Module 3 - Exceptions

# Overview and Objectives

## In this module we will learn about Exceptions. While we use the Java Exception system, the knowledge we learn is more general and applicable to other languages. Among various topics on Exceptions, we will learn about how to integrate Exceptions as part of the program specification and how to write specifications with Exceptions to indicate unexpected behaviors. We will also learn how to use Exceptions, such as checked and unchecked ones, properly.

## 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 key concepts and reasons for using Exceptions
2. integrate Exceptions into program specifications and write correct specifications with exceptions
3. compare and describe exceptions terminologies and concepts (e.g, checked vs. unchecked exceptions)

# 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 4: Exceptions
* Joshua Bloch. Effective Java. Third Edition. Addison-Wesley Professional, 2017, ISBN 978-0-13-468599-1.
  + Chapter 10

# Module 3 Learning Unit 1 – Abstractions (MLO 1) [~0.5 hour]

* Partial procedures, where there are requires (preconditions on inputs), are generally a bad idea.
  + no guarantee that the procedure will be called on inputs satisfying requires.
  + when run on invalid inputs, the behavior of the procedure is undefined, e.g., it can do anything such as returning errors or run forever.
  + difficult to track down error.
* Total procedures, where there are no requires, are more robust
  + it works for all inputs (and in the case it doesn't like certain inputs, at least it will tell the user so, e.g., by raising an exception!)
* Exception:
  + A good approach to turn partial into total
  + conveys information about unusual situations
  + allows a procedure to terminate either normally, by returning a result, or exceptionally.

# Learning Unit 2 – Specifications with Exceptions (MLO 2) [~1 hour]

* Has throw clause in header

public static int fact (int n) throws NonPositiveException

* states that fact can terminate by throwing an exception; specifically, an object of type NonPositiveException.
* In general, specifications of proceduring throwing exceptions must
  1. specify exceptions thrown in headers
  2. explain in **effects** clause what causes exceptions to be thrown

public static int search (int[ ] a, int x)  
 throws NullPointerException, NotFoundException  
 // EFFECTS: If a is null throws NullPointerException; else if x is not  
 // in a throws NotFoundException; else returns i such that x < a[i].

* example of method throwing multiple exceptions. Notice headers has 2 exceptions and effects describe when these 2 exceptions are thrown.
* When a procedure has side effects, its specification must make clear how these interact with exceptions. Exactly how side-effects occur must be described in the effects clause.

public static void addMax (Vector v, Integer x)  
 throws NullPointerException, NotSmallException  
 // REQUIRES: All elements of v are Integers.  
 // MODIFIES: v  
 // EFFECTS: If v is null throws NullPointerException; if v contains an  
 // element larger than x throws NotSmallException; else adds x to v.

# Learning Unit 2 – Effective Exceptions (MLO 1, 2, 3) [2 hour]

* Exception types are subtypes of either Exception or RunTimeException
  + This is a brief description of Java exceptions. The Javadocs (<https://docs.oracle.com/javase/8/docs/api/java/lang/Exception.html>) has much more details and also might be more up to date.

## Checked and Unchecked Exceptions

* Unchecked exceptions, e.g., NullPointerException, IndexOutofBoundException, are sub-types of RuntimeException
* checked exceptions (runtime exception and errors), e.g., IOException, are subtypes of Exception but not of RuntimeException.

## Item 69: Use exceptions only for exceptional conditions

* Exceptions are, as their name implies, to be used only for exceptional conditions; they should never be used for ordinary control flow.
* A well-designed API must not force its clients to use exceptions for ordinary control flow
* Summary: Exceptions are designed for exceptional conditions. Don’t use them for ordinary control flow, and don’t write APIs that force others to do so.

## Item 70: Use checked exceptions for recoverable conditions and runtime exceptions for programming errors

* use checked exceptions for conditions from which the caller can reasonably be *expected to recover*.
  + force caller to handle the exception in a catch clause or popagate it outward
* Use runtime exceptions to indicate programming errors.
  + The great majority of runtime exceptions indicate **precondition/requires** violations. E.g., ArrayIndexOutOfBoundsException might indicate a violation on a precondition about an array input.
* Summary:
  + throw checked exceptions for recoverable conditions and unchecked exceptions for programming errors.
  + When in doubt, throw unchecked exceptions.
  + Provide methods on your checked exceptions to aid in recovery.

## Item 71: Avoid unnecessary use of checked exceptions

* Summary: when used sparingly, checked exceptions can increase the reliability of programs; when overused, they make APIs painful to use.

