CSE 331
Software Design & Implementation

Kevin Zatloukal
Summer 2016

Identity, equals, and hashCode
(Based on slides by Mike Ernst, Dan Grossman, David Notkin, Hal Perkins, Zach Tatlock)
Overview

• Notions of object equality are important in any language

• In Java, collection classes depend on `equals` and `hashCode`
  – EJ 47: “Know and use the libraries”
    • “every programmer should be familiar with the contents of java.lang and java.util”
  – e.g., TreeSet may not work properly if `equals` is wrong
  – e.g., HashSet may not work properly if `hashCode` is wrong
Object equality

A simple idea??

- Two objects are equal if they have the same value

A subtle idea: intuition can be misleading

- Same object or same contents?
- Same concrete value or same abstract value?
- Same right now or same forever?
- Same for instances of this class or also for subclasses?
- When are two collections equal?
  - How related to equality of elements? Order of elements?
  - What if a collection contains itself?
- How can we implement equality efficiently?
Expected properties of equality

Reflexive \[ \text{a.equals}(a) == \text{true} \]
- Confusing if an object does not equal itself

Symmetric \[ \text{a.equals}(b) \iff \text{b.equals}(a) \]
- Confusing if order-of-arguments matters

Transitive \[ \text{a.equals}(b) \&\& \text{b.equals}(c) \implies \text{a.equals}(c) \]
- Confusing again to violate centuries of logical reasoning
Expected properties of equality

**Reflexive**

\[ a.equals(a) == \text{true} \]

- Confusing if an object does not equal itself

**Symmetric**

\[ a.equals(b) \iff b.equals(a) \]

- Confusing if order-of-arguments matters

**Transitive**

\[ a.equals(b) \land b.equals(c) \Rightarrow a.equals(c) \]

- Confusing again to violate centuries of logical reasoning

A relation that is reflexive, transitive, and symmetric is an *equivalence relation*
Reference equality

- Reference equality means an object is equal only to itself
  - \( a == b \) only if \( a \) and \( b \) refer to (point to) the same object

- Reference equality is an equivalence relation
  - Reflexive
  - Symmetric
  - Transitive

- Reference equality is the \textit{smallest} equivalence relation on objects
  - “Hardest” to show two objects are equal (must be same object)
  - Cannot be smaller without violating reflexivity
  - Sometimes but not always what we want
What might we want?

- Sometimes want equivalence relation bigger than ==
  - Java takes OOP approach of letting classes *override* `equals`

```java
date d1 = new Date(12,27,2013);
date d2 = new Date(12,27,2013);
date d3 = d2;
// d1==d2 ?
// d2==d3 ?
// d1.equals(d2) ?
// d2.equals(d3) ?
```
Object.equals method

```java
public class Object {
    public boolean equals(Object o) {
        return this == o;
    }
    ...
}
```

- Implements reference equality
- Subclasses can override to implement a different equality
- But library includes a contract `equals` should satisfy
  - Reference equality satisfies it
  - So should any overriding implementation
  - Balances flexibility in notion-implemented and what-clients-can-assume even in presence of overriding
equals specification

public boolean equals(Object obj) should be:

• reflexive: for any reference value \( x \), \( x.equals(x) == true \)

• symmetric: for any reference values \( x \) and \( y \),
  \( x.equals(y) == y.equals(x) \)

• transitive: for any reference values \( x \), \( y \), and \( z \), if \( x.equals(y) \)
  and \( y.equals(z) \) are true, then \( x.equals(z) \) is true

• consistent: for any reference values \( x \) and \( y \), multiple
  invocations of \( x.equals(y) \) consistently return true or
  consistently return false (provided neither is mutated)

• For any non-null reference value \( x \), \( x.equals(null) \) should
  be false
Why all this?

• Remember the goal is a contract:
  – weak enough to allow different useful overrides
  – strong enough so clients can assume equal-ish things
    • example: To implement a set
    – this gives a good balance in practice

• In summary:
  – equivalence relation
  – consistency, but allow for mutation to change the answer
  – asymmetric with \texttt{null} (other way raises exception)
    • final detail: argument of \texttt{null} must return \texttt{false}
An example

A class where we may want `equals` to mean equal contents

```java
public class Duration {
    private final int min; // RI: min>=0
    private final int sec; // RI: 0<=sec<60
    public Duration(int min, int sec) {
        assert min>=0 && sec>=0 && sec<60;
        this.min = min;
        this.sec = sec;
    }
}
```

- Should be able to implement what we want and satisfy the `equals` contract...
public class Duration {
    ...
    public boolean equals(Duration d) {
        return this.min==d.min && this.sec==d.sec;
    }
}

Two bugs:
1. Violates contract for null (not that interesting)
   - Can add if(d==null) return false;
     • But our fix for the other bug will make this unnecessary
2. Does not override Object’s equals method (more interesting)
Overloading versus overriding

In Java:

− A class can have multiple methods with the same name and different parameters (number or type)
− A method overrides a superclass method only if it has the same name and exact same argument types

So `Duration`’s `boolean equals(Duration d)` does not override `Object`’s `boolean equals(Object d)`

− Sometimes useful to avoid having to make up different method names
− Sometimes confusing since the rules for what-method-gets-called are complicated
− [Overriding covered in CSE143, but not overloading]
Example: no overriding

```java
public class Duration {
    public boolean equals(Duration d) {...}
    ...
}
Duration d1 = new Duration(10,5);
Duration d2 = new Duration(10,5);
Object o1 = d1;
Object o2 = d2;
d1.equals(d2); // true
o1.equals(o2); // false(!)
d1.equals(o2); // false(!)
o1.equals(d2); // false(!)
```

