Module 8 - Polymorphic Abstractions; Lambdas

# Overview and Objectives

## OVERVIEW

In this module we will learn about polymorphism, an important concept in Object Oriented programming. We will learn about the benefits of using polymorphism, in particular how they overcome issues when polymorphism was not used. We will also learn about using anonymous classes and especially lambda functions to write more precise and shorter code.

## 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:

1. explain and demonstrate the concept of Polymorphism and use Polymorphic Data Abstraction
2. understand and properly apply the different ways of checking equalities
3. distinguish and proper use of lambda/anonymous functions

# 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 8: Polymorphic Abstractions
* Joshua Bloch. Effective Java. Third Edition. Addison-Wesley Professional, 2017, ISBN 978-0-13-468599-1.
  + Chapter 7 - Item 42

# Learning Unit 1 – Motivation (MLO 1) [~0.5 hour]

* Previously we define collections of ints or doubles. This is rather limiting because they are restricted to specific types (ints or string).
* It would be better to define the collection type just once yet have it work for all types of elements
* **Polymorphic abstractions** is the solution for this. These abstractions are called polymorphic because they work for many types, e.g., Vector is polymorphic to its element.
  + Polymorphism generalizes abstractions so that they work for many types. It allows us to avoid having to redefine abstractions when we want to use them for more types; instead, a single abstraction becomes much more widely useful.
  + A procedure or iterator can be polymorphic with respect to the types of one or more arguments. A data abstraction can be polymorphic with respect to the types of elements its objects contain.

# TODO Learning Unit 2 – (MLO 1, 2) [~2.5 hour]

## Polymorphic Data Abstraction

* Fig. 8.1 (Liskov 8) provides a specification of a Set abstraction.
  + Set objects contain *heterogeneous* collections of elements (similar to Vector).
  + The specification of Set is similar to IntSet except that Set's methods (such as insert and isIn) take Objects as arguments or return them as results.
  + Because objects can be compared in various ways (using either == or equals), the overview states what equality test is being used.

## Equality Revisited

* Several ways to check if two objects are equal
  + **Object identity** [A==B] (same object)
  + **Object state** [A.counter = B.counter] (similar objects)
  + **Object property** [A.area() = B.area()] (practically same)
* Some trouble when storing mutable types in collections
  + assume a collection that does not allow publication (e.g., Set)
  + Aim: to store mutable types Vector
  + void insert (Object x) {  
     for all elements in collection{  
     if (element[i].equals(x)) return; // no duplicates  
     }   
     collection.addElement(x);
    - so far looks fine
  + Now let's try this client code (using the Set code in Fig. 8.1 of Liskov 8 which uses Vector)
    - equals for Vector returns true if 2 vectors have the same state

Set s = new HashSet(); //   
Vector x = new Vector(); //   
Vector y = new Vector(); //   
s.insert(x); //   
s.insert(y); // y is not added to s because   
s.contains(y) // true  
x.add(new Integer(3)) ;  
s.isIn(y); // false

* The solution to this problem is using immutable object
  + Consider Fig. 8.3 where we declare an immutable Container with
  + public boolean equals (Object x) {  
     if (! x instanceOf Container) return false;  
     return (el == ((Container) x.el)); }
  + then this client code works fine

Set s = new Set( );  
Vector x = new Vector( );  
Vector y = new Vector( );  
s.insert(new Container(x));  
s.insert(new Container(y));  
x.add(new Integer(3));  
if (s.isIn(new Container(y))) // will get here

## Equality of Mutable Types

* If two distinct List objects contain the same sequence of elements, then equals() reports that they are equal.
  + Suppose we make a List , and then drop it into a Set:
  + List<String> list = new ArrayList<>();  
    list.add("a");  
      
