Module 13 - Enums and Annotations

# Overview and Objectives

## OVERVIEW

In this module we learn about two special-purpose families of reference types in Java: a kind of class called an enum type, and a kind of interface called an annotation type. This module discusses best practices for using these type families.

## 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 the key concepts and usage of Annotation and Enums (CL01)
2. Define and use Annotations/Enums properly (CL01)

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

# Learning Materials [~100 pages, ~3.5 hours]

## TEXTBOOK READINGS

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

# Learning Unit 1 – Annotations (MLO 1,2) [~0.5 hour]

## Annotations

* Annotations provide data about a program that is not part of the program itself.
* They have no direct effect on the operation of the code they annotate.
* An annotation is an attribute of a program element, e.g., attribute, method, class, package. It is a metadata (data about data).
* There are built-in annotations and user defined annotations
* Information for the compiler — Annotations can be used by the compiler to detect errors or suppress warnings.
* Information to Software tools — Software tools can process annotation information to generate code (javadoc), documentation, XML files, and so forth.
* Runtime processing — Some annotations are available to be examined at runtime.

## Built-in Annotations

-Javadoc annotations

/\*\*  
 \* Locate a value in a collection.  
 \*  
 \* @param value the sought-after value  
 \* @return the index location of the value  
 \* @throws NotFoundException  
 \*/  
 int search( Object value ) { …

* @transient - an ad hoc annotation indicating that a field should be ignored by the serialization subsystem
* @Deprecated class Y {  
  public abstract int foo();  
  }  
    
  class X extends Y {  
   @SuppressWarnings(“unchecked”) List numbers;  
   @Override public int foo() { ... }  
  }
* @Deprecated indicates that the marked element is deprecated and should no longer be used. The compiler generates a warning whenever a program uses a method, class, or field with the @Deprecated annotation. When an element is deprecated, it should also be documented using the Javadoc @deprecated tag.
* // Javadoc comment follows   
  /\*\*   
  \* @deprecated … note how the d is lower case  
  \* explanation of why it was deprecated   
  \*/   
  @Deprecated   
  static void deprecatedMethod() { } }
* @Override—the @Override annotation informs the compiler that the element is meant to override an element declared in a superclass. To avoid subtle errors, e.g., equals(MyClass f) vs. equals(Object o)
* // mark method as a superclass method   
  // that has been overridden   
  @Override   
  int overriddenMethod() { }
* @SuppressWarnings—tells the compiler to suppress specific warnings that it would otherwise generate. Below, the annotation causes the warning to be suppressed.
  + Java has two categories or warnings: "deprecation" and "unchecked"
  + To suppress both categories of warnings, use the following syntax:
    - @SuppressWarnings({"unchecked", "deprecation"})
    - // use a deprecated method and tell   
      // compiler not to generate a warning   
      @SuppressWarnings("deprecation")   
      void useDeprecatedMethod() {   
      // deprecation warning suppressed   
      objectOne.deprecatedMethod();   
      }

## User Defined Annotations

* Definition:
* public @interface Author {  
   String name();  
   String date();  
  }
* Usage:
* @Author(   
  name = "Benjamin Franklin",   
  date = "3/27/1763"   
  )   
  class MyClass() { }

## Annotation Type Declaration

* Similar to normal interface declarations:
* An at-sign @ precedes the interface keyword
* Each method declaration defines an element of the annotation type
* Methods can have default values

public @interface RequestForEnhancement {  
 int id();  
 String synopsis();  
 String engineer() default "[unassigned]";   
 String date() default "[unimplemented]";   
}

## Annotating Declarations

* An annotation instance consists of
  + the "@" sign
  + the annotation name
  + a parenthesized list of name-value pairs
* Example:
* @RequestForEnhancement(  
   id = 2868724,  
   synopsis = "Enable time-travel",  
   engineer = "Mr. Peabody",  
   date = "4/1/3007"  
  )  
  public static void travelThroughTime(Date destination) { ... }
* In annotations with a single element, the element should be named value:
* public @interface Copyright {  
   String value();  
  }
* It is permissible to omit the element name and equals sign (=) in a single-element annotation:
* @Copyright("2002 Yoyodyne Propulsion Systems")  
  public class OscillationOverthruster { ... }
* If no values, then no parentheses needed:
* public @interface Preliminary { }   
  @Preliminary public class TimeTravel { ... }

