Module 10 - Common Java Contracts

# Overview and Objectives

In this module you will learn about nonfinal methods in Object such as equals, hashCode, and toString, and when and how to properly override them. We will look at common issues when attempting to extend these methods and learn standard receipes to properly override them.

## 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 common issues and required contracts associated with overriding common methods, e.g., equals
2. compare and describe common (flaw) ways of overriding common methods such as equals and hashcode
3. understand standard receipes and correct ways to override common methods

# 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 3

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

* Object is designed primarily for extension.
  + Its nonfinal methods (equals, hashCode, toString, clone, and finalize) have explicit general contracts designed to be overridden.
  + Classes overriding these methods need to obey their general contracts
  + Otherwise would prevent other classes that depend on the contracts (such as HashMap and HashSet) from functioning proerly
* This modules tells us when and how to override nonfinal methods of Object

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

## Item 10: Obey the general contract when overriding equals

* Equals implements an **equivalence** relation
  + Reflexive: For any non-null reference value x, x.equals(x) must return true.
  + Symmetric: For any non-null reference values x and y, x.equals(y) must return true if and only if y.equals(x) returns true.
  + Transitive: For any non-null reference values x, y, z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) must return true.
  + Consistent: For any non-null reference values x and y, multiple invocations of x.equals(y) must consistently return true or consistently return false, provided no information used in equals comparisons is modified.
  + For any non-null reference value x, x.equals(null) must return false.

### Example: Violating Symmetric

// Broken - violates symmetry!  
 public final class CaseInsensitiveString {  
 private final String s;  
  
 public CaseInsensitiveString(String s) {  
 this.s = Objects.requireNonNull(s);  
 }  
  
 // Broken - violates symmetry!  
 @Override public boolean equals(Object o) {  
 if (o instanceof CaseInsensitiveString)  
 return s.equalsIgnoreCase(  
 ((CaseInsensitiveString) o).s);  
 if (o instanceof String) // One-way interoperability!  
 return s.equalsIgnoreCase((String) o);  
 return false;  
 }  
 ... // Remainder omitted  
 }  
  
 // example 1  
 CaseInsensitiveString cis = new CaseInsensitiveString("Polish");  
 String s = "polish";  
 cis.equals(s); //returns true.  
 s.equals(cis); // returns false,  
  
 // example 2  
 List<CaseInsensitiveString> list = new ArrayList<>();  
 list.add(cis);  
 list.contains(s); // We don't really know -- implementation dependence.  
 //Once we violate equal contracts, we simply don't know how things would behave  
  
  
 //a fix : only deal with case when o CaseInsensitiveString  
 @Override public boolean equals(Object o) {  
 return o instanceof CaseInsensitiveString &&  
 ((CaseInsensitiveString) o).s.equalsIgnoreCase(s);  
 }  
  
\*\*\* Example: Violating Transitivity  
  
\*\*\*\* DO Group Exercise 1  
  
#+begin\_src java  
 public class Point {  
  
 private final int x;  
 private final int y;  
  
 public Point(int x, int y) {  
 this.x = x;  
 this.y = y;  
 }  
  
 @Override public boolean equals(Object o) {  
 if (!(o instanceof Point))  
 return false;  
 Point p = (Point)o;  
 return p.x == x && p.y == y;  
 }  
  
 ... // Remainder omitted  
 }  
  
 public class ColorPoint extends Point {  
 private final Color color; // new field  
  
 public ColorPoint(int x, int y, Color color) {  
 super(x, y);  
 this.color = color;  
 }  
 ... // Remainder omitted  
  
 // breaks symmetry (does not break transitivity)  
 @Override public boolean equals1(Object o) {  
 if (!(o instanceof ColorPoint))  
 return false;  
 return super.equals(o) && ((ColorPoint) o).color == color;  
 }  
  
 // does not break symmetry, breaks transitivity  
 @Override public boolean equals2(Object o) {  
 if (!(o instanceof Point))  
 return false;  
  
 //ignore color when doing mixed comparison  
 if (!(o instanceof ColorPoint)) //if o instanceof Point  
 return o.equals(this); //then call equals of Point   
  
