Module 6: Iteration Abstraction; Method Guidelines

# Overview and Objectives

## In this module we will learn about iterator and its concept and usages in traversing and accessing collections. We will also learn various techniques to effectively create and check methods

## 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 and use key concepts of iteration abstraction (CLO 1, CLO 4)
2. Construct and use methods (e.g., overloading, signatures) safely and appropriately (CLO 1, CLO 2)

# 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 6
* Joshua Bloch. Effective Java. Third Edition. Addison-Wesley Professional, 2017, ISBN 978-0-13-468599-1.
  + Chapter 8: Methods

# Learning Unit 1 – Interation Abstraction (MLO 1) [~3 hours]

* We have lots of common collections
  + List (Interface)
    - AbstractList
      * ArrayList, Linked List, Vector
  + Set (Interface)
    - AbstractSet
      * HashSet, TreeSet
  + Map (Interface)
    - AbstractMap
      * HashMap, TreeMap
  + Arrays
  + Stack -subtype of Vector

### How to best visit each element of a collection?

* iteration abstraction: implemented as an **iterator**
* Iterators permit users to iterate over arbitrary types of data
* Example Goal: **for all elements of the set, do action**
* As a client, compute the sum of elements in IntSet:
* public static int setSum (IntSet s) throws NullPointerException  
   // EFFECTS: If s is null throws NullPointerException else  
   // returns the sum of the elements of s.
* LiskovSet.java: no easy way! Must use choose()/remove()!!
* Current IntSet abstraction does not allow client to efficiently iterate over an IntSet
* public static int setSum (IntSet s) throws NullPointerException {  
   int[ ] a = new int[s.size( )];  
   int sum = 0;  
   for (int i = 0; i < a.length; i++) {  
   a[i] = s.choose( );  
   sum = sum + a[i];  
   s.remove(a[i]);   
   }  
   // the above destroys the original set!!  
    
   // restore elements of s  
   for (int i = 0; i < a.length; i++) s.insert(a[i]);  
    
   return sum;  
  }

### Possible fixes…

* If setSum() were an IntSet operation, we could implement it efficiently by manipulating the rep of IntSet. However:
  + setSum() does not naturally fit as an IntSet operation
  + IntSet might we untouchable
* Must be a way to implement such procedures efficiently outside the type without destroying the collection
* One possibility is to provide a members() method:
* public int[ ] members ( ) // new method  
   // EFFECTS: Returns an array containing the elements of this,  
   // each exactly once, in some arbitrary order

Then

public static int setSum (IntSet s) {  
 int[ ] a = s.members( );  
 int sum = 0;  
 for (int i = 0; i < a.length; i++) sum = sum + a[i];  
 return sum;  
}

### Possible fixes

* members() makes it easier to use IntSets, however it is inefficient:
  + if the IntSet is large, space requirements = IntSet itself + the array
* Why not have an operation that returns the rep vector?
  + bad since it destroys data abstraction by exposing the rep
* What is needed is a mechanism that is convenient + efficient + preserves abstraction:
  + An iterator causes the items we want to iterate over to be produced incrementally
  + *for each item i produced by iterator A, perform some action on i*
* Each iteration of the loop produces a new item, which is then acted on by the body of the loop

### Why use iterators? (summary)

* Separation of concerns:
  + The iterator is responsible for producing the items (only)
  + The code containing the loop defines the action to be performed on them
* Advantages:
  + The iterator can be used in different modules that perform different actions on the items, and it can be implemented in different ways without affecting these modules
  + Since the iterator causes items to be produced one at a time, it avoids the space and time problems discussed earlier
    - No need for a large data structure to contain the items
    - If performing a search loop, the iterator can be stopped as soon as the item of interest is found
* Why not use while/for loops?
  + While/for are useful in conjunction with arrays, which are indexed, but does not mesh well with non-indexed collections like IntSet
  + Iterators provide convenient iteration for index + non-indexed collections