## Annotation vs. Comments

* Suppose that a software group has traditionally begun the body of every class with comments providing important information:
* public class Generation3List extends Generation2List {   
  // Author: John Doe   
  // Date: 3/17/2002   
  // Current revision: 6   
  // Last modified: 4/12/2004   
  // By: Jane Doe   
  // Reviewers: Alice, Bill, Cindy   
    
  // class code goes here   
  }
* The syntax for using annotation instead is:
* public @interface ClassPreface {   
  String author();  
  String date();   
  int currentRevision() default 1;   
  String lastModified() default "N/A";   
  String lastModifiedBy() default "N/A";   
    
  // Note use of array   
  String[] reviewers();   
  }
* Example:
* @ClassPreface (   
  author = "John Doe",   
  date = "3/17/2002",   
  currentRevision = 6,   
  lastModified = "4/12/2004",   
  lastModifiedBy = "Jane Doe",   
  // Note array notation   
  reviewers = {"Alice", "Bob", "Cindy"}   
  )   
  public class Generation3List extends Generation2List {   
  // class code goes here   
  }

## Annotations

* To make the information in @ClassPreface appear in Javadoc-generated documentation, you must annotate the @ClassPreface definition itself with the @Documented annotation:

// import this to use @Documented   
import java.lang.annotation.\*;   
@Documented   
@interface ClassPreface {   
 // Annotation element definitions   
}

* Example – JUnit Annotations
  + Annotations in JUnit 4:
    - @Test – annotates test method
    - @Before, @After– annotates setUp() and tearDown() methods for each test
    - @BeforeClass, @AfterClass – class-scoped setUp() and tearDown()
    - @Ignore – do not run test
* Prefer Annotations to Naming Patterns
  + Prior to 1.5, Naming Patterns were common
  + Example: JUnit test methods
  + void testSafetyOverride() // Junit 3.x thinks this is a test  
    void tsetSafetyOverride() // Oops! Engineers can’t type
  + Annotations Are Far Better
    - Diagnostics for Misspelled Annotations
    - Annotations Allow Parameters
  + Examples:
    - mock of JUnit
    - // Marker annotation type declaration  
      import java.lang.annotation.\*;  
        
      /\*\*  
       \* Indicates that the annotated method is a test method.  
       \* Use only on parameterless static methods  
       \*/  
      @Retention (RetentionPolicy.RUNTIME)  
      @Target(ElementType.METHOD)  
      public @interface Test {}  
        
      // Program with annotations  
      public class Sample {  
       @Test public static void m1() {} // Test should pass  
       public static void m2() {} // Not a @Test  
       @Test public static void m3() { // Test should fail  
       throw new RuntimeException(“Boom”); }  
       @Test public void m4() // Invalid nonstatic use  
      }
    - The Simple Version of JUnit
    - // Sample code processes marker annotations – See Bloch for variations  
      import java.lang.reflect.\*;  
        
      public class RunTests {  
       public static void main(String[] args) {  
       int tests = 0; int passed = 0;  
       Class testClass = Class.forName(args[0]);  
       for (Method m : testClass.getDeclaredMethods()) {  
       if (m.isAnnotationPresent(Test.class)) {  
       tests++;  
       try { m.invoke(null); passed++; }  
       catch (InvocationTargetException ite) {  
       System.out.println(m + “ failed: “ + ite.getCause()); }  
       catch (Exception e) {  
       System.out.println(“Invalid @Test: “ + m); }  
       } } }  
       System.out.printf(“Pass: %d, Fail: %d%n”, passed, tests – passed);  
      } }

## Consistently Use the @Override Annotation

* Most Important Standard Annotation
  + Regular Use Prevents Overload/Override Bugs
    - public boolean equals (SomeClass c) { …}
* IDEs Can Provide Code Inspections
  + Override Exactly Where You Want
    - And nowhere else
* @Override Allowed on Interface Methods
  + Important for Abstract Classes and Interfaces

## User Marker Interfaces to Define Types

* Marker Annotations (Item 35) Are Not Types
  + Interfaces Are Types
* Marker Interfaces Do Not Add Methods
  + Unlike Mixin Interfaces

Example Marker Interfaces

* Serializable // Marks Object as Serializable
* Set // Arguably a marker interface
* If You Want a Type, Do Use an Interface
  + If You Don’t Want a Type, Don’t (See Item 19)

