Module 7 - Type Abstraction

# Overview and Objectives

## OVERVIEW

This modules discusses **types** as a way to enhance the use of data abstraction by defining families (or classes) of related types. All members of a family have similar behavior: they all have certain methods, and calls on those methods behave in similar ways. Family members may differ by extending the behavior of the common methods or by providing additional methods. In particular, these concepts are captured in the classical Liskov Substitution Principle, which will be discussed and practiced throughout this module. We will also learn about dynamic dispatching, a powerful mechanism in a language such as Java that determines at runtime which methods of a class/type to use.

## 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. Explain the key concepts of typing hierachy and Liskov Principle of Substitution (CLO 1, CLO 4)
2. Construct and use specifications according to Liskov Principle of Substitution (CLO 2, CLO 4)
3. Explain the key concepts of Dynamic Dispatching (CLO 1, CLO 4)

# 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 7: Type Hierachy

# Learning Unit 1 – Type Hierachy (MLO 1) [~2 hours]

* Type hierarchy is used to define type families consisting of a

supertype and its subtypes.

* The hierarchy can extend through many levels
* **Subtypes**: Some type families are used to provide multiple implementations of a type.
  + The **subtypes** provide **different implementations** of their supertype
  + subtypes **extend the behavior** of their supertype, (e.g., providing extra methods)
* The **Liskov substitution principle** requires that \*subtypes behave

in accordance with the specification of their supertype\*

## Liskov Substitution Principle (SP)

"In any client code, if the supertype object is substituted by a subtype object, the client’s expectations will still be satisfied"

* SP: client is interfacing with a supertype, someone replaces the supertype with a subtype, then the client should still be happy w.r.t. the specification of supertype
* SP is the essence of Inheritance. The goal of inheritance is code reuse, but it could be problematic if not used properly

In C++ you could have a superclass/subclass that is not a supertype/subtype, i.e. the "is-a" relationship does not hold (not just conceptually)

* This cannot happen in Java

## Are Subclasses Subtypes?

* The subtyping test (Liskov Substitution Principle)

– Code written for A should work for any subclass of A

class A{int x;};  
class B: public A{};  
void main()  
{  
A obj;  
obj.x;  
}  
  
  
class A {public: int x;};  
class B: public A{};  
void main()  
{  
B b;  
b.x; // legal  
}  
//B "is-a" A -> true  
  
class A{public: int x};  
class B: private A{};  
  
void main()  
{  
B b;  
b.x; // illegal  
}  
// B "is-a" A -> false  
//B is not a subtype of A  
//Since B does not have all the attributes of A

* Subtype : public in A must be public in B

## Meaning of subtypes

* Subtypes behavior must support supertype behavior – (SP)
* If using a subtype, need to make sure that SP holds… some properties to worry about:
  1. Signature Rule (easy to satisfy… Exceptions?)
  2. Methods Rule (main concern)

### Signature Rule

* Subtypes must have all methods of supertype
* Signatures of methods must be compatible with supertype signature
  + Return types identical
* Guaranteed by Java compiler (this is why it is not a major concern)
* Caution: Overriding vs. overloading (see Bloch)
  + public boolean equals(Foo foo) {…} // use @Override
  + public boolean equals(Object foo) {…}
* Exceptions
  + Signature Rule allows Subtype to omit throwing exceptions What if the contract of the supertype requires an

exception to be thrown … problem!

### Methods Rule

* Compiler is not totally in control here
* Need to make sure that the contracts of subtype & supertype are in sync
* Suppose SortedIntSet extends IntSet
* IntSet x = new IntSet();  
  IntSet y = new SortedIntSet(); // cast needed?  
  x.insert(3); //What is this\_post?... 3 is inserted/unique  
  y.insert(3); //What is this\_post?... 3 is inserted/unique + sorted: is postcond of supertype satisfied?
* Cannot take away methods!

1. Subtype API should at least be equal or greater than supertype API. Java guarantees that all supertype methods are there in the subtype… but "omitting a method exception" violates this property.
2. Must maintain the contract!
   1. Only care about overridden method: client can only see supertype methods (none of the new added methods)
   2. Precondition rule: Must satisfy supertype precondition + Subtype is allowed to weaken the precondition!
   3. Post condition rule: Must satisfy supertype postcondition Subtype is allowed to strengthen the post condition
3. Precondition rule
   * Subtype is allowed to **weaken** the precondition!
     + Super //Requires: x > 5
     + Case 1: Sub //Requires x > 6
     + Case 2: Sub // Requires x > 4
     + x>5 => x>4 : Yes
     + x>5 => x>6? : No
     + // So Case 2, x > 4 is what we want
     + // Note that the precondition of True is weakest
4. Post condition rule
   * Subtype is allowed to strengthen the postcondition
     + Super // Effects: returns y <= 5
     + Case1: Sub //Effects: returns y <= 4 (allows less)
     + Case2: Sub: //Effects: returns y <= 6 (allows more)
     + y <=4 > y < 5 : Yes
     + y <= 6 > y < 5 : No