# Iterator Interface

public interface Iterator<E> {  
 boolean hasNext();  
 E next();  
 default void remove() {   
 throw new UnsupportedOperationException("remove"); }  
 default void forEachRemaining(Consumer<? super E> action) {  
 Objects.requireNonNull(action);  
 while (hasNext()) action.accept(next());  
 }  
 }  
 class CollectionExample<E> implements Iterable<E> {  
 public Iterator<E> iterator() {  
 return new IteratorExample<>(); // Generator instance  
 }  
 }  
 public class IteratorExample<E> implements Iterator<E> { // Generator  
 IteratorExample() {}   
 public boolean hasNext() { return true; }  
 public E next() { return null; }  
 public void remove() {} // has default implementation   
 void forEachRemaining(Consumer<? super E> action) {} // has default implementation   
 }  
  
 class Main {  
 public static void main(String[] args)  
 {  
 // Create Linked List  
 LList<String> myList = new LList<>();  
  
 // Add Elements  
 myList.add("abc");  
 myList.add("mno");  
 myList.add("pqr");  
 myList.add("xyz");  
  
 // Iterate through the list using For Each Loop  
 for (String string : myList)  
 System.out.println(string);  
 }  
 }  
  
  
  
 // Note  
 LList<String> lst = ...;  
 for (String str : lst) {  
 System.out.println(str);  
 }  
 is rewritten by the compiler into something like this:   
 LList<String> lst = ...;  
 Iterator iter = lst.iterator();  
 while (iter.hasNext()) {  
 String str = iter.next();  
 System.out.println(str);  
 }  
  
  
 class LList<T> implements Iterable<T> {  
 Node<T> head, tail;  
  
 public void add(T data) { …}  
 public Iterator<T> iterator() {  
 return new ListIterator<T>(this);  
 }  
 ...  
 }  
  
class ListIterator<T> implements Iterator<T> {  
 Node<T> current;  
  
 // initialize pointer to head of the list for iteration  
 public ListIterator(LList<T> list) {  
 current = list.getHead();  
 }  
  
 // returns false if next element does not exist  
 public boolean hasNext() {  
 return current != null;  
 }  
  
 // return current data and update pointer  
 public T next() {  
 T data = current.getData();  
 current = current.getNext();  
 return data;  
 }  
  
 // implement if needed  
 public void remove() {  
 throw new UnsupportedOperationException();  
 }  
}

### Specification

public boolean hasNext() ;  
// Effects: Returns true if there are more elements to yield else returns false  
  
public Object next();  
// Modifies: this  
// Effects: If there are more results to yield, returns the next result and modifies the state of this to record the yield. Otherwise, throws NoSuchElementException

* NoSuchElementException: is an unchecked exception because of the expectation that most will call hasNext() prior to calling next()

### Examples: Poly and IntSet

class Poly {  
public Iterator terms() // should be called iterator()  
 // Effects: Returns a generator that will produce exponents  
 // of nonzero terms of this (as Integers) up to the degree,  
 // in order of increasing exponent  
}  
  
class IntSet {  
public Iterator elements() // should be called iterator()  
 // Effects: Returns a generator that will produce all the elements  
 // of this (as Integers) each exactly once, in arbitrary order  
 // Requires: this must not be modified while the  
 // generator is in use  
}

1. Example: Poly

* Poly p ... // p = 2 + 3 x2+ 4 x5  
  Iterator itr = p.iterator(); // called terms() by Liskov  
   // itr = [0,2,5]  
  itr.hasNext() // return true, itr = [0,2,5]  
  itr.next() // return 0, itr = [2,5]  
  itr.next() // return 2, itr = [5]  
  itr.hasNext() // return true, itr = [5]  
  itr.next() // return 5, itr = []  
  itr.hasNext() // return false, itr = []  
  itr.next() // return NSEE, itr = []  
   // NSEE = NoSuchElementException