    Set<List<String>> set = new HashSet<List<String>>();  
    set.add(list);
  + We can check that the set contains the list we put in it, and it does:
  + set.contains(list) // true
  + But now we mutate the list: s
  + list.add("goodbye");
  + And it no longer appears in the set!
  + set.contains(list) // false!
* Using the same example, when we iterate over set, we find the list, but contains() says it’s not there!
* for (List<String> l : set) {   
   set.contains(l) // false!   
  }
  + If the set’s own iterator and its own contains() method disagree about whether an element is in the set, then the set clearly is broken.
  + What’s going on?
    - In Collection classes like List, mutations affect the result of equals() and hashCode(). When the list is first put into the HashSet, it is stored in the hash bucket corresponding to its hashCode() result at that time. When the list is subsequently mutated, its hashCode() changes, but HashSet doesn’t realize it should be moved to a different bucket. So it can never be found again.
    - public int hashCode() {   
       int hashCode = 1;  
       for (E e : this) hashCode = 31\*hashCode + (e==null ? 0 : e.hashCode());  
       return hashCode;  
       }
  + When equals() and hashCode() can be affected by mutation problem!
* The specification of java.util.Set states:
  + Note: Great care must be exercised if mutable objects are used as set elements. The behavior of a set is not specified if the value of an object is changed in a manner that affects equals comparisons while the object is an element in the set.
* Solution:
  + For mutable types, equals() should implement referential equality, or use containers (see Liskov)
  + For immutable types, equals() should compare contents, i.e., override equals() and hashCode()

## Overriding equals

* Object class equals is ‘==‘ check
* Overriding equals means providing a check other than object identity.
* Usually it provides object state check
* Overriding equals in a mutable class
* A.equals(B) is true/false at different times
* Immutable classes don’t suffer from this problem

## Autoboxing and Equality

* Autoboxing/Unboxing is meant to let you use primitive types interchangeably with wrapper types. However:
* Integer x = new Integer(3);  
  Integer y = new Integer(3);  
  x.equals(y) // true // makes sense  
  x == y  false // expected
* But for primitive types like int:
* (int)x == (int)y // returns true
* So primitive and wrapper types cannot always be used interchangeably. Consider:
* Map<String, Integer> a = new HashMap(), b = new HashMap();  
  a.put("c", 130);   
  b.put("c", 130);  
  a.get("c") == b.get("c") // false  
  (int)a.get("c") == (int)b.get("c") // true
* More examples:
* Integer x = new Integer(300);  
  Integer y = new Integer(300);  
  x.equals(y) // true  
  x == y // false   
    
  Integer x = 300;  
  Integer y = 300;  
  x.equals(y) // true   
  x == y // false   
    
  Integer x = 3;  
  Integer y = 3;  
  x.equals(y) // true   
  x == y // true
* /\*\*  
   \* Returns an {@code Integer} instance representing the specified  
   \* {@code int} value. If a new {@code Integer} instance is not  
   \* required, this method should generally be used in preference to  
   \* the constructor {@link #Integer(int)}, as this method is likely  
   \* to yield significantly better space and time performance by  
   \* caching frequently requested values.  
   \*  
   \* This method will always cache values in the range -128 to 127,  
   \* inclusive, and may cache other values outside of this range.  
   \*  
   \* @param i an {@code int} value.  
   \* @return an {@code Integer} instance representing {@code i}.  
   \* @since 1.5  
   \*/  
  public static Integer valueOf(int i) {  
   if (i >= IntegerCache.low && i <= IntegerCache.high)  
   return IntegerCache.cache[i + (-IntegerCache.low)];  
   return new Integer(i);  
  }

## Wrapper Classes

* Warpper Class (1)
  + Why is sum1() so much slower than sum2()?
    - sum1
    - public static long sum1() {   
       Long sum = 0L;  
       for (long i = 0; i <= Integer.MAX\_VALUE; i++) {  
       sum = sum + i;  
       }  
       return sum;  
      }  
      // ~7 seconds to run
    - sum2
    - public static long sum2() {   
       long sum = 0L;  
       for (long i = 0; i <= Integer.MAX\_VALUE; i++) {  
       sum = sum + i;  
       }  
       return sum;  
      }  
      // ~1 second to run
  + bytecode level…
    - sum1
    - aload\_0  
      invokevirtual java/lang/Long/longValue()J // Unboxing  
      lload\_1  
      ladd  
      invokestatic java/lang/Long/valueOf(J)Ljava/lang/Long; // Autoboxing  
      astore\_0
    - sum2
    - lload\_0  
      lload\_2  
      ladd  
      lstore\_0
* Warpper Class (2)
  + == and != are applicable to references
  + <, >, <=, >= induce unboxing

public static int compare(Integer i, Integer j) {   
 return (i < j) ? -1 : (i == j ? 0 : 1);   
}  
compare(new Integer(32), new Integer(42)) // expecting -1, got -1  
compare(new Integer(52), new Integer(42)) // expecting 1, got 1  
compare(new Integer(42), new Integer(42)) // expecting 0, got 1