## Reference

* <http://docs.oracle.com/javase/tutorial/java/javaOO/annotations.html>
* An Introduction to Java Annotations
  + <http://www.developer.com/java/other/article.php/3556176>

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

## Enumerations (Enums)

* Some Types have a small/finite set of immutable values, such as:
  + MonthOfYear: January, February, …
  + DayOfWeek: Monday, Tuesday, …
  + CompassPoints: north, south, east, west
* It makes sense to define this small/finite set of values as named constants, called an enumeration.
* Java has the enum construct to make this convenient:  
  public enum Month { JANUARY, FEBRUARY, MARCH, ..., DECEMBER };
  + This enum defines a type Month, in the same way that class and interface define new types.
  + It also defines a set of named values (instances), which are shown in all-caps because they are effectively public static final constants (by convention). So you can now write:
  + Month thisMonth = MARCH;
* This idea is called an enumeration because you are explicitly listing all possible instances.
* C also has it
* enum week{Mon, Tue, Wed, Thur, Fri, Sat, Sun}; // {0, 1, 2, 3, 4, 5, 6}  
    
  int main() {  
   enum week day;  
   day = Wed;  
   printf("%d",day); //prints 2  
   return 0;  
  }
* All enum types have some automatically-provided operations:
  + ordinal() is the index of the value in the enumeration, so JANUARY.ordinal() returns 0.
  + compareTo() compares two values based on their ordinal numbers.
  + name() returns the name of the value’s constant as a string, e.g. JANUARY.name() returns "JANUARY".
  + toString() has the same behavior as name()
  + values() returns an array of all of the values of the enum in the order they are declared
* Example:
  + considering the following:
  + public enum DayOfWeek { MONDAY, TUESDAY…};  
    DayOfWeek day = …  
    if (day.equals(SATURDAY) || day.equals(SUNDAY)) {  
     System.out.println("It's the weekend");  
    }
  + You can also write it safely like below.
  + if (day == SATURDAY || day == SUNDAY) {  
    System.out.println("It's the weekend");  
    }
  + Why the above is possible?
    - In an enumeration there is only ever one object in memory representing each value of the enumeration, and there is no way for a client to create more (no constructors!).
    - So == is no different than equals().
    - Therefore the code below is also safe:
    - switch (direction) {  
      case NORTH: return "polar bears";  
      case SOUTH: return "penguins";  
      case EAST: return "elephants";  
      case WEST: return "llamas";  
      }
  + But unlike int values (as in C), enumerations have more static checking:
  + // static error: MONDAY has type DayOfWeek, not type Month   
    Month firstMonth = MONDAY;
  + Rich enum Example(1)
  + public enum Month { JANUARY, FEBRUARY, MARCH, ..., DECEMBER };   
      
    VS.  
      
    public enum Month {  
     JANUARY(31), FEBRUARY(28),  
     MARCH(31), APRIL(30),  
     MAY(31), JUNE(30),  
     JULY(31), AUGUST(31),  
     SEPTEMBER(30), OCTOBER(31),  
     NOVEMBER(30), DECEMBER(31);  
      
     private final int daysInMonth;  
      
     // Constructor not visible to clients. Only used to initialize the constants above.  
     private Month(int daysInMonth) {  
     this.daysInMonth = daysInMonth;  
     }  
      
     // enums also have an automatic, invisible field:  
     // private final int ordinal; takes on values 0, 1, ... for each value in the enum.
  + Rich enum Example(2)
  + public int getDaysInMonth(boolean isLeapYear) {  
     if (this == FEBRUARY && isLeapYear) {  
     return daysInMonth+1;  
     } else {  
     return daysInMonth;  
     }  
    }  
      
    public Month nextSemester() {  
     switch (this) {  
     case JANUARY:  
     return FEBRUARY;  
     case FEBRUARY: case MARCH: case APRIL: case MAY:  
     return JUNE;  
     case JUNE: case JULY: case AUGUST:  
     return SEPTEMBER;  
     case SEPTEMBER: case OCTOBER:  
     case NOVEMBER: case DECEMBER:  
     return JANUARY;  
     default:  
     throw new RuntimeException("can't get here");  
     }  
    }
  + Item 34: Use Enums instead of int Constants
  + // The int enum pattern – severely deficient  
     public static final int APPLE\_FUJI = 0;  
     public static final int APPLE\_PIPPIN = 1;  
     public static final int APPLE\_GRANNY\_SMITH = 2;  
      
     public static final int ORANGE\_NAVEL = 0;  
     public static final int ORANGE\_TEMPLE = 1;  
     public static final int ORANGE\_BLOOD = 2;  
      
