Module 9 - Generics

# Overview and Objectives

## In this module we will talk about generics, i.e., parametric polymorphism. We will learn the benefits and proper usage of generics. Specifically, we will learn when we should use generics and how to convert code that does not use generics to one that does.

## 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 demonstrate the benefits and key concepts of generics
2. Compare and describe specification terminologies and concepts (e.g., raw types vs. generic types, arrays vs. lists, wild cards)
3. Convert from non-generic code to generic code, from code using arrays to one using lists.

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

# Learning Materials [3 hours]

## TEXTBOOK READINGS

* Joshua Bloch. Effective Java. Third Edition. Addison-Wesley Professional, 2017, ISBN 978-0-13-468599-1.
  + Chapter 5

# Learning Unit 1 – Intro to Generics (MLO 1) [~0.5 hour]

* So far we talked about **subtype polymorphism** as in relationship between sub and supertype (e.g., Liskov Substitution Principle: supertype operations should work on suptype)
* Another form of polymorphism is **Parametric Polymorphism**, in which a function or a type/class can be written generically so that it can handle values/parameters identically regardless of their type
* A Generic class enables the programmer to specify, with a single class declaration, a set of related types
  + One can write a generic sorting method that could work with Integer arrays, Double arrays, String arrays and so on, to sort the array elements
  + But this can be also done with Subtype Polymorphism?!! Why Generics?
* Generics allow for catching invalid types at **compile time** (instead of runtime)
  + A generic class or method permits you to specify allowable types.
  + If you attempt to use the class or method with an incompatible type, a compile error occurs (as opposed to runtime)
* Example
* // Before Java 5  
  public interface Comparable{  
   public int compareTo (Object o)  
  }  
  Comparable c = new Date();  
  c.compareTo("red"); // RUNTIME ERROR  
    
  // Java 5  
  public interface Comparable<T>{  
   public int compareTo (T o)  
  }  
  Comparable<Date> c = new Date();  
  c.compareTo("red"); // COMPILE ERROR
* Without generics, had to cast every object read from a collection.
* With generics, compiler inserts casts for you automatically and tells compile time if try to insert an object of the wrong type (safer and clearer programs)

# Learning Unit 2 (MLO 1,2,3) [3 hrs]

## Item 26: Don’t Use Raw Types in New Code

* List is a raw type
* List<E> is a generic interface
* List<String> is a parameterized type
  + String is the actual type parameter corresponding to E

// Now a raw collection type – don’t do this  
 private final Collection stamps = ; // Intention: Contains only Stamps  
// Erroneous insertion of coin into stamp collection  
 stamps.add(new Coin(...)); // Oops! We’re set up for ClassCastException later  
  
// Parameterized collection type - typesafe   
 private final Collection<Stamp> stamps = ...;  
 stamps.add(new Coin(...)); // result is instead a compile time error, which is good  
  
// Now a raw iterator type – don’t do this!  
 for (Iterator I = stamps.iterator(); i.hasNext(); ) {  
 Stamp s = (Stamp) i.next(); // might throw a ClassCastException  
 ...// Do something with the stamp  
 }  
// for-each loop over parameterized collection – typesafe  
 for (Stamp s: stamps) { // No (explicit) cast – same code as above  
 ...// Do something with the stamp  
}

List<String> strings = new ArrayList<String>();  
unsafeAdd(strings, new Integer(42));  
String s = strings.get(0); //can cause error at runtime  
  
 // note use of raw types  
 private static void unsafeAdd(List list, Object o) {  
 list.add(o);  
 }  
  
private static void unsafeAdd( List<Object> list, Object o) {  
 list.add(o);  
}

## Item 27: Eliminate Unchecked Warnings

* Generics result in many compiler warnings (annoying but good)
* Eliminate them by fixing your code if possible
* Some fixes are easy:
  + Set<Lark> exaltation = new HashSet(); // warning
  + Set<Lark> exaltation = new HashSet <Lark>(); // no warning
  + Set<Lark> exaltation = new HashSet <>(); // no warning
* As a last resort, suppress the warnings
* If you can’t eliminate a warning, but you can prove that the code that provoked the warning is typesafe, then (and only then) suppress the warning with an @SuppressWarnings("unchecked") annotation.

Set<Lark> exaltation = new HashSet(); // warning  
 Set<Lark> exaltation = new HashSet<Lark>(); // no warning  
  
  
 public <T> T[] toArray (T[] a) {  
 if (a.length < size)  
 @SuppressWarnings(“unchecked”)  
 //copyOf copyes Objects so it would warn that Objects[] is not the same as T[]  
 T[]results = (T[]) Arrays.copyOf(elements, size, a.getClass());  
 return results  
  
 System.arraycopy(elements, 0, a, 0, size);  
 if (a.length > size) a[size] = null;  
 return a; }  
  
/\*  
 ArrayList.java:305: warning [unchecked] unchecked cast  
found : Object[], required T[]   
 return (T[]) Arrays.copyOf(elements, size, a.getClass());  
 \*/