* Suppose that (i < j) is false, then (i==j) is evaluated, if i and j refer to distinct Integer instances that represent the same int value, this comparison will return false.
  + fix:
  + public static int compare(Integer iBoxed, Integer jBoxed) {  
    int i = iBoxed, j = jBoxed;   
    return i < j ? -1 : (i == j ? 0 : 1);  
    }  
    // Should have used int instead of Integer in the first place
* Warpper Class (3)
  + When should we use Warpper Class?
    - To populate collections and hashmaps
    - When using generics
    - When using reflection (method invocation)
    - Watch out of Caching in general

## Polymorphism

* Generalize abstractions
  + They should work for many types
  + E.g.: IntSet could be generalized to Set
    - Not just store integers, but other data types
  + Saves us from creating new data abstractions for each data type (like PolySet, FloatSet, etc.)
  + Compare IntSet with HashSet, TreeSet
* Polymorphism is expressed through Type Hierarchy
* A polymorphic variable of type T can reference objects of type T or any of its subtypes
* A function might be polymorphic with respect to the types of its arguments
  + E.g., we could define a function to remove an element of an arbitrary type from a Vector
* Polymorphic procedures
  + Procedures can be polymorphic with respect to types of arguments
  + E.g.: Intset.insert(int x) becomes Set.Insert(Object x) or overloaded Set.Insert(…) with the specified list of types
* Polymorphic Data abstractions
  + Uniform methods for different types
    - “easy” polymorphism
  + element subtype (Comparable, Addable)
    - Pre planning.
    - Unique way for all subtypes
    - Comparable
      * Provides uniform way to compare elements
      * Abstracts from types
      * All types compared in a similar manner
    - Addable
      * Provides uniform way to add elements
      * Abstracts from types
      * All types added in a similar manner
  + related subtype (Comparator, Adder)
    - post planning, class designer did not provide it
    - create a related type for each object type
  + Both kinds use interfaces for generalization
* Related Subtype:
  + After classes have been designed
  + We want a collection to store and operate on any of such types
  + Some client code may already exist! We don’t want it to break.
  + So we create related subtype
  + Accompanies each type, supports desired operations
  + Example problem (figure 8.8):
    - We want to sum up all the elements in a set. SumSet class must maintain a running sum of all Integers, Floats or Poly’s stored.
    - We store one type of object at a time
    - SumSet a stores only Polys
    - SumSet b stores only Integers
    - public class SumSet {  
       private Vector els;  
       private Object s;  
       private Adder a;  
       public SumSet(Adder p) throws NPE {  
       els = new Vector();  
       a = p;  
       s = p.zero();  
       }  
       public void insert(Object x) throws NPE, CCE {  
       // M: this // E: if x is null throw NPE; if x cannot be added to this  
       // throw CCE; else adds x to this and adjusts the sum  
       Object z = a.add(s, x);  
       if (!els.contains(x)) {  
       els.add(x);  
       s = z;  
       }  
       public Object sum() { //E: return sum of elements in this  
       return s;  
       }  
       }

## Additional Methods

* Suppose we want to define an OrderedList type. We need a way to *order* the element.
  + We can achieve this using the Comparable interface, where arguments and results are now Comparable, e.g., In Fig. 8.5
  + boolean isIn (Comparable el)  
    // effects: if el is in this returns true else false
* Comparable:
  + comparable object is capable of comparing itself with another object (override compareTo)
    - e.g., compare age of a Person
    - what if in addition, we also want to compare the name of a Person? Not possible, already implement compareTo
  + requires *pre-planning*
* Comparator
  + *post-planning*
* Example: Comparable vs Comparator
* //comparable  
  class Person implements Comparable{  
   int age;  
   String name;  
   int years\_of\_experiences;  
    
   public int compareTo(Person p){//restricted to age  
   age.compare(p.age);  
   }  
  }  
    
