Equality and Object Contract

+ Inheritance

+ Generics Review
Object Contract

• Every class extends Object and inherits equals method:
  – public boolean equals (Object o)

• If you decide to override equals, you must follow Object Contract
The Object Contract

• equals must define an equivalence relation
  – reflexive, symmetric, transitive

• equals must be consistent
  – repeated calls to the method must yield the same result provided no information used in equals comparisons on the object is modified

• for a non-null reference x, x.equals (null) should return false

• hashCode must produce the same result for two objects that are deemed equal by the equals method.
Hash table

• Have array initialized to size corresponding to number of elements that we expect to be inserted.
• When inserting, we compute the hashcode of the key, and convert it into an index in the array's range
• Value is then inserted at that index.
• Rep invariant: keys are in slots based on hash codes
Hashing

• Consider simple hash function:
  – Always return same value
  – Is this legal?
  – Why is this bad?

• Why do we need to always override hashCode when we override equals?
  – (hint: if you don't override hashCode at all, you'll get the one from Object, which is based on the address of the object)
Equality + Inheritance Example

```java
public class Duration {
    private final int day;
    private final int sec;
    public Duration(int Day, int sec) {
        this.day = day; this.sec = sec;
    }
}

Duration d1 = new Duration(10,5);
Duration d2 = new Duration(10,5);
System.out.println(d1.equals(d2)); //returns false
```
Adding equals method:

```java
public boolean equals(Duration d) {
    if (d==null)
        return false;
    return d.day == d.day && d.sec == sec;
}
Duration d1 = new Duration(10,5);
Duration d2 = new Duration(10,5);
System.out.println(d1.equals(d2)); //returns true

Satisfy object contract? Not quite...
Object d1 = new Duration(10,5);
Object d2 = new Duration(10,5);
System.out.println(d1.equals(d2)); //returns false

Usually want to avoid this!
```
@Override //compiler will warn if type mismatch
public boolean equals(Object o) {
    if(!(o instanceof Duration))
        return false;
    Duration d = (Duration) o;
    return d.day == day && d.sec == sec;
}
Object d1 = new Duration(10,5);
Object d2 = new Duration(10,5);
System.out.println(d1.equals(d2));
//returns true!
And add some inheritance!

public class NanoDuration extends Duration {
    private final int nano;
    public NanoDuration(int day, int sec, int nano) {
        super(day, sec);
        this.nano = nano;
    }

    //But what about the equals method?!
}
public boolean equals(Object o) {
    if(! (o instanceof NanoDuration))
        return false;
    NanoDuration nd = (NanoDuration) o;
    return super.equals(nd) && nano == nd.nano;
}

Duration d1 = new NanoDuration(5,10,15);
Duration d2 = new Duration(5,10);
System.out.println(d1.equals(d2));
//false
System.out.println(d2.equals(d1));
//true

We violate symmetry!
public boolean equals(Object o) {
    if(! (o instanceof NanoDuration))
        return super.equals(o);
    NanoDuration nd = (NanoDuration) o;
    return super.equals(nd) && nano == nd.nano;
}

Duration d1 = new NanoDuration(5,10,15);
Duration d2 = new Duration(5,10);
Duration d3 = new NanoDuration(5,10,30);
System.out.println(d1.equals(d2));
    //true
System.out.println(d2.equals(d3));
    //true
System.out.println(d1.equals(d3));
    //false

We just violated transitivity!
Checking Exact Class

• In Duration, avoid comparing to instances of subtype

```java
public boolean equals(Object o) {
    if(o==null || (!o.getClass().equals(getClass())))
        return false;
    Duration d = (Duration) o;
    return d.day == day && d.sec == sec;
}
```

• But now every subtype must override equals!
Just avoid inheritance altogether!

• Use composition!

```java
public class Nanoduration {
    private final Duration duration;
    private final int nano;
    //...
}
```
hashCode for Duration

• 1
  – Always safe, but makes hash tables completely inefficient

• day
  – Safe, but inefficient for Durations that differ in sec field only

• day+sec
  – Safe and changes in any field will tend to change code
Equality Summary

• Can be tricky when comparing superclass and subclass objects

• Possible solutions:
  – Have superclass reject equality when comparing to subclass (o.getClass().equals(getClass ()))
  – Composition: Separate the superclass from the subclass, i.e. ColorPoint doesn’t inherit from Color, instead it contains a Color in its representation

• Don’t forget to override the hashCode too!
Inheritance

• Unlike method invocation, inheritance breaks encapsulation (information hiding)
• Can be very problematic (e.g. think of equals example)!
• Review lecture notes for a good example!
Composition

• Don’t extend existing class
• Give class private field that references existing class
• Existing class is component of new one
• The good:
  – Very easy to reason about
  – No slippery self-calls
  – Implementation inheritance is not exposed
  – Example of a “wrapper” class
• The bad:
  – Can’t substitute anymore for parent class
  – Can be a bit tedious to write delegation methods
Generics Revisited

• Imagine a world without generics 😞
Consider the animal kingdom (and a zoo!)

• See
  http://download.oracle.com/javase/tutorial/java/generics/subtyping.html
General Rules

• Use Container<? extends A> when you want to read from container
• Use Container<? super A> when you want to write to container

static <T> void copyList(List<? super T> out, List<? extends T> in)
{
    for (T e: in) {
        out.add(e)
    }
}
Topics we have covered:

- Event based programming
- Generics (today!)
- Concurrency
- Equality (today!)
- Inheritance (today!)
- Performance engineering (Monday!)