### Abstraction Function

* All generators have the same **abstract state** = a sequence of the items that remain to be generated. The **AF** thus needs to map the rep to this sequence
* Very close to a Stack
  + top = next() element
  + Why not List but Stack? Once visited, element cannot be revisited
* remove() adds complexity
* What if multiple iterators are working simultaneously?
* **Only mutable data types are problematic**

### Af(c) for Poly Iterator

Poly p ... // p = 2 + 3 x2+ 4 x5  
  
AF(itr.hasNext()) = [0,2,5] //true  
AF(itr.next()) = [2,5] //0  
AF(itr.next()) = [5] //2  
AF(itr.hasNext()) = [5] //true  
AF(itr.next()) = [] //5  
AF(itr.hasNext()) = [] //false  
AF(itr.next()) = [] //NSEE

### Implementation (Fig 6.8)

public class Poly{  
 // Rep …  
 public Iterator terms() {return new PolyGen(this);} // iterator()  
 // inner class  
 private static class PolyGen implements Iterator {  
 private Poly p; // the Poly being iterated  
 private int n; // the next term to consider  
 PolyGen (Poly it){  
 //Requires: it !=null  
 p = it;  
 if(p.trms[0] == 0) n=1; else n= 0;  
 }   
  
 public boolean hasNext() {return n<= p.deg;}  
  
 public Object next () throws NSEE{  
 for(int e = n; e <= p.deg; e++) {  
 if (p.trms[e] != 0) {  
 n= e+1;  
 return new Integer(e);  
 }  
 }  
 throw new NSEE(“Poly.terms”);  
 } // end PolyGen  
 }

### Inner Class

* private class
* visibility only inside the class where defined
* no outside code can see/instantiate it
* if it has public methods && an instance available, outside code can call it

### State for iterator

* How to figure out the state?
* Same way as AF(c) for Data Abstraction
  + Ask yourself: What do I need to send back to the client?
* Example Rep state: [2,0,3,0,0,4]
* What if PolyGen was immutable?

### Another example: PrimesGen

private static class PrimesGen implements Iterator{  
 private Vector ps; // primes yielded  
 private int p; // next candidate to try  
 PrimesGen () { p =2 ; ps = new Vector();} //constructor  
 public boolean hasNext() {return true;} // always true  
 public Object next() throws NSEE {  
 if (p==2) {p=3; return 2;}  
 for (int n=p; true; n = n+2){  
 … //Prime number generation  
 }  
 }  
}// end of PrimesGen

1. Abstract State for PrimesGen?
   * Iterator itr = num.allPrimes();
   * AF(c) = [2,3,5,7,11,13,17,19, …] -No end? Can we figure out the length of the tail?
     + What does hasNext() have to do in this case?
     + [2,3,5,7,9,…]

* Integer x = (Integer) itr.next(); [3,5,7,9,11,..] Integer y = (Integer) itr.next(); [5,7,9,11,13,17,…]

### Exercises

* What if there is an upper bound on the prime numbers?
* Suppose primes <100.
* What will AF(c) be like?
* What will hasNext() implementation do?
* How will implementation of PrimesGen change?

### Another Exercise

public Interface TwoWayIterator {  
 Object next ();  
 Object previous ();  
 boolean hasNext();  
 boolean hasPrevious();

* Suppose we want to go back AND forward
* How does the AF look like?
  + Still a stack?
  + What other state information is needed?
* How to implement this for Poly?

### What about supporting remove()?

* The contract for remove():
  + Removes from the underlying collection the last element returned by the iterator (optional operation). This method can be called only once per call to next. The behavior of an iterator is unspecified if the underlying collection is modified while the iteration is in progress in any way other than by calling this method.
* This is complex!
* What is the new abstract state?