 //o is colorpoint  
 return super.equals(o) && ((ColorPoint) o).color == color;  
  
 }  
  
  
 ColorPoint a (1,2,Blue);  
 ColorPoint b (1,2,Red);  
 Point c (1,2);  
  
 //break symmetry  
 c.equals1(a); //True  
 a.equals1(c); // False  
  
 //break transitivity  
 c.equals2(a); //True  
 a.equals2(c); // True  
  
 a.equals(c); //True  
 c.equals(b); //True  
 a.equals(b); //False ; break transitivity

* So how to fix this?
  + There is **no way** to extend an instantiable class and add a value component while preserving the equals contract, unless you’re willing to forgo the benefits of object-oriented abstraction.

@Override public boolean equals(Object o) {  
  
 if (o == null || o.getClass() != getClass())  
 return false;  
  
 Point p = (Point) o;  
 return p.x == x && p.y == y;  
}  
  
a.equals(c); //True  
c.equals(b); //True  
a.equals(b); //True  
  
  
//But this breaks LSP  
//points = {Point(1,2), Point(3,4)}  
//c1 = ColorPoint(1,2,Blue)  
//c1 should be in points (because c1 is still a point), but using this equals method, c1 is not in points because of diff types

## Item 11: Always override hashCode when you override equals

* This is because of a contract of hashCode: equal objects must have equal hash codes
* How to **not** write hashCode ?
* // The worst possible legal hashCode implementation - never use!  
  @Override public int hashCode() { return 42; }
  + what's the issue? It's legal because it ensures that equal objects have the same hash code. It’s bad because it ensures that every object has the same hash code. Therefore, every object hashes to the same bucket, and hash tables degenerate to linked lists. Programs that should run in linear time instead run in quadratic time. For large hash tables, this is the difference between working and not working.
* Receipt for writting Hashcode
* result = hash(v1)  
  result += 31 \* v1 + hash(v2)  
  result += 31 \* v1 + hash(v3)

## Item 12: Always override toString

## Item 13: Override clone judiciously

## Item 14: Consider implementing Comparable

## Instructor Screencast: TITLE

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

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

Consider Bloch's Point/ColorPoint example. For today, ignore the hashCode() issue.

public class Point { // routine code  
 private int x; private int y;  
 ...  
 @Override public boolean equals(Object obj) { // Standard recipe  
 if (!(obj instanceof Point)) return false;  
  
 Point p = (Point) obj;  
 return p.x == x && p.y == y;  
 }  
}  
  
public class ColorPoint extends Point { // First attempt: Standard recipe  
 private COLOR color;  
 ...  
 @Override public boolean equals(Object obj) {  
 if (!(obj instanceof ColorPoint)) return false;  
  
 ColorPoint cp = (ColorPoint) obj;  
 return super.equals(obj) && cp.color == color;  
 }  
}  
  
public class ColorPoint extends Point { // Second attempt: DON'T DO THIS!  
 private COLOR color;  
 ...  
 @Override public boolean equals(Object obj) {  
 if (!(o instance of Point)) return false;  
  
 // If obj is a normal Point, be colorblind  
 if (!(obj instanceof ColorPoint)) return obj.equals(this);  
  
 ColorPoint cp = (ColorPoint) obj;  
 return super.equals(obj) && cp.color == color;  
 }  
}

1. What is the equals() contract? How do you implement equal in this example?
2. Why is the instanceof operator for?
3. Write client code that shows a contract problem with the first attempt at ColorPoint (i.e., what contract does it break?)
4. Write client code that shows a contract problem with the second attempt at ColorPoint (i.e., what contract does it break?).
5. Some authors recommend solving this problem by using a different standard recipe for equals().
   * What's the key difference?
   * Which approach do you want in the following code:
   * public class CounterPoint extends Point  
      private static final AtomicInteger counter =  
      new AtomicInteger();  
       
     public CounterPoint(int x, int y) {  
      super (x, y);  
      counter.incrementAndGet();  
     }  
     public int numberCreated() { return counter.get(); }  
       
     @Override public boolean equals (Object obj) { ??? }  
     }  
       
       
     // Client code:  
       
     Point p = PointFactory.getPoint(); // either a Point or a CounterPoint  
     Set<Point> importantPoints = // a set of important points  
      boolean b = PointUtilities.isImportant(p); // value?

