# Overview and Objectives

This module focuses on Procedural (or Method) Abstraction, which allows us to “hide” irrelevant details from complicate classes and programs and help reasoning about programs. Specifically, we will look at abstraction by specifications, which reveal essential behaviors of programs without disclosing implementation details. Finally, we will look at designing and implementing effective specifications.

## 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 benefits and key concepts of procedural (method) abstraction
2. Compare and describe specification terminologies and concepts (e.g., total vs partial procedurals, requires/effects/modifies)
3. Consruct correct and desirable formal specifications for procedurals

# 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 3: Procedural Abstraction

# Learning Unit 1 – Benefits of Abstractions (MLO 1) [~0.5 hour]

* Abstraction “abstracts” programs from “irrelevant” details,
  + describing only those details that are relevant (e.g., for inputs, we might only care about the types of the input is integer, for an isPrime procedure, we only care about “what” it does (check that a number is prime), but we don’t care about “how” it is implemented)
* Allows for easier reasoning and understanding (only care about relevant details and ignore irrelevant ones)
* Also allows us to change to another implementation without affecting the meaning of any program that uses the abstraction
  + E.g., we can change the implementation of isPrime to a different algorithm, rewrite it in a different languages, etc.
* Benefits:
  + Locality: The implementation of an abstraction can be read or written without needing to examine the implementations of any other abstractions.
    - programs can understand what the program does without having to analyze its implementation details
    - allow developers to work independently
  + Modifiability: An abstraction can be reimplemented without requiring changes to any abstractions that use it.
    - allow for tuning performance

# Learning Unit 2 – Program Specifications and Abstractions (MLO 1, 2) [~2.5 hour]

* **Procedural Abstraction using Specifications** allows us to break down the program into modules (procedures) while hiding the details of implementations
* **The Specifications** reveal the essential information about the behavior of a module (procedure) without disclosing all the details of its implementation
* **Procedural Abstraction by Specification** allows us to change the implementation of a procedure without affecting the meaning of any code that uses the procedure
* Program/Procedural Specifications: abstraction defining program behavior
* Unlike an implementation, we don’t write specification in a programming language (e.g., while the implementation in Java, the specification is not)
  + usually written as comment in English

## Specification of a procedural:

* consists of **header** and description of **effects**.
* **Header** gives the name of the procedure, the number, order, and types of its parameters, and the type of its result
  + it also lists any exceptions thrown by the procedure (we’ll discuss more about exception in later modules)
  + void removeDupls (Vector v);  
    float sqrt (float x);
* **Effects** tells us the behavior of the procedural. Consists of three parts: the requires, modifies, and effects clauses
* // REQUIRES: This clause states any constraints on inputs (i.e., preconditions)  
  // EFFECTS: This clause defines the behavior (i.e., postconditions)   
  // MODIFIES: This clause identifies all modified inputs
  + **Requires**: states the constraints over the inputs (e.g., input x in sqrt cannot be negative). If there is no require, then the procedural is **total**. Otherwise, the procedura ls **partial**
  + **Effects**: describes the behavior of the procedure for all inputs satisfies the requires clause.
    - Effects only happen under the assumption that the requires clause is satisfied. If the requires are not satisfied, effects are undefined.
  + Modifies: lists the names of any inputs that are modified by the procedure. If some inputs are modified, we say the procedure has a **side effect**.

### Example

public class Arrays {  
 // OVERVIEW: This class provides a number of standalone procedures that  
 // are useful for manipulating arrays of ints.  
  
 public static int search (int[ ] a, int x)  
 // EFFECTS: If x is in a, returns an index where x is stored;  
 // otherwise, returns -1.  
  
 public static int searchSorted (int[ ] a, int x)  
 // REQUIRES: a is sorted in ascending order  
 // EFFECTS: If x is in a, returns an index where x is stored;  
 // otherwise, returns -1.  
  
 public static void sort (int[ ] a)  
 // MODIFIES: a  
 // EFFECTS: Rearranges the elements of a into ascending order  
 // e.g., if a = [3, 1, 6, 1] before the call, on return a = [1, 1, 3, 6].  
}

* A specification of a class, Arrays, which provides a number of standalone procedures that are useful for manipulating arrays of integers. In the specification, search and searchSorted do not modify their inputs, but sort modifies its input, as indicated in the modifies clause.
  + Note the use of an example in the sort specification. Examples can clarify a specification and should be used whenever convenient.
* sort and search are total (no require); searchSorted is partial; it only does its job if its argument array is sorted.

### Example: sortedSearch

public static int sortedSearch (int[]a, int x)  
// Requires/Preconds: a is sorted in ascending order  
// Effects/Postconds: if x is in a returns an index where // x is stored; otherwise, returns -1

### Example: sort

public static void sort (int[] a)  
// Modifies: a  
// Effects: rearranges the elements of a into ascending order  
// E.g. if a = [3,1,6,1] before the call, then  
// a = [1,1,3,6] after the call

## Instructor Screencast: TITLE

# Learning Unit 3 - Implementing Procedures and Designing Procedural Abstraction (MLO 2,3) (1 hrs)

## Implementation Procedures

* The implementation of a procedure should satisfy the procedural specification, i.e., produce the behavior defined by its specification.
  + modify only those inputs that appear in the modifies clause;
  + and if all inputs satisfy the requires clause, it should produce the result specified in the effects clause.