### Iterable vs. Iterator

* Only one method required:
  + public Iterator<T> iterator();
* Allows very nice code:
* // Note: that Collection implements Iterable  
  // Side note: String does NOT implement Iterable  
  Set<String> mySet = new HashSet<String>();  
  // populate mySet with various Strings  
  for (String s : mySet) { // auto invocation of iterator(), next()  
   // do something with s  
  }

## Mutation and Iterators

### Iterating over arrays and lists

* An iterator is mutable, it traverses a collection and returns the elements one by one.
* Iterators are used under the covers in Java when you’re using "for (… : …)":
* List<String> lst = ...;  
  for (String str : lst) {  
   System.out.println(str);  
  }

is rewritten by the compiler into something like this:

List<String> lst = ...;  
Iterator iter = lst.iterator();  
while (iter.hasNext()) {  
 String str = iter.next();  
 System.out.println(str);  
}

An iterator has two methods:

* next() returns the next element in the collection. It is a mutator method, it advances the iterator so that the subsequent call to next() will return a different element
* hasNext() tests whether the iterator has reached the end of the collection.
* /\*\*  
  A MyIterator is a mutable object that iterates over the elements of an ArrayList<String>, from first to last. This is just an example to show how an iterator works. In practice, you should use the ArrayList's own iterator object, returned by its iterator() method.  
   \*/  
  public class MyIterator {  
   private final ArrayList<String> list;  
   private int index;  
    
   /\*\*  
   \* Make an iterator.  
   \* @param list list to iterate over  
   \*/  
   public MyIterator(ArrayList<String> list) {  
   this.list = list;  
   this.index = 0;  
   }  
    
   /\*\*  
   \* Test whether the iterator has more elements to return.  
   \* @return true if next() will return another element, false if all elements have been returned  
   \*/  
   public boolean hasNext() {  
   return index < list.size();  
   }  
    
   /\*\*  
   \* Get the next element of the list.  
   \* Requires: hasNext() returns true.  
   \* Modifies: this iterator to advance it to the element following the returned element.  
   \* @return next element of the list  
   \*/  
   public String next() {  
   final String element = list.get(index); // might throw IndexOutOfBoundsException  
   ++index; // Mutation  
   return element;  
   }  
  }
* Why do iterators exist?

Linked lists, maps, hash tables have different kinds of internal representations. The iterator concept allows a single uniform way to access them all.

### Mutation undermines an iterator

* Suppose we have a list of course subjects represented as strings, like ["6.005", "8.03", "9.00"] . We want a method dropCourse6 that will delete the Course 6 subjects from the list. The specification looks as follows:
* /\*\*  
   \* Drop all subjects that are from Course 6.   
   \* Modifies subjects list by removing subjects that start with "6."  
   \*   
   \* @param subjects list of subject numbers  
   \*/  
  public static void dropCourse6(ArrayList<String> subjects)
* The developer designs tests that cover the various parts of the input space:
* // Testing strategy:  
  // subjects.size: 0, 1, n  
  // contents: no 6.xx, one 6.xx, all 6.xx  
  // position: 6.xx at start, 6.xx in middle, 6.xx at end  
    
  // Test cases:  
  // [] => []  
  // ["8.03"] => ["8.03"]  
  // ["14.03", "9.00", "21L.005"] => ["14.03", "9.00", "21L.005"]  
  // ["2.001", "6.01", "18.03"] => ["2.001", "18.03"]  
  // ["6.045", "6.005", "6.813"] => []
* The tests passed on the implementation below except for: // dropCourse6(["6.045", "6.005", "6.813"]); expected [], actual ["6.005"]

public static void dropCourse6(ArrayList<String> subjects) {  
 MyIterator iter = new MyIterator(subjects);  
 while (iter.hasNext()) {  
 String subject = iter.next();  
 if (subject.startsWith("6.")) {  
 subjects.remove(subject);  
 }  
 }   
}

dropCourse6 failed! Why? Track the value of index …

Index = 0 ["6.045", "6.005", "6.813"]  
list.get(0), subject = “6.045”, remove “6.045” ["6.005", "6.813"]  
Index =1   
list.get(1), subject = “6.813”, remove “6.813” ["6.005"]  
Index = 2