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

Consider a variation of Liskov's IntSet example (Figure 5.10, page 97)

public class IntSet implements Cloneable {  
 private List<Integer> els;  
 public IntSet () { els = new ArrayList<Integer>(); }  
 ...  
 @Override  
 public boolean equals(Object obj) {  
 if (!(obj instanceof IntSet)) return false;  
  
 IntSet s = (IntSet) obj;  
 return ???  
 }  
  
 @Override  
 public int hashCode() {  
 // see below  
 }  
  
 // adding a private constructor  
 private IntSet (List<Integer> list) { els = list; }  
  
 @Override  
 public IntSet clone() {  
 return new IntSet ( new ArrayList<Integer>(els));  
 }  
  
}

1. How should the equals() method be completed?
2. Analyze the following ways to implement hashCode()? If there is a problem, give a test case that shows the problem.
   1. not overridden at all
   2. return 42;
   3. return els.hashCode();
   4. int sum = 0; for (Integer i : els) sum += i.hashCode(); return sum;
3. What's the problem with clone() here (something with subtyping)? Give a test case that shows the problem.
4. Fix clone() in two very different ways.

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

## Purpose

Get familiar with common methods including equals, hashcode, and clone and provide correct ways to override them.

## Instructions

As it happens, Liskov's implementation of clone() for the IntSet class (see figure 5.10, page 97) is wrong.

1. Use the version of IntSet from the in-class exercise. Implement a subtype of IntSet to demonstrate the problem. Your solution should include appropiate executable code in the form of JUnit tests.
2. Provide a correct implementation of clone() for IntSet. Again, give appropriate JUnit tests.
3. Correctly override hashCode() and equals(). Note that the standard recipe is not appropriate in this (unusual) case (why?).

Grading Criteria: In addititon to code and tests, your deliverable is a story. Explain what is going on at each stage of the exercise. The GTA will primarily grade your story.

## Deliverable

* Submit a .java file for your implementation.

## Due Date

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

# 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

The quiz is 30 minutes in length. The quiz is closed-book.

Consider the following code.

class Apple {  
 // rep-inv: name != null  
 private String name;  
 public Apple (String name) {   
 if (name == null) throw new NPE(...);  
 this.name = name;  
 }  
 @Override public boolean equals (Object o) {  
 if (!(o instanceof Apple)) { return false; }  
 Apple a = (Apple) o;   
 return name.equals(a.name);  
 }  
 @Override public int hashCode() { // see questions below }  
 @Override public String toString() { return name; }  
}  
class AppleTracker extends Apple {  
 private static Set<String> inventory = new HashSet<String> ();  
 public AppleTracker (String name) { super(name); inventory.add(name);}  
 public static Set<String> getInventory() { return Collections.unmodifiableSet(inventory);}  
}  
// client code  
Apple a = new Apple("Winesap");  
AppleTracker at1 = new AppleTracker("Winesap");  
AppleTracker at2 = new AppleTracker("Fuji");

Mark each of the following either **True** or **False**:

1. The equals() method in the AppleTracker class is inherited from the Apple class.

* a.equals(at1) sometimes returns true and sometimes returns false.
* The equals() method in the Apple class relies on the rep-invariant to satisfy its contract.
* AppleTracker adds client-visible state to Apple objects.
* a.equals(at1) and at1.equals(a) are both true.
* a.equals(at2) and at2.equals(a) are both false.
* at1.equals(a) and a.equals(at2) are both true, but at1.equals(at2) is false.
* It would correct to implement hashCode() as return name.hashCode();
* It would correct to inherit hashCode() from the Object class.
* Bloch would object to replacing o instanceof Apple with a predicate built atop getClass().

## Deliverable

Use the link above to take the quiz.

## Due Date

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