Producer

* Consider a void mutator method in class C: public void mutator1(...
  + Corresponding producer method: public C producer1(...) // same params different implementation/return
* Consider a non-void mutator method in class C: public S mutator2(...)
  + Corresponding observer/producer methods are:
  + public S observerPart(…)   
    public C producerPart(…)
  + Note that non-void mutator needs to be split into **two** methods. Example: pop() in Stack vs. pop()/producer, top()/observer in ImmutableStack

### Typical Transformation

* Typical method in mutable class Foo: public void foo(T1 t1, T2, t2, ...) {modify “this”}
  + Immutable version of Foo:
  + public Foo foo(T1 t1, T2, t2, …) {  
     Foo f = ...  
     ...  
     return f;  
    }

## Disadvantage

* Typical approach:
  + Provide immutable class
  + Provide mutable companion
  + Clients choose on performance needs
* Example in Java Library:
  + String (Immutable)
  + StringBuilder (Companion Mutable Class)
* How to speed up immutability?
  + Static factories can cache frequently used items to speed up immutability (e.g., cashing strings)

<https://docs.oracle.com/javase/7/docs/api/java/lang/String.html#intern>()

## Bloch-Item 17 - Complex (Complex.java)

* Notice how the arithmetic operations create and return a new Complex instance rather than modifying this instance.
* pattern is known as the **functional approach** because methods return the result of applying a function to their operand, without modifying it (scala, javascript).
  + Contrast it to the procedural or **imperative approach** in which methods apply a procedure to their operand, causing its state to change.
* Note that the method names are prepositions (such as plus) rather than verbs (such as add).

## Advantages

* Immutable objects are **inherently thread-safe**; they require **no synchronization** (which might cause deadlocks).
* They cannot be corrupted by multiple threads accessing them concurrently.
* Immutable objects can be **shared freely**.

## Disadvantages

* The major disadvantage of immutable classes is that they **require a separate object for each distinct value**.
* For example, suppose that you have a million-bit BigInteger and you want to change its low-order bit: BigInteger moby = ...; moby = moby.flipBit(0);
* The flipBit method creates a new BigInteger instance, also a million bits long, that differs from the original in only one bit. This is costly operation w.r.t. time and space
* This is why the Java Library provides an immutable companion to mutable type: BigInteger vs BitSet; String vs StringBuilder

## How to make a class immutable

* Don’t provide any mutators
* Make all fields private and final
* Do not allow the return of aliases to mutable attributes (use Defensive Copying)
* Make the class final so that it cannot be extended, or make all the getters final to prevent overriding (Missed by Liskov): given A, B, an object of A could be maliciously substituted by an object of B. So calling a getter on a B might return malicious data, while you are assuming that you are dealing with the safe A.

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

Consider a simple generic Queue implementation.

public class Queue <E> {  
  
 private List<E> elements;  
 private int size;  
  
 public Queue() {  
 this.elements = new ArrayList<E>();  
 this.size = 0;  
 }  
  
 public void enQueue (E e) {  
 elements.add(e);  
 size++;  
 }  
  
 public E deQueue () {  
 if (size == 0) throw new IllegalStateException("Queue.deQueue");  
 E result = elements.get(0);  
 elements.remove(0);  
 size--;  
 return result;  
 }  
  
 public boolean isEmpty() {  
 return size == 0;  
 }  
}

1. Rewrite Queue to be **immutable** (i.e., enQueue and deQueue) Keep the representation variables elements and size.
2. Add contracts to all methods.

# Assignment – (MLO 1, 2) [~2 hours]

## Purpose

Get familiar with mutability and review method specifications.

## Instructions

Given the following MapPoly class, a map-based version of Liskov's Poly

/\*  
 \* 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));  
 }  
}

* Rewrite MapPoly so that it is *mutable*. Keep the same representation.
* Rewrite the overview, the method signatures, the method specifications, and the methods themselves. You do not need to rewrite the abstraction function and representation invariant for this exercise.
  + Turn in a **story**. This means that it is possible to grade your assignment simply by reading it, as if it were part of a textbook. In particular, every place you make a decision to change something in the code (or not), you should have a description of what you did (or didn't do) and why you did (or didn't do) it.
  + Remember that part of your group is responsible for synthesizing a solution, and part of your group is responsible for checking the result.

### Grading Criteria

* Correct transformation of Poly
* Clarity of your story.
* Reasonable division of synthesis vs. checking.

## Deliverable

* Submit a .java file for your implementation.

## Due Date

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

# Module 4 Quiz (MLO 1, 2) [~.5 hour]

## Purpose

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

## Instructions

**Immutatiblity**: Consider a partial version of a *mutable* Stack class:

public class Stack {  
  
 private Object[] elements;  
 private int size = 0;  
  
 public Stack() { this.elements = new Object[0]; }  
  
 public void push (Object e) {  
 ensureCapacity(); // implementation omitted from quiz  
 elements[size++] = e;  
 }  
  
 public Object pop () {  
 if (size == 0) throw new IllegalStateException("Stack.pop");  
 Object result = elements[--size];  
 elements[size] = null;  
 return result;  
 }  
}

Also consider the following client interaction with Stack

Stack s = new Stack();  
s.push("cat");  
s.push("dog");  
System.out.println(s.pop());

**Q**: What methods would be needed to convert the client interaction into an equivalent interaction with an *immutable* version called IStack? For each method, give **just the method header** - not the contract or implemetation. Note: the mutable version of pop has two behaviors: modifies the Stack **and** returns the top element.

public class IStack{ public IStack push(Object e){ // return new IStack() …. }

public IStack pop (){ // return new IStack object }

public Object top(){ // return the top of the stack }

}

The quiz is 30 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.