## Item 28: Prefer Lists to Arrays

* Lists play well with generics, arrays do not
  + To support backward compatibility Generics have restrictions
  + This is illegal: new E[], new List<E>[], new List<String>[]
* Arrays are **covariant**; generics are **invariant**
  + Sub[] is a subtype of Super[] -> covariant
  + List<Sub> is not a subtype of List<Super> -> invariant

// Fails at runtime  
Object[] objectArray = new Long[1];  
objectArray[0] = “I don’t fit in!”; // Throws ArrayStoreException  
  
// Won’t compile  
List<Object> o1 = new ArrayList<Long>();  
o1.add(“I don’t fit in!”); // Incompatible types

* Arrays are reified: This means that arrays know and enforce their element type at runtime.
  + If you try to put a String into an array of Long, you’ll get an ArrayStoreException.
  + Generics are erased, they exist at compile time only
* Generics excel at compile time (better). Arrays excel at runtime.

### Example: Covariance vs Invariance

Object[] objectArray = new Long[1]; // compiler error? runtime error?  
// Long[] is-a Object[]: covariant  
objectArray[0] = “Do I fit in?”; // compiler error? runtime error?  
// putting a string in an array of Longs Throws ArrayStoreException: reified  
  
  
List<Object> o1 = new ArrayList<Long>(); // compiler error? runtime error?  
// does not compile: invariant  
o1.add(“Do I fit in?”); // Incompatible types

* Not compiling is better than a runtime exception.
* This is basically an argument for why invariance is preferable to covariance for generics.

### At this point, do Group Excercise 2 (Chooser)

## Item 29: Prefer Generic Types

**Generify** code

private Object[] elements; // private E[] elements (1st attempt) /=> private List<E> elements ..

## Prefer Generic Methods (Item 30)

// Uses raw types – unacceptable! (Item 23)  
public static Set union (Set s1, Set s2) {   
 Set result = new HashSet(s1); // Generates a warning   
 result.addAll(s2); // Generates a warning  
 return result;  
}  
  
// Generic method   
 public static <E> Set <E> union (Set <E> s1, Set <E> s2) {   
 Set <E> result = new HashSet <E> (s1);   
 result.addAll(s2);   
 return result;  
}

## Bounded WildCards

public class Stack <E> {  
 public Stack()  
 public void push( E e )   
 public E pop()  
 public boolean isEmpty()  
 }  
  
 // pushAll method without a wildcard type – deficient!  
 // only add E (but not its subtype)  
 public void pushAll( Iterable<E> src) {  
 for (E e : src) { push(e); }  
 }  
  
  
 // wildcard type for parameter that serves as an E producer  
 // allows everything that is subtypes of E  
 public void pushAll( Iterable<? extends E> src) {  
 for (E e : src) { push(e); }  
 }  
  
  
 // wildcard type for parameter that serves as an E consumer  
 public void popAll ( Collection<? super E> dst) {  
 while (!isEmpty()) { dst.add(pop()); }  
 }  
  
//PECS: procer extends and consumer super

# In class 1 (MLO 1,2,3) [.25 hrs]

Given the following variable declarations, independently consider the given 6 sequences of Java instructions.

String string = "bat";  
Integer x = 7;  
Object[] objects;  
List rawList;  
List < Object > objectList;  
List < String > stringList;

Identify any code that results in a compiler error or warning. Identify any code that raises a runtime exception. Once a compiler error is noted, you do not need to analyze the sequence further.

1. objects = new String[1];  
   objects[0] = string;  
   objects[0] = x;
2. objects = new Object[1];  
   objects[0] = string;  
   objects[0] = x;
3. stringList = new ArrayList < String >();  
   stringList.add(string) ;
4. objectList = new ArrayList < String >();  
   objectList.add(string) ;
5. objectList = new ArrayList < Object >();  
   objectList.add(string) ;  
   objectList.add(x) ;
6. rawList = new ArrayList();  
   rawList.add(string) ;  
   rawList.add(x) ;

# In class 2 (MLO 1,2,3) [.5 hrs]

// Chooser - a class badly in need of generics!  
// Bloch 3rd edition, Chapter 5, Item 28: Prefer lists to arrays  
  
public class Chooser {  
 private final Object[] choiceArray;  
  
 public Chooser (Collection choices) {  
 choiceArray = choices.toArray();  
 }  
  
 public Object choose() {  
 Random rnd = ThreadLocalRandom.current();  
 return choiceArray [rnd.nextInt(choiceArray.length)];  
 }  
}

* First, simply generify by adding a type to the Chooser class. What is the compiler error with this approach?
* How can you turn the compiler error into a compiler warning?
* Can this warning be suppressed? Should it?
* How can you adopt Bloch's advice about arrays and lists to get a typesafe Chooser class without doing anything else that is complicated?
* Add rep invariants and contracts (e.g., throw exceptions in unwanted cases); check if code satisfies these; and if not modify code to satisfy them. This question will take the most time!
* Add a addChoice method to the API and write appropriate contracts for it