    // Mixing apples and oranges! Type safety violation. The compiler cannot help.  
     int i = (APPLE\_FUJI – ORANGE\_TEMPLE) / APPLE\_PIPPIN;

## Problems With Int Enum Pattern

* Total Lack of Type Safety (enum safety “Month firstMonth = MONDAY;”)
* Problematic Programs
  + Names Compiled to Constants in Client Code
  + Renumbering Requires Recompiling Clients (a No-No for the Java community!)
* Inconvenient for Printing: need to keep an array of strings indexed by the constants
* Alternative “String Enum Pattern” same problems: type safety, client code…

## Java Enum Types

* Similar to the Singleton Pattern (what is that?)
  + But exports multiple instances
* Guarantee Compile-Time Type Safety
  + Declaration of Apple Cannot Hold an Orange
* Each Enum Has its own Namespace
  + No Need To Prefix Constants With Type Name
    - The enum has to be in a package
    - The client need to use “static import”
* No Need to Recompile Clients (important for the Java community)
* Also… have rich functionality not present in classes and interfaces
* Example:
  + Rich Enum
  + public enum Planet { // Enum type with data and behavior  
     MERCURY (3.302e+23, 2.439e6),  
     VENUS (4.869e+24, 6.052e6),  
     EARTH (5.975e+24, 6.378e6),…; // plus MARS, JUPITER, etc.  
      
     private final double mass;  
     private final double radius;  
     private final double surfaceGravity;  
     private static final double G = 6.67300e-11; // Universal G  
     private Planet (double mass, double radius) { // Constructor, cannot be made public by compiler  
     this.mass = mass; this.radius = radius;  
     surfaceGravity = G\* mass / (radius \* radius);}  
      
     public double mass() { return mass; }  
     public double radius() { return radius; }  
     public double surfaceGravity() { return surfaceGravity; }  
      
     public double surfaceWeight (double mass) {   
     return mass \* this.surfaceGravity; } // F = ma  
    }
  + Using the Enum
  + public class WeightTable { // CLIENT  
     public static void main (String[] args) {  
     double earthWeight = Double.parseDouble (args[0]);  
     double mass = earthWeight / Planet.EARTH.surfaceGravity();  
     // All Enums have a static values() method  
     // All Enums have a sensible (and Overridable) toString()  
     for (Planet p : Planet.values()) // values() = array  
     System.out.printf (“Weight on %s is %f%n”, p, p.surfaceWeight(mass));  
     }   
    }  
      
    // Output:  
    Weight on MERCURY is 66.133672  
    Weight on VENUS is 158.383926  
    Weight on EARTH is 175.000000  
    ...

## More Examples

* Wrong way of Providing Different Behavior

// Enum type that switches on its own value – similar to tagged classes  
public enum Operation {  
 PLUS, MINUS, TIMES, DIVIDE;  
  
 // Do the arithmetic op represented by constant  
 double apply (double x, double y) {  
 // your-own dynamic dispatching “code smell”  
 switch (this) {   
 case PLUS: return x + y;  
 case MINUS: return x – y;  
 case TIMES: return x \* y;  
 case DIVIDE: return x / y;  
 }  
 throw new AssertionError(“Unknown op: “ + this); // what if the switch list   
 // was not complete?  
 }  
}

* Better: Constant Specific Methods

// Enum type with constant-specific method implementations  
public enum Operation {  
 PLUS { double apply (double x, double y) { return x + y; } },  
 MINUS { double apply (double x, double y) { return x - y; } },  
 TIMES { double apply (double x, double y) { return x \* y; } },  
 DIVIDE { double apply (double x, double y) { return x / y; } };  
  
 // abstract apply() ensures each constant provide definition  
 abstract double apply(double x, double y);  
}  
  
// each constant/instance has its own apply() method!!  
// …different from subtyping/overriding

* Constant Specific Methods + Data

// Enum type with constant-specific class bodies and data  
public enum Operation {  
 PLUS(“+”) { double apply (double x, double y) { return x + y; } },  
 MINUS(“-”) { double apply (double x, double y) { return x - y; } },  
 TIMES(“\*”) { double apply (double x, double y) { return x \* y; } },  
 DIVIDE(“/”){ double apply (double x, double y) { return x / y; } };  
  
 private final String symbol;  
 Operation (String symbol) { this.symbol = symbol; }  
 @Override public String toString() { return symbol; }  
  