    
  //comparator  
  class NamePerson implements Comparator{  
   public int compare (Person p1, Person p2){  
   //can do by name  
   }  
  }  
  class ExperiencePerson implements Comparator{  
   public int compare (Person p1, Person p2){  
   //can do by experience  
   }  
  }  
    
    
  Coolections.sort(collection\_of\_person, new NamePerson())
* Example: Set vs. IntSet
* Set stores Object types, IntSet stores ints
  + IntSet is homogeneous: the compiler guarantees that only ints could be inserted in IntSet
  + No such guarantee is provided for Set. It could contain a mix of types
  + The use of Generics (“Parametric Polymorphism”) will enable the compiler to disallow different types to populate the Set
  + What if we need an ordered Set?
    - This can be achieved by defining a supertype, all of whose subtypes have a comparison method
    - java.util.Comparable
* Example: Ordered List code (fig 8.5)

public class OL {  
 private boolean empty;  
 private OL left, right;  
 private Comparable val;  
 public void addEl(Comparable el) throws NPE, DE, CCE  
 // M: this // E: if el is null throw NPE else if el is in this throw DE else if el is incomparable to elements in this throw CCE else add el to this  
 if (el == null) throw new NPE(...)  
 if (empty) {  
 left = new OL();  
 right = new OL();  
 val = el;  
 empty = false;  
 return;  
 }  
 int n = el.compareTo(val);  
 if (n == 0) throw new DE(...);  
 if (n < 0) left.addEl(el);  
 else right.addEl(el);  
}

* Stores elements which implement Comparable interface
* Bug in addEl() (first line)
* “if (val == null)” should be “if (el == null)”
* Specs: order of exceptions!
* Very similar to TreeSet
* What is the abstract state?

# Learning Unit 2 – Lambdas (MLO 1, 2) [~1 hour]

* Block 7 Item 42: Prefer lambdas to anonymous classes
  + Sorting a list of string using anonymous class
  + // Anonymous class instance as a function object - obsolete!  
    Collections.sort(words, new Comparator<String>() {  
     public int compare(String s1, String s2) {  
     return Integer.compare(s1.length(), s2.length());  
     }  
    });
  + Works ok, but too verbose. In Java 8, we can use lambda

// Lambda expression as function object (replaces anonymous class)  
Collections.sort(words,  
 (s1, s2) -> Integer.compare(s1.length(), s2.length()));

# In-class Exercise (MLO 1, 2, 3) [.5 hours]

public class Person {  
  
 public enum Sex {  
 MALE, FEMALE  
 }  
  
 String name;  
 Sex gender;  
 String emailAddress;  
  
 public int getAge() {  
 // ...  
 }  
  
 public void printPerson() {  
 // ...  
 }  
}

Approach 1: Create Methods That Search for Members That Match One Characteristic.

One simplistic approach is to create several methods; each method searches for members that match one characteristic, such as gender or age. **Create a method that prints members that are older than a specified age**.

Limitation: This approach can potentially make your application brittle, which is the likelihood of an application not working because of the introduction of updates (such as newer data types). Suppose that you upgrade your application and change the structure of the Person class such that it contains different member variables; perhaps the class records and measures ages with a different data type or algorithm. You would have to rewrite a lot of your API to accommodate this change. In addition, this approach is unnecessarily restrictive; what if you wanted to print members younger than a certain age, for example?

Approach 2: Create More Generalized Search Methods.

Create a method is more generic than the one in the previous approach. It prints members within a specified range of ages.

Limitation: What if you want to print members of a specified sex, or a combination of a specified gender and age range? What if you decide to change the Person class and add other attributes such as relationship status or geographical location? Although this method is more generic, trying to create a separate method for each possible search query can still lead to brittle code. You can instead separate the code that specifies the criteria for which you want to search in a different class.

Approach 3: Specify Search Criteria Code in a Local Class

Instead of writing filtering functions, use a new interface and class for each search you plan. Use the following filtering criteria for example: filters members that are eligible for Selective Service in the United States: those who are male and between the ages of 18 and 25:

Limtation: Although this approach is less brittle—you don't have to rewrite methods if you change the structure of the Person—you still have additional code: a new interface and a local class for each search you plan to perform in your application. Because one of the class implements an interface, you can use an anonymous class instead of a local class and bypass the need to declare a new class for each search.