### Stronger vs weaker specs

* This spec for find():
* static int findExactlyOne(int[] a, int val)  
   requires: val occurs exactly once in a  
   effects: returns index i such that a[i] = val
* can be replaced with:
* static int findOneOrMore,AnyIndex(int[] a, int val)  
   requires: val occurs at least once in a  
   effects: returns index i such that a[i] = val
  + which has a weaker precondition.
  + Old client code works! since it guarantees that only one occurs

12

* Also, findExactlyOne can be replaced with:
* static int findOneOrMore,FirstIndex(int[] a, int val)  
  requires: val occurs at least once in a  
  effects: returns lowest index i such that a[i] = val
  + which has a stronger postcondition.
* Old client code works?
  + Client guarantees that one occurs so the lowest=only will get returned
* What about this specification:
* static int findCanBeMissing(int[] a, int val)  
   requires: nothing  
   effects: returns index i such that a[i] = val, or -1 if no such i
  + which has a weaker precondition, and a stronger postcondition
* Client guarantees that one occurs so the -1 will never get returned

# Learning Unit 2 – Dispatching (MLO 3) [~2 hours]

## Dispatching

Object[] x = new Object[2];  
x[0] = new String(“abc”);  
x[1] = new Integer(1);  
for(int i=0; i<x.length;i++)  
System.out.println(x[i].toString());

* Compiler does not complain
* Which toString() method is called? Object.toString(), String.toString() or Integer.toString()?

## Liskov 7.8, 7.9, 7.10

public class Counter{ // Liskov 7.8  
public Counter() //EFF: Makes this contain 0  
public int get()  
//EFF: Returns the value of this  
public void incr() //MOD: this //EFF: Increments value of this  
}  
public class Counter2 extends Counter { // Liskov 7.9  
public Counter2() //EFF: Makes this contain 0  
public void incr() // MOD: this //EFF: double this  
}  
public class Counter3 extends Counter { // Liskov 7.10  
public Counter3(int n) //EFF: Makes this contain n  
public void incr(int n) // MOD: this //EFF: if n>0 add n to this  
}  
public class Counter{ // Liskov 7.8  
public Counter() //EFF: Makes this contain 0  
public int get()  
//EFF: Returns the value of this  
public void incr() //MOD: this //EFF: Increments value of this  
}  
public class Counter2 extends Counter { // Liskov 7.9  
public Counter2() //EFF: Makes this contain 0  
public void incr() // MOD: this //EFF: double this  
}

* How many methods in Counter? 2
* How many methods in Counter2? 2
* Methods rule analysis involves what?
  + get() was inherited no need. Analyzing Counter2.incr():
  + precond is true for subtype and supertype
  + postcond: does “double this” always "increments value of this"? No (case of 0 or -1)

public class Counter{ // Liskov 7.8  
public Counter() //EFF: Makes this contain 0  
public int get()  
//EFF: Returns the value of this  
public void incr() //MOD: this //EFF: Increments value of this  
}  
public class Counter3 extends Counter { // Liskov 7.10  
public Counter3(int n) //EFF: Makes this contain n  
public void incr(int n) // MOD: this //EFF: if n>0 add n to this  
}

* How many methods in Counter3? 3
* Client cares about “get()” and “incr()” only. It cannot even see "incr(int)"
* Methods rule analysis involves what?
  + get() and incr() were inherited no need to analyze
  + Counter3.incr(int) cannot be seen no need to analyze

### Summary

* Signature rule: Careful with over-load vs. ride
  + Counter2 ok? yes
  + Counter3 ok? yes
* Methods rule:
  + Precondition rule:
    - Counter 2 ok? yes
    - Counter 3 ok? yes
* Postcondition rule:
  + Counter 2 ok? no
  + Counter 3 ok? yes

## MaxIntSet Example (Fig 7.5)

public class MaxIntSet extends IntSet {  
private int biggest; // biggest element of set if not empty  
public MaxIntSet() {super (); } //Why call super() ???  
public void insert (int x) {  
if (size() == 0 || x > biggest) biggest = x;  
super.insert(x); }  
public int max () throws EmptyException {  
if (size() == 0) throw new EmptyException (“MaxIS.max”);  
return biggest; }