 // abstract apply() ensures each constant provide definition  
 abstract double apply(double x, double y);  
}

## Use Instance Fields Instead of Ordinals

* Every Enum has an Associated Ordinal
  + Returns the Position of Constant
  + Don’t Use This!
    - Maintenance Nightmare
    - Brings Back the Problems With “Int Enum Pattern”
* Simple Solution
  + Use an Instance Field Instead
* Instance Fields vs. Ordinals

// Abuse of ordinal to derive an associated value – DON’T DO THIS  
public enum Ensemble {  
 SOLO, DUET, TRIO, QUARTET, QUINTET, SEXTET, SEPTET, OCTET,   
 NONET, DECTET;  
  
 public int numberOfMusicians() { return ordinal() + 1; }  
}  
// What if: you add a DOUBLE\_QUARTET? You rearrange the constants?  
  
// Good Solution: Use instance fields instead  
public enum Ensemble {  
 SOLO(1), DUET(2), TRIO(3), QUARTET(4), QUINTET(5), SEXTET(6), SEPTET(7), OCTET(8),   
 DOUBLE\_QUARTET(8), NONET(9), DECTET(10), TRIPLE\_QUARTET(12);  
  
 private final int numberOfMusicians;  
 Ensemble(int size) { this.numberOfMusicians = size; }  
 public int numberOfMusicians() { return numberOfMusicians; }  
}

## Item 36: Use EnumSet Instead of Bit Fields

// Bit field enumeration constants  
// All the disadvantages of int enum constants  
// Hard to understand when printed; No easy way to iterate through  
public class Text {  
 public static final int STYLE\_BOLD = 1 << 0; // 1   
 public static final int STYLE\_ITALIC = 1 << 1; // 2  
 public static final int STYLE\_UNDERLINE = 1 << 2; // 4   
 public static final int STYLE\_STRIKETHROUGH = 1 << 3; // 8  
  
 // Parameter is bitwise OR of zero or more STYLE\_ constants  
 public void applyStyles (int styles) {...}  
}  
text.applyStyles(STYLE\_BOLD | STYLE\_ITALIC); // 1 OR 10 = 11 = 3

* Example Use of EnumSet:

// EnumSet = Set that can only contains enums  
public class Text {  
 public enum Style {BOLD, ITALIC, UNDERLINE, STRIKETHROUGH}  
  
 // Any Set could be passed in, but EnumSet is clearly best  
 // Standard practice to pass interface instead of Class  
 public void applyStyles (Set<Style> styles) {...}   
}  
  
// Client code  
text.applyStyles(EnumSet.of(Style.BOLD, Style.Italic));

## Item 37: SKIP-Use Map Instead of Ordinal Indexing

* Problem:
  + You want to index into an array, but instead of ints, you have an enum
* Bad Solution:
  + Use ordinal() method to index into array
* Good Solution:
  + Use an EnumMap instead
* Example Class with Enumerated Type

public class Herb {  
 enum Type {ANNUAL, PERENNIAL, BIENNIAL }  
  
 final String name; // getters would be better here  
 final Type type;   
  
 public Herb(String name, Type type) {  
 this.name = name; this.type = type;  
 }  
  
 @Override public String toString() { return name; }  
}

* Example of What Not to Do

// Using ordinal() to index an array – DON’T DO THIS!  
Herb[] garden = ...;  
  
// Indexed by herb.Type.ordinal()  
Set<Herb>[]) herbsByType = (Set<Herb>[]) new Set[Herb.Type.values().length];  
for (int i= 0; i < herbsByType.length; i++)   
 herbsByType[i] = new HashSet<Herb>();  
  
for (Herb h : garden)   
 herbsByType[ h.type.ordinal() ].add(h);  
  
// Print the results  
for (int i=0; i < herbsByType.length; i++) {  
 System.out.printf(“%s: %s%n”,   
 Herb.Type.values()[i], herbsByType[i]);  
}  
// Problems: Arrays don’t play well with generics; unchecked casts;   
// label outputs by hand; ints don’t provide type-safety of enums

* Associating Data with an Enum

// Using EnumMap to assoicate data with an enum  
Map<Herb.Type, Set<Herb>> herbsByType = new EnumMap<Herb.Type, Set<Herb>> (Herb.type.class);  
  
for (Herb.type t : Herb.Type.values())  
 herbsByType.put(t, new HashSet<Herb>());  
  
for (Herb h : garden)  
 herbsByType.get(h.type).add(h);  
  
System.out.println (herbsByType);  
  
// This solution is cleaner; shorter; no unsafe cast;  
// no need to label outputs, no possibility of error in computing  
// array indices.  
  