## Item 72: Favor the use of standard exceptions

## Item 73: Throw exceptions appropriate to the abstraction

* it's bad when a method throws an exception that has no apparent connection to the task that it performs (happens when method propagates an exception thown by a lower-level abstraction)).
* (exception translation) higher layers should catch lower-level exceptions and, in their place, throw exceptions that can be explained in terms of the higher-level abstraction.
* // Exception Translation  
  try {  
   ... // Use lower-level abstraction to do our bidding  
  } catch (LowerLevelException e) {  
   throw new HigherLevelException(...);  
  }

## Item 74: Document all exceptions thrown by each method

* This is how we include and specify exceptions in the specification of the method
* Always declare checked exceptions individually, and document precisely the conditions under which each one is thrown
* If an exception is thrown by many methods in a class for the same reason, you can document the exception in the class’s documentation comment

## Item 75: Include failure-capture information in detail messages

* To capture a failure, the detail message of an exception should contain the values of all parameters and fields that contributed to the exception.
* /\*\*  
   \* Constructs an IndexOutOfBoundsException.  
   \*  
   \* @param lowerBound the lowest legal index value  
   \* @param upperBound the highest legal index value plus one  
   \* @param index the actual index value  
   \*/  
  public IndexOutOfBoundsException(int lowerBound, int upperBound,  
   int index) {  
   // Generate a detail message that captures the failure  
   super(String.format(  
   "Lower bound: %d, Upper bound: %d, Index: %d",  
   lowerBound, upperBound, index));  
    
   // Save failure information for programmatic access  
   this.lowerBound = lowerBound;  
   this.upperBound = upperBound;  
   this.index = index;  
  }
* Don't use sensitive private details in the detail message

## Item 76: Strive for failure atomicity

* A failed method invocation should leave the object in the state that it was in prior to the invocation
* If cannot do this, then the API documentation should clearly indicate what state the object will be left in.

## Item 77: Don’t ignore exceptions

* Don't use an empty catch block (defeat the purpose of exception)
* If have to ignore an exception, then provide comment as well as named the exception variable ignored
* Future<Integer> f = exec.submit(planarMap::chromaticNumber);  
  int numColors = 4; // Default; guaranteed sufficient for any map  
  try {  
   numColors = f.get(1L, TimeUnit.SECONDS);  
  } catch (TimeoutException | ExecutionException ignored) {  
   // Use default: minimal coloring is desirable, not required  
  }

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

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

# Assignment – (MLO 1, 2) [~2 hours]

## Purpose

Practicing using exceptions in code

## Instructions

You are given the following exceptions:

* IllegalArgumentException
* NumberFormatException
* NullPointerException
* IOException
* IndexOutOfBoundsException
* OutOfMemoryError
* StackOverflowError

declare the exceptions in the method signature using throws and handle approach is that you will be forced to handle the exception

1. For each exception above, write a method whose code throws that exception.

You cannot use "throw new …" inside the method to achieve that.

main() {  
 throwsIAE();  
 throwsNFE();  
 throwsNPE();  
 throwsIOE();  
 throwsIOOBE();  
 throwsOOME();  
 throwsSOE();  
}

1. Write appropriate try/catch/finally statements around each method call.

## Deliverable

* Submit a .java file for your implementation.

## Due Date

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

# Module 3 Quiz (MLO 1, 2, 3) [~.5 hour]

## Purpose

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

## Instructions

Consider the following 2 specifications, the second of which has an associated implementation (note that the 2nd implementation uses the specification of the first):

public static int minIndex (int [] a) {... implementation omitted ... }  
 // EFFECTS: if a null throw NullPointerException, else if a.length=0 throw  
 // IllegalArgumentException else return index of some min element in a.  
  
public static void setSmall (int [] a, int i) {  
 // REQUIRES: a != null, 0 <= i < a.length  
 // MODIFIES a  
 // EFFECTS: rearranges elements in array a so that some smallest element is at index i  
 int j = minIndex(a); int t = a[j]; a[j] = a[i]; a[i] = t; }

1. Suppose we wish to transform the setSmall precondition a != null into defined behavior with an exception.
   1. What Java exception would you use for this transformation?
   2. Rewrite the precondition and postcondition for setSmall() to achieve this result.
   3. Does the given implementation of setSmall() satisfy this revised specification?
2. Suppose we wish, in addition, to transform the setSmall() precondition 0 <= i < a.length into defined behavior with an exception.
   1. What exception would you use this transformation? Hint: Note that i is described as an "index".
   2. Extend your rewrite of the precondition and postcondition for setSmall() to achieve this result.
   3. Does the given implementation of setSmall() satisfy this revised specification?

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.