Approach 4: Specify Search Criteria Code in an Anonymous Class

Use an anonymous class to address the issue with Approach 3.

Limtation: This approach reduces the amount of code required because you don't have to create a new class for each search that you want to perform. However, the syntax of anonymous classes is bulky considering that the CheckPerson interface contains only one method. In this case, you can use a lambda expression instead of an anonymous class, as described in the next section.

Approach 5: Specify Search Criteria Code with a Lambda Expression

Use lambda expression to address the limitation the previous approach.

# In-class Exercise (MLO 1, 2, 3) [.5 hours]

# Assignment – (MLO 1, 2) [~2 hours]

## Purpose

Practing lambdas and anonymous functions

## Instructions

public class Person {  
  
 public enum Sex {  
 MALE, FEMALE  
 }  
  
 String name;  
 Sex gender;  
 String emailAddress;  
  
 public int getAge() {  
 // ...  
 }  
  
 public void printPerson() {  
 // ...  
 }  
}

Approach 1: Create Methods That Search for Members That Match One Characteristic.

One simplistic approach is to create several methods; each method searches for members that match one characteristic, such as gender or age. **Create a method that prints members that are older than a specified age**.

Limitation: This approach can potentially make your application brittle, which is the likelihood of an application not working because of the introduction of updates (such as newer data types). Suppose that you upgrade your application and change the structure of the Person class such that it contains different member variables; perhaps the class records and measures ages with a different data type or algorithm. You would have to rewrite a lot of your API to accommodate this change. In addition, this approach is unnecessarily restrictive; what if you wanted to print members younger than a certain age, for example?

Approach 2: Create More Generalized Search Methods.

Create a method is more generic than the one in the previous approach. It prints members within a specified range of ages.

Limitation: What if you want to print members of a specified sex, or a combination of a specified gender and age range? What if you decide to change the Person class and add other attributes such as relationship status or geographical location? Although this method is more generic, trying to create a separate method for each possible search query can still lead to brittle code. You can instead separate the code that specifies the criteria for which you want to search in a different class.

Approach 3: Specify Search Criteria Code in a Local Class

Instead of writing filtering functions, use a new interface and class for each search you plan. Use the following filtering criteria for example: filters members that are eligible for Selective Service in the United States: those who are male and between the ages of 18 and 25:

Limtation: Although this approach is less brittle—you don't have to rewrite methods if you change the structure of the Person—you still have additional code: a new interface and a local class for each search you plan to perform in your application. Because one of the class implements an interface, you can use an anonymous class instead of a local class and bypass the need to declare a new class for each search.

Approach 4: Specify Search Criteria Code in an Anonymous Class

Use an anonymous class to address the issue with Approach 3.

Limtation: This approach reduces the amount of code required because you don't have to create a new class for each search that you want to perform. However, the syntax of anonymous classes is bulky considering that the CheckPerson interface contains only one method. In this case, you can use a lambda expression instead of an anonymous class, as described in the next section.

Approach 5: Specify Search Criteria Code with a Lambda Expression

Use lambda expression to address the limitation the previous approach.

## Deliverable

* Submit a .java file for your implementation.

## Due Date

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

# TODO 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

Set<String> t = // See questions below  
  
 t.add("antelope");  
 t.add("dog");  
 t.add("cat");  
  
// t.toString() is ???

1. Suppose t is instantiated as Set<String> t = new TreeSet<String>();. At the end of the computation, what is t.toString()?

**Answer**: [antelope, cat, dog]

1. Suppose t is instantiated as Set<String> t = new TreeSet<String>((x,y) -> x.length() - y.length());. At the end of the computation, what is t.toString()?

**Answer**: [dog, antelope]

1. Suppose t is instantiated as Set<String> t = new TreeSet<String>((x,y) -> y.compareTo(x));. At the end of the computation, what is t.toString()?

**Answer**: [dog, cat, antelope]

1. Which of the above Comparator implementations is problematic? and why?

**Answer**: 2, compare(a,b) is not consistent with a.equals(b).

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.