// Note that an EnumMap is just a special kind of Map

## Item 38: SKIP-Emulate Extensible Enums with Interfaces

* Enum Types Cannot be Extended
  + public enum Sub extends Super // doesn’t compile
* Arguably, this is a good thing
  + No True Type Relation in Extensible Enums
* However, Interfaces Can Help
* // Emulate enum extension  
  // Client code uses interface I1, not Sub or Super  
  public enum Super implements I1 // compiles fine  
  public enum Sub implements I1 // share interface  
  Collection <I1> myEnums = ... // client uses Sub or Super

## Instructor Screencast: TITLE

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

Consider the following (bad) Java, implementing the "C style" enum pattern:

public class Coins {  
 public static final int PENNY = 1;  
 public static final int NICKLE = 5;  
 public static final int DIME = 10;  
 public static final int QUARTER = 25;  
}

1. Give example code that illustrates a type safety problem with Coins. Work through a range of expressions from "probably ok" to "clearly wrong".
2. What code would you need to turn a nickel into a string? Explain how this could go wrong at runtime.
3. What code would you need to iterate through the coins?
4. Would extensions to this particular enum be likely to require recompilation of client code? Explain.
5. Write a decent Java Enum for coins.
6. Turn a nickle into a string.
7. Iterate though the coins.

Consider Bloch's example:

// Abuse of ordinal to derive an associated value – DON’T DO THIS  
public enum Ensemble {  
 SOLO, DUET, TRIO, QUARTET, QUINTET,   
 SEXTET, SEPTET, OCTET, NONET, DECTET;  
  
 public int numberOfMusicians() { return ordinal() + 1; }  
}

Explain why it's wrong, fix it, and add another enum with an overlapping number of musicians.

# Assignment – (MLO 1, 2) [~2 hours]

NO ASSIGNMENT

## Purpose

## Instructions

**TO BE UPDATED LATER…**

You are provided with a basic JUnit-like tool (Sample.java and RunTest.java). Expand this implementation to support tests that take in parameters.

//Samples.java  
// Marker annotation type declaration  
  
import java.lang.annotation.\*;  
  
/\*\*  
 \* Indicates that the annotated method is a test method.  
 \* Use only on parameterless static methods  
 \*/  
@Retention (RetentionPolicy.RUNTIME)  
@Target(ElementType.METHOD)  
@interface Test {}  
  
// Program with annotations  
public class Sample {  
 @Test public static void m1() {} // Test should pass  
 public static void m2() {} // Not a @Test  
 @Test public static void m3() { // Test should fail  
 throw new RuntimeException("Boom"); }  
 public void m4() { }   
 @Test public void m5() { } // Invalid nonstatic use  
 public static void m6() { }   
 @Test public static void m7() { // fail  
 throw new RuntimeException("Crash");  
 }  
 public static void m8() { }  
}  
  
 //RunTests.java  
 // Sample code processes marker annotations – See Bloch for variations  
 import java.lang.reflect.\*;  
  
 public class RunTests {  
  
 public static void main(String[] args) throws ClassNotFoundException{  
 int tests = 0; int passed = 0;  
 Class testClass = Class.forName(args[0]);  
 for (Method m : testClass.getDeclaredMethods()) {  
 if (m.isAnnotationPresent(Test.class)) {  
 tests++;  
 try { m.invoke(null); passed++; }  
 catch (InvocationTargetException ite) {  
 System.out.println(m + " failed: " + ite.getCause()); }  
 catch (Exception e) {  
 System.out.println("Invalid @Test: " + m);   
 }  
 }   
 }  
  
 System.out.printf("Pass: %d, Fail: %d%n", passed, tests-passed);  
  
 }  
  
  
 }

## Deliverable

* Submit a .java file for your implementation.

## Due Date

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

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

NO QUIZ

## Purpose

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

## Instructions

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.