SOLUTIONS: First attempt

public class Chooser<T> {  
 private final T[] choiceArray;  
  
 public Chooser (Collection<T> choices) {  
 choiceArray = choices.toArray();// compiler errors: cannot convert to T,  
  
 @supresswarning..  
 choiceArray = (T[]) choices.toArray(); //cast to (T[]), got a warning, supress it because we know it is safe because choiceArray is of type T  
 }  
  
 public T choose() {   
 Random rnd = ThreadLocalRandom.current();  
 return choiceArray [rnd.nextInt(choiceArray.length)];

List attempt

public class Chooser {  
 private final List<T> choiceArray;  
  
 //RepInv: choicearray is not Null and not empty  
  
 //POST: @throw IAE if choices is empty  
 //POST: @throw NPE if choice contains null  
 //Post: create a chooser with choices  
 public Chooser (Collection<T> choices) {  
 if (choices.size() == 0) throw new IllegalArgumentException(); // ADD  
 if (choices.contains(null)) throw new NullPointerExeption();//ADD  
 choiceArray = new ArrayList<>();  
 }  
  
 //POST: @throws ISE if empty, else return random choice  
 //CHECK: choiceArray never changed so RI maintained,  
 public Object choose() {  
 if(choiceList.size() == 0) throw IllegalStateException(); // NEW CODE  
 Random rnd = ThreadLocalRandom.current();  
 return choiceArray [rnd.nextInt(choiceArray.length)];  
 }  
  
 //Post @throw NPE if choice is null  
 //POST: add choice to this  
 public void addChoice(T choice){  
 if (choice == null) throw new NullPointerException();  
 choiceList.add(choice);  
 }  
}

# In class 3 (MLO 1,2,3) [.5 hrs]

public class BoundedQueue {  
  
 private Object rep[];  
 protected int front = 0;  
 protected int back = -1;  
 private int size = 0;  
 protected int count = 0;  
  
 public BoundedQueue(int size) {  
 if (size > 0) {  
 this.size = size;  
 rep = new Object[size];  
 back = size - 1;  
 } }  
  
 public boolean isEmpty() { return (count == 0); }  
  
 public boolean isFull() { return (count == size); }  
  
 public int getCount() { return count; }  
  
 public void put(Object e) {  
 if (e != null && !isFull()) {  
 back++;  
 if (back >= size)  
 back = 0;  
 rep[back] = e;  
 count++;  
 } }  
  
 public Object get() {  
 Object result = null;  
 if (!isEmpty()) {  
 result = rep[front];  
 rep[front] = null;  
 front++;  
 if (front >= size)  
 front = 0;  
 count--;  
 }  
 return result;  
 }  
}

* **Generify**!
* Can you add a putAll() method? A getAll() method?
* Recall that we used this same example in in-class 6 as a vehicle for applying Liskov's ideas to make code easier to understand.

# Assignment – (MLO 1, 2, 3) [~2 hours]

## Purpose

Practicing using Generics.

## Instructions

Consider the BoundedQueue example from the in-class exercise.

Complete the generic part of the exercise: The result should be fully generic, and there should not be any compiler warnings. You should adopt Bloch's advice about lists vs. arrays; doing so will eliminate the need for many of the instance variables.

Keep the same methods, but update the behavior (and document with contracts!) to include exception handling for all cases not on the happy path.

Include the constructor in your considerations. In particular, consider whether you think a zero-sized buffer is a reasonable possibility. Document your reasoning. This is less about a right vs. wrong answer than a careful consideration of the consequences of the decision.

Add putAll() and getAll(). Define the method signatures carefully. Use exception-handling consistent with that for get() and put(). Use bounded wildcards as appropriate. Note that putAll() has a special case where there isn't sufficient space in the bounded queue. Adopt a solution you think Bloch and/or Liskov would approve of. In particular, Bloch prefers that when methods throw exceptions, there is no change to the state of the object.

### Grading Criteria

As before, turn in a clear, concise story demonstrating completion of the assignment.

## Deliverable

* Submit a .java file for your implementation.

## Due Date

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

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

## Purpose

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

## Instructions

Consider the following code:

public class Example <E> {  
 String string = "ant";  
 Integer seven = 7;  
 E e = null;  
 Object[] objects;  
 List < Object > listObject;  
 List < E > listE;  
 public void m() {  
 // Java code for questions appears here  
 }  
}

Independently consider the following 5 sequences of Java instructions. For each sequence, what of the following choices will happen ? (i) compiler warning; (ii) compiler error; (iii) runtime exception; or (iv) normal run

objects = new E[1];  
objects[0] = e;

listE = new ArrayList < E >();  
listE.add(e);  
listObject = listE;

listObject = new ArrayList < String >();  
listObject.add(string) ;  
listObject.add(seven) ;

objects = new Object[1];  
objects[0] = string;  
objects[0] = seven;

objects = new String[1];  
objects[0] = string;  
objects[0] = seven;

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.