### MaxIntSet.remove()

public void remove (int x) {  
super.remove(x);  
if (size()==0 || x <biggest) return;  
Iterator g = elements(); // find the new biggest  
biggest = ((Integer) g.next()).intValue();  
while (g.hasNext() {  
int z = ((Integer) g.next()).intValue();  
if (z>biggest) biggest = z;  
}

* Need to call supertype’s remove functionality. (private rep!)
* Must maintain subtype’s rep invariant

### MaxIntSet Abstract State

// Overview: MaxIntSet is a subtype of IntSet with an additional // method, max, to determine the maximum element of the set

* Two possible abstract states:
  + {x1, x2, … xN} - same as IntSet
  + <biggest, {x1, x2, … xN}> - visible abstract state

Which one to choose?

* Second may seem more natural

### MaxIntSet.repOk()

public boolean repOk() {  
if (!super.repOk()) return false; // all ints, no duplicates  
if (size() == 0) return true;  
boolean found = false;  
Iterator g = elements();  
// biggest is actually the max  
while(g.hasNext()) {  
int z = ((Integer)g.next()).intValue();  
if (z>biggest) return false;  
if (z==biggest) found = true;  
return found;  
}

### repOK() and Dynamic Dispatching

public class IntSet {  
public void insert(int x) {...; repOk();}  
public void remove(int x) {...; repOk();} // where to?  
public boolean repOk() {...}  
}  
public class MaxIntSet extends IntSet {  
public void insert(int x) {...; super.insert(x); repOk();}  
public void remove(int x) {super.remove(x); ...; repOk();}  
public boolean repOk() {super.repOk(); ...;}  
}  
MaxIntSet s = {3, 5}; s.remove(5);

* Which repOK() is being called?
  + Depends on the live object!!
* What does the default constructor in MaxIntSet do?
* What do the "…" do?
* How does the call work out?
* What is the abstract state of a MaxIntSet? There are two options. What are they?
* an call repOK() from within a JUnit test…after the assertion
* Why not call repOK() at the end of the methods?
  + You can—but watch out for the dynamic dispatching behavior
  + Due to dynamic dispatching, a repOK() in IntSet will call MaxIntSet.repOK(), because "this" is of type MaxInteSet
  + This might make repOK() return false

## Instructor Screencast: TITLE

# In Class 1 (MLO 1, 2, 3) [.5 hours]

class A:  
 public void reduce (Reducer x)  
 // Effects: if x is null throw NPE  
 // else if x is not appropriate for this throw IAE  
 // else reduce this by x  
  
 class B:  
 public void reduce (Reducer x)  
 // Requires: x is not null  
 // Effects: if x is not appropriate for this throw IAE  
 // else reduce this by x  
  
 class C:  
 public void reduce (Reducer x)  
 // Effects: if x is null return (normally) with no change to this  
 // else if x is not appropriate for this throw IAE  
 // else reduce this by x

Analyze the "methods rule" for reduce() in each of these cases: Note: Some analysis may not be necessary. If so, indicate that.

B extends A.  
Precondition Part:  
Postcondition Part:  
-----------------------------------  
C extends A.  
Precondition Part:  
Postcondition Part:  
-----------------------------------  
A extends B.  
Precondition Part:  
Postcondition Part:  
-----------------------------------  
C extends B.  
Precondition Part:  
Postcondition Part:  
-----------------------------------  
A extends C.  
Precondition Part:  
Postcondition Part:  
-----------------------------------

## Sols:

B extends A. Precondition Part: not satisfied LSP because B has stronger pre than A Postcondition Part: not statisfied because A has stronger post

C extends A. Precondition Part: C's pre <= A's pre , satisfied Postcondition Part: C's post >= A's post,

* A's post is stronger than C's post because NPE is preferred: LSP is violated
* C's post is stronger than A's post because normal return is preferred: LSP is satisfied
* A and C's behaviors are not comparable : LSP is violated

A extends B. Precondition Part: A has no pre and therefore is weaker than B – satisfies LSP Postcondition Part:

* Since A is stronger because it handles null (throwing NPE), but B does not – satisfies LSP
* If taken into account the precond of B, which disallow x being null, then the posts of A and B are the same , satisfies LSP

C extends B.