* This isn’t a bug in MyIterator only, but also in the built-in iterator in ArrayList, and the “for (… : …)”.
* However, the iterator in ArrayList detects the problem and throws an exception (ConcurrentModificationException)
* for (String subject : subjects) {  
   if (subject.startsWith("6.")) {  
   subjects.remove(subject);  
   }  
  }
* One way to fix this problem is to use the remove() method of Iterator, so that the iterator adjusts its index appropriately:

Iterator iter = subjects.iterator();  
while (iter.hasNext()) {  
 String subject = iter.next();  
 if (subject.startsWith("6.")) {  
 iter.remove(); // should reduce index  
 }  
}  
  
//Index = 0 ["6.045", "6.005", "6.813"]  
//list.get(0), subject = “6.045”, remove “6.045” ["6.005", "6.813"]  
//Index =0   
//list.get(0), subject = “6.813”, remove “6.813” ["6.005"]  
//Index = 0

* However, this doesn’t fix the whole problem. What if there are other Iterators currently active over the same list? They won’t all be informed!

# Learning Unit 2 – Methods (Bloch 8) (MLO 2) [~2.5 hour]

## Item 49: Check Parameters for Validity

* Check and Document
  + Genuine Preconditions – use assert() facility
    - Often for nonpublic methods
    - Particularly important for parameters that are stored away for later use
    - Early detection!
* Postconditions – use exceptions and @throws tag
* Don’t throw the wrong exception
  + E.g. Client passes no index, but sees IOOBE
* Make sure necessary exceptions are thrown
  + E.g.Client expects NPE, but gets normal return instead
* Examples
* // Private helper function for a recursive sort  
  // Notice the use of assertions for genuine preconditions  
  private static void sort (long a[], int offset, int length){  
   assert a != null;  
   assert offset >= 0 && ofset <= a.length;  
   assert length >= 0 && length <= a.length – offset;  
   // Do the computation  
    
    
    
  // Big Integer example of checking for documented exception  
  @throws ArithmeticException if m is les than or equal to 0  
  public BigInteger mod (BigInteger) {  
   if (m.signum() <= 0) throw new ArithmeticException("Modulus <= 0: " + m);  
  }  
    
  }

## Item 50: Make Defensive Copies When Needed

* Java is a safe language
  + No buffer overflow, array overruns, wild pointers, memory corruption as in C, C++
* But you still need to insulate your class from client classes
* Best Approach
  + Assume Clients of your class will do their best to destroy your invariants
  + This is actually what happens in any type of security attack
* Surprisingly easy to unintentionally provide access to internal state
* Example
* // Broken “immutable” time period class  
   public final class Period {  
   private final Date start;  
   private final Date end;  
   public Period (Date start, Date end) {  
   if (start.compareTo(end) > 0) throw new IAE(…);  
   this.start = start; this.end = end; //oops – should make defensive copies  
   }  
   public Date start() { return start;} // oops again!   
   public Date end() { return end;} // oops yet again!  
   }  
   // Attack code  
   Date start = new Date();  
   Date end = new Date();  
   Period p = new Period(start, end);  
   end.setYear(78); // Attack 1: Modify internals of p via passed reference  
   p.end().setYear(78); // Attack 2: Modify internals of p via returned state  
    
  // Repaired constructor and getters – now Period really is immutable  
  public Period (Date start, Date end) {  
   // Note: clone() \*not\* used to make defensive copy  
   // Reason: Date class not final; hence return type may not be java.util.Date  
   this.start = new Date(start.getTime()); // Defensive copy  
   this.end = new Date(end.getTime()); // Defensive copy  
    
   // Defensive copies made \*before\* exception check to avoid TOCTOU attack  
   if (this.start.compareTo(end) > 0) throw new IAE(…);  
  }  
    