## Properties of Precedural and Their Implementations

In general, we want the following properties:

Minimality

One specification is more minimal than another if it contains fewer constraints on allowable behavior.

Underdetermined behavior

A procedure is underdetermined if for certain inputs its specification allows more than one possible result.

Deterministic implementation

An implementation of a procedure is deterministic if, for the same inputs, it always produces the same result. Implementations of underdetermined procedures are almost always deterministic.

Generality

One specification is more general than another if it can handle a larger class of inputs.

Moreover, if possible, we prefer **total** instead of **partial** procedures

total

a procedure is **total** if its behavior specified for all legal inputs; otherwise, it is partial. The specification of a partial procedure always contains a requires clause.

Partial

procedures are less safe than total ones. Therefore, they should be used only when the context of use is limited or when they enable a substantial benefit, such as better performance.

* When possible, the implementation should check the constraints in the requires clause and throw an exception if they are not satisfied.

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

Consider the following implementation:

public static List<Integer> tail (List<Integer> list) {  
  
 // REQUIRES/PRECONDS: ???  
 // EFFECTS/POSTCONDS: ???  
  
 List<Integer> result = new ArrayList<Integer>(list);  
 result.remove(0);  
 return result;  
}

Hint: also look at the Javadoc (for remove)

1. What does the *implementation* of tail do in each of the following cases? How do you know: Running the code or reading an API description?
   * list = null
   * list = []
   * list = [1]
   * list = [1, 2, 3]
2. Write a **partial** specification that matches the “happy path” part of the implementation’s behavior (happy path: normal execution, no exception or crashing or something unexpected).
3. Rewrite the specification to be **total**. Use exceptions if needed.
4. The resulting specification might have a problem. What is it? (hint: specification should be more general and not tied to the implementation)
5. *Rewrite* the specification to address this problem. *Rewrite* the code to match the new specification.

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

Understanding Contracts

Consider the 3 methods hasNext , next, and remove in the Java [Iterator](https://docs.oracle.com/javase/7/docs/api/java/util/Iterator.html) interface:

* For each method, identify all preconditions and postconditions.
* For each precondition, identify a specific input that violates the precondition.
* For each postcondition, identify an input specific to that postcondition.

### Instructor Screencast: TITLE

### Interactive Element: TITLE

### Instructor Screencast: TITLE

Link to MP4 File

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

## Purpose

For this assignment, you’ll build a *very* small piece of Java for a contract with preconditions, transform the contract so that all preconditions become postconditions (i.e., make it a *total* contract), and then re-implement appropriately.

## Instructions

* Consider a method that calculates the number of months needed to pay off a loan of a given size at a fixed *annual* interest rate and a fixed *monthly* payment. For instance, a $100,000 loan at an 8% annual rate would take 166 months to discharge at a monthly payment of $1,000, and 141 months to discharge at a monthly payment of $1,100. (In both of these cases, the final payment is smaller than the others; I rounded 165.34 up to 166 and 140.20 up to 141.) Continuing the example, the loan would never be paid off at a monthly payment of $100, since the principal would grow rather than shrink.

Define a Java class called Loan. In that class, write a method that satisfies the following specification:

/\*  
 @param principal: Amount of the initial principal  
 @param rate: Annual interest rate (8% rate expressed as rate = 0.08)  
 @param payment: Amount of the monthly payment  
\*/  
public static int months (int principal, double rate, int payment)  
// Requires: principal, rate, and payment all positive and payment is sufficiently large to drive the principal to zero.  
// Effects: return the number of months required to pay off the principal

Note that the precondition is quite strong, which makes implementing the method easy. You should use double precision arithmetic internally, but the final result is an integer, not a floating point value. The key step in your calculation is to change the principal on each iteration with the following formula (which amounts to monthly compounding):

newPrincipal = oldPrincipal \* (1 + monthlyInterestRate) - payment;

The variable names here are explanatory, not required. You may want to use different variables, which is fine.

*To make sure you understand the point about preconditions, your code is required to be minimal. Specifically, if it possible to delete parts of your implementation and still have it satisfy the requirements, you’ll earn less than full credit.*

* Now modify months so that it handles **all** of its preconditions with exceptions. Use the standard exceptions recommended by Bloch. Document this with a revised contract. You can use JavaDoc or you can simply identify the postconditions.

## Deliverable

* Submit a .java file for your implementation.
* *Grading Criteria*:
  + Adherence to instructions.
  + Minimal implementation.
  + Preconditions are correctly converted to exceptions.
  + Syntax: Java compiles and runs.

## Due Date

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

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

## Purpose

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

## Instructions

Specify and implement a procedure isPrime that determines whether an integer is prime.

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.