* preconds: C is weaker than B - satisfies LSP
* postconds:
  + since C can handle null input, C is stronger than B
  + since for non-null cases as required by the pre of B, both C and B hae same

A extends C. Precondition Part: Postcondition Part:

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

Consider the following:

public class Counter{ // Liskov 7.8  
 public Counter() //EFF: Makes this contain 0  
 public int get() //EFF: Returns the value of this  
 public void incr() //MOD: this //EFF: makes this larger  
 }  
public class Counter2 extends Counter { // Liskov 7.9  
 public Counter2() //EFF: Makes this contain 0  
 public void incr() // MOD: this //EFF: double this  
 }  
public class Counter3 extends Counter { // Liskov 7.10  
 public Counter3(int n) //EFF: Makes this contain n  
 public void incr(int n) // MOD: this //EFF: if n>0 add n to this  
 }

1. Is there a constraint about negative/zero values for this? How do we know?
2. What methods are in the Counter2 API?
3. Is Counter2 a valid subtype of Counter?
4. What methods are in the Counter3 API?

# Assignment – (MLO 1, 2) [~2 hours]

## Purpose

Practicing Type Abstraction

## Instructions

Consider the following Market class.

class Market {  
 private Set<Item> wanted; // items for which prices are of interest  
 private Bag<Item, Money> offers; // offers to sell items at specific prices  
 // Note: Bag isn't a Java data type. Here, the bag entries are pairs.  
  
 public void offer (Item item, Money price)  
 // Requires: item is an element of wanted  
 // Effects: add (item, price) to offers  
  
 public Money buy(Item item)  
 // Requires: item is an element of the domain of offers  
 // Effects: choose and remove some (arbitrary) pair (item, price) from  
 // offers and return the chosen price  
 }

1. Suppose that offers are only accepted if they are lower than previous offers.

* class Low\_Bid\_Market extends Market {  
   public void offer (Item item, Money price)  
   // Requires: item is an element of wanted  
   // Effects: if (item, price) is not cheaper than any existing pair  
   // (item, existing\_price) in offers do nothing  
   // else add (item, price) to offers
* Is Low\_Bid\_Market a valid subtype of Market? Appeal to the methods rule to back up your answer.

1. Suppose that the buy() method always chooses the lowest price on an item.

* class Low\_Offer\_Market extends Market {  
   public Money buy(Item item)  
   // Requires: item is an element the domain of offers  
   // Effects: choose and remove pair (item, price) with the   
   // lowest price from offers and return the chosen price
* Is Low\_Offer\_Market a valid subtype of Market? Appeal to the methods rule to back up your answer.

### Grading Criteria

This is purely a "paper and pencil" exercise. No code is required. Write your answer so that it is easily understandable by someone with only a passing knowledge of Liskov's rules for subtypes.

## Deliverable

* Submit a .java file for your implementation.

## Due Date

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

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

## Purpose

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

## Instructions

class A {  
 public Iterator compose (Iterator itr)  
 // Requires: itr is not null  
 // Modifies: itr  
 // Effects: if this is not appropriate for itr throw IAE  
 // else return generator of itr composed with this  
class B {  
 public Iterator compose (Iterator itr)  
 // Modifies: itr  
 // Effects: if itr is null throw NPE   
 // else if this is not appropriate for itr throw IAE  
 // else return generator of itr composed with this  
class C {  
 public Iterator compose (Iterator itr)  
 // Modifies: itr  
 // Effects: if itr is null return iterator equal to this  
 // else if this is not appropriate for itr throw IAE  
 // else return generator of itr composed with this

Analyze the compose() method in each of these cases according to Liskov's Principle of Substitution. For each case, state if the precondition and the postcondition parts are satisfied or fail, and **justify**.

1. B extends A.
2. C extends A.
3. A extends B.
4. C extends B.
5. B extends C.

**Solution**

1. B extends A.

Precondition Part: Answer: Satisfied: weaker precondition in B Postcondition Part: Answer: Satisfied: Identical behavior where A defined, with additional behavior where A not defined. 2.C extends A. Precondition Part: Answer: Satisfied: weaker precondition in C Postcondition Part: Answer: Satisfied: Identical behavior where A defined, with additional behavior where A not defined.

1. A extends B.

Precondition Part: Answer: Not satisfied: stronger precondition in A Postcondition Part: Answer: No analysis required

1. C extends B.

Precondition Part: Answer: Satisfied: same preconditions Postcondition Part: Answer A: Not satisfied: Inconsistent behavior in case where itr is null Answer B: Satisfied, stronger post (assuming returning a an iterator equal to this)

1. B extends C.

Precondition Part: Answer: Satisfied: same preconditions Postcondition Part: Answer: Not satisfied: Inconsistent behavior in case where itr is null

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.