  // clone also safe here, but constructors or static factories better (Item 11)  
  public Date start() { return new Date(start.getTime()); }   
  public Date end() { return new Date(end.getTime());}   
  }
* Make defensive copies of all mutable data provided by client
* Make defensive copies of all mutable internal state returned to client
  + Includes all arrays of length >0
* Arguably, the lesson is that Immutable objects should be used where possible
  + No copying necessary, so you can’t forget!
* Caveat: Occasionally, you can’t afford the copy 
  + Heavy performance penalty?
  + Do you trust the client?
    - From a security perspective, trust is a bad thing
  + Some objects are explicitly handed off
    - Hence, it is not possible to protect the object

## Item 51: Design Method Signatures Carefully

* Choose methods names carefully
* Don’t go overboard providing convenience methods
* Avoid long parameter lists
  + May require multiple methods instead of one
  + View idiom helpful here
    - Example: Finding first or last element in a sublist
    - Standalone method requires 3 arguments
    - java.util.List interface provides a subList() view
    - indexOf(), lastIndexOf() can be applied to subList() result
    - Resulting API has very high “power to weight” ratio
* For parameter types, favor interfaces over classes
* public static boolean TreeSet<Color> filterPrimary(TreeSet<Color> s)  
  // vs.  
  public static boolean Set<Color> filterPrimary(Set<Color> s)
* Prefer two-element enum types to boolean parameters
* boolean Fahrenheit  
  // vs.   
  enum TemperatureScale { FAHRENHEIT, CELSIUS }
  + Think about adding Kelvin temperatures to these types

## Item 52: Use Overloading Judiciously

* Overriding
  + Normal OO Pattern
  + Dynamically determined by runtime system
* Overloading
  + Exceptional OO Pattern
    - Statically determined by compiler
* Avoid confusing uses of overloading, especially in APIs
* Overloading documentation in Java Language Specification is 33 pages long!
* Safe Policy: Don’t export two overloadings with the same number of parameters

// Broken! – What does this program print?  
// Problem: classify() is overloaded, not overridden  
public class CollectionClassifier {  
 public static String classify(Set<?> s) { return “Set”; }   
 public static String classify(List<?> l) { return “List”; }   
 public static String classify(Collection<?> c) { return “Collection”; }  
  
 public static void main(String[] args) {  
 Collection <?>[] collections = {  
 new HashSet<String>(),  
 new ArrayList<BigInteger>(),  
 new HashMap<String, String>().values()  
 };  
 for (Collection <?> c : collections)   
 System.out.println(classify(c));  
 }  
}

// Broken! – Autoboxing and overloading combine for total confusion!  
public class SetList {  
 public static void main(String[] args) {  
 Set <Integer> set = new TreeSet <Integer>();  
 List<Integer> list = new ArrayList<Integer>();  
 for (int i = -3; i < 3; i++) {   
 set.add(i); list.add(i);  
 }   
 for (int i = 0; i < 3; i++) {   
 set.remove(i); list.remove(i);  
 }  
 System.out.println(set + “ “ + list);  
 // Expect [-3, -2, -1] [-3, -2, -1]  
 // Actual [-3, -2, -1] [-2, 0, 2]   
 // Key: List interface overloads remove(E e) and remove(int i)  
 // Prior to generics (Java 1.5), this wasn’t a problem   
 }  
}

## Item 53: Use Varargs Judiciously

* Variable number of argument introduce in Java 1.5 to mimic printf() in C
* Generally, avoid
* But sometimes, it’s very useful

// Simple use of varargs  
static int sum(int ... args) {  
 int sum = 0;  
 for (int arg : args)  
 sum += arg;  
 return sum;  
}

// Example of WRONG way to use varargs  
// Runtime (instead of compile time) notice for 0 arguments  
// Ugly for loop  
static int min(int ... args) {  
 if (args.length == 0) throw new IAE (“Too Few Arguments”);  
 int min = args[0];  
 for (int i = 1; i < args.length; i++)  
 if (args[i] < min) min = args[i];  
 return min;  
}  
// Much better version   
static int min(int firstArg, int ... remainingArgs) {  
 int min = firstArg;  
 for (int arg : remainingArgs)   
 if (arg < min) min = arg;  
 return min;  
}

## Item 54: Return Empty Arrays or Collections, Not Nulls

// Common example  
private final List<Cheese> cheesesInStock = . . .;  
/\*\*  
 \*@return an array containing all of the cheeses in the shop,  
 \* or null if no cheeses are available for purchase  
 \*/  
public Cheese[] getCheeses() {  
 if (cheesesInStock.size() == 0) return null;  
 ...  
// Client code  
Cheese[] cheeses = shop.getCheeses();  
if (cheeses != null &&  
 Arrays.asList(cheeses).contains(Cheese.STILTON))  
 System.out.println(“Jolly good, just the thing.”);  
// vs.  
if (Arrays.asList(cheeses).contains(Cheese.STILTON))  
 System.out.println(“Jolly good, just the thing.”);

* Making client handle null as special case is undesirable
* Performance penalty of creating an empty array is almost always irrelevant (See Item 67)
* Null return idiom probably a holdover from C
  + C programs must separately return arrays and array lengths
* Model for Lists (which are preferable to Arrays)
* // The right way to return a copy of a collection  
  public List<Cheese> getCheeseList() {  
   if (cheesesInStock.isEmpty())   
   return Collections.emptyList(); // Always return same list  
   else  
   return new ArrayList<Cheese>(cheesesInStock);  
  }

## Item 55: Return optionals judiciously

* Java 8 added the a facility for optional returns
* Integrates with the Java stream mechanism
* Optionals are similar in spirit to checked exceptions in that they force the user of an API to confront the fact that there may be no return value.
* Don’t wrap Java containers in optionals
  + Return empty containers instead (Item 54)

// Returns maximum value in collection as an Optional<E>  
public static <E extends Comparable<E>>   
 Optional<E> max(Collection<E> c) {  
 if (c.isEmpty())   
 return Optional.empty();  
  
 E result = null;   
 for (E e: c)  
 if (result == null || e.compareTo(result) > 0)  
 result = Objects.requireNonNull(e);  
  
 return Optional.of(result);  
}  
// Using an optional to provide a chosen default value  
String lastWordInLexicon = max(words).orElse("No words...");

## Item 56: Write doc comments for all exposed API Elements

* Precede every exported class, interface, constructor, method, and field description with a doc comment
* Doc comment should describe CONTRACT between method and its client
* No two members or constructors should have the same summary description
* Every method should have
  + @param tag for each parameter
  + @return tag (unless return type is void)
  + @throws tag for each exception (both checked and unchecked)

// Good Example  
/\*\*  
 \* Returns the element at the specified position in this list  
 \*  
 \* <p>This method is <i>not</i> guaranteed to run in constant time. In some  
 \* implementations it may run in time proportional to the element position.  
 \*  
 \* @param index index of element to return; must be non-negative  
 \* and less than the size of this list  
 \* @return the element at the specified position in this list  
 \* @throws IndexOutOfBoundsException if the index is out of the range  
 \* ({@code index < 0 || index >= this.size()})  
 \*/  
 E get(int index)

## Instructor Screencast: TITLE

# TODO Module 3 Learning Unit 2 – (MLO 1, 2) [~2.5 hour]

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

Consider the Java Iterator<E> interface:

public boolean hasNext();  
public E next() throws NoSuchElementException  
 public void remove() throws IllegalStateException

1. What is the abstract state of an iterator without the remove() method?
2. Work through an example iterating over a list of strings: ["bat", "cat", "dog"]
3. What is the abstract state of an iterator with a previous() method?
4. What is the abstract state of an iterator with the remove() method?
5. Design an immutable version of the iterator.
   1. How is hasNext() handled?
   2. How is next() handled?
   3. How is remove() handled?
6. Exercise the immutable iterator with some sample client code.

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

Consider the example in Bloch's Item 50 (3rd Edition):

// Broken “immutable” time period class  
public final class Period { // Question 3  
 private final Date start;  
 private final Date end;  
  
 /\*\*  
 \* @param start the beginning of the period  
 \* @param end the end of the period; must not precede start  
 \* @throws IAE if start is after end  
 \* @throws NPE if start or end null  
 \*/  
  
 public Period (Date start, Date end) {  
 if (start.compareTo(end) > 0) throw new IAE();  
 this.start = start; this.end = end; // Question 1  
 }  
 public Date start() { return start;} // Question 2  
 public Date end() { return end;} // Question 2  
}

1. Write code that shows the problem the line marked // Question 1.
2. Write code that shows the problem the lines marked // Question 2.
3. Suppose that the class declaration were:

* public class Period { // Question 3
  + Write code that shows the problem.

1. Bloch fixes the constructor as follows:

* public Period (Date start, Date end) {  
   this.start = new Date(start.getTime()); // Defensive copy  
   this.end = new Date(end.getTime()); // Defensive copy  
    
   if (this.start.compareTo(end) > 0) throw new IAE();
  1. Bloch states that clone() would be inappropriate for copying the dates. Write code that shows the problem.
  2. Bloch defers the exception check until the end, which seems to violate normal practice. What's the problem with checking early?

# Module 3 Assignment – (MLO 2) [~2 hours]

## Purpose

Practicing immutablity via Bloch Item 50

## Instructions

* Let us reconsider the Period class shown below

#+begin\_src java  
  
 // Broken “immutable” time period class  
 public final class Period { // Question 3  
 private final Date start;  
 private final Date end;  
  
 /\*\*  
 \* @param start the beginning of the period  
 \* @param end the end of the period; must not precede start  
 \* @throws IAE if start is after end  
 \* @throws NPE if start or end null  
 \*/  
  
 public Period (Date start, Date end) {  
 if (start.compareTo(end) > 0) throw new IAE();  
 this.start = start; this.end = end; // Question 1  
 }  
 public Date start() { return start;} // Question 2  
 public Date end() { return end;} // Question 2  
 }

* Suppose that the class declaration were:
* public class Period { // Question 3
* Write code that shows the problem. That is, you should not only break the immutability of the Period class by writing a suitable sublcass, but you should also develop a plausible case where a client ends up "in trouble" due to the loss of immutability.
* Turn in a **story**.

### Grading Criteria

Grading is in part the technical aspect of breaking immutability, and in part that your client case is plausible.

## 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) [~.5 hour]

## Purpose

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

## Instructions

The specification for Liskov's elements() method is given below.

* Note 1: A Liskov Iterator has only the the hasNext() and next() methods.
* Note 2: As discussed in class the abstract state for such an Iterator is a Stack of objects yet to be produced.
* public Iterator elements()   
  // EFFECTS: Returns a generator that will produce all the elements of  
  // this (as Integers), each exactly once, in arbitrary order.  
  // REQUIRES: this must not be modified while the generator is in use

Consider the code below which uses elements(). Line numbers have been added for reference purposes.

0: IntSet s = new IntSet();   
  
1: s.insert(2);  
2: s.insert(8);  
3: Iterator itr = s.elements();   
4: itr.next();   
5: itr.next();   
6: // See questions below  
7: itr.next();

* show the (stack) contents of itr after line 3

**Sol**: [2,8] or [8,2]

* show the contents of itr after line 5.

**Sol**: []

* If line 6 is s.insert(12); show the contents of itr after line 6?

**Sol**: Undefined behavior or violating the requirements of iteration

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.