CSE331 Summer 2016
public class Duration {
    public boolean equals(Object d) {...}
    ...
}
Duration d1 = new Duration(10, 5);
Duration d2 = new Duration(10, 5);
Object o1 = d1;
Object o2 = d2;
d1.equals(d2); // true
o1.equals(o2); // true [overriding]
d1.equals(o2); // true [overriding]
o1.equals(d2); // true [overriding]
d1.equals(o1); // true [overriding]
A little more generally

• Won’t go through all the overloading-resolution rules here

• In short, Java:
  – Uses (compile-time) types to pick the signature (at compile-time)
    • In example: if receiver or argument has compile-time type Object, then only signature taking an Object is “known to work,” so it is picked
  – At run-time, uses dynamic dispatch to choose what implementation with that signature runs
    • In un-fixed example: the inherited method is the only one with the take-an-Object signature
    • In fixed example: Overriding matters whenever the run-time class of the receiver is Duration
But wait!

This doesn’t actually compile:

```java
public class Duration {
    ...
    public boolean equals(Object o) {
        return this.min==o.min && this.sec==o.sec;
    }
}
```
public class Duration {
    public boolean equals(Object o) {
        if(! o instanceof Duration)
            return false;
        Duration d = (Duration) o;
        return this.min==d.min && this.sec==d.sec;
    }
}

• Cast cannot fail
• We want equals to work on any pair of objects
• Gets null case right too (null instanceof C always false)
• So: rare use of cast that is correct and idiomatic
  – This is what you should do (cf. Effective Java)
Satisfies the contract

```java
public class Duration {
    public boolean equals(Object o) {
        if(! o instanceof Duration)
            return false;
        Duration d = (Duration) o;
        return this.min==d.min && this.sec==d.sec;
    }
}
```

• Reflexive: Yes
• Symmetric: Yes, even if o is not a Duration!
  – (Assuming o’s equals method satisfies the contract)
• Transitive: Yes, similar reasoning to symmetric
Even better

• Defensive Tip: use the `@Override` annotation when overriding

```java
public class Duration {
    @Override
    public boolean equals(Object o) {
        ...
    }
}
```

• *Compiler warning* if not actually an override
  – Catches bug where argument is `Duration` or `String` or ...
  – Alerts reader to overriding
    • Concise, relevant, *checked* documentation
Where are we?

• Done:
  – Understanding the `equals` contract
  – Implementing `equals` correctly for `Duration`
    • `override Object.equals(Object)`
    • satisfying the contract [for all types of arguments]

• Alas, matters get worse for subclasses of `Duration`
  – no perfect universal solution…
  – each option involves trade-offs
Two subclasses

class CountedDuration extends Duration {
    public static numCountedDurations = 0;
    public CountedDuration(int min, int sec) {
        super(min,sec);
        ++numCountedDurations;
    }
}

class NanoDuration extends Duration {
    private final int nano;
    public NanoDuration(int min, int sec, int nano){
        super(min,sec);
        this.nano = nano;
    }
    public boolean equals(Object o) { ... }
    ...
}
CountedDuration is (probably) fine

- CountedDuration does not override equals
  - inherits Duration.equals(Object)

- Will (implicitly) treat any CountedDuration like a Duration when checking equals
  - o instanceof Duration is true if o is CountedDuration

- Any combination of Duration and CountedDuration objects can be compared
  - equal if same contents in min and sec fields
  - works because o instanceof Duration is true when o is an instance of CountedDuration
NanoDuration is (probably) not fine

• If we don’t override equals in NanoDuration, then objects with different nano fields will be equal

• Using what we have learned:

```java
@Override
public boolean equals(Object o) {
    if (!(o instanceof NanoDuration))
        return false;
    NanoDuration nd = (NanoDuration) o;
    return super.equals(nd) && nano == nd.nano;
}
```

• But we have violated the equals contract
  – Hint: Compare a Duration and a NanoDuration
The symmetry bug

```java
public boolean equals(Object o) {
    if (!(o instanceof NanoDuration))
        return false;
    NanoDuration nd = (NanoDuration) o;
    return super.equals(nd) && nano == nd.nano;
}
```

This is **not symmetric**!

```
Duration d1 = new NanoDuration(5, 10, 15);
Duration d2 = new Duration(5, 10);
d1.equals(d2);  // false
```

```
d2.equals(d1);  // true
```
Fixing symmetry

This version restores symmetry by using Duration’s `equals` if the argument is a `Duration` (and not a `NanoDuration`)

```java
public boolean equals(Object o) {
    if (!(o instanceof Duration))
        return false;
    // if o is a normal Duration, compare without nano
    if (!(o instanceof NanoDuration))
        return super.equals(o);
    NanoDuration nd = (NanoDuration) o;
    return super.equals(nd) && nano == nd.nano;
}
```

Alas, this *still* violates the `equals` contract

– Transitivity…
The transitivity bug

\[
\begin{align*}
\text{Duration } d_1 &= \text{new NanoDuration}(1, 2, 3); \\
\text{Duration } d_2 &= \text{new Duration}(1, 2); \\
\text{Duration } d_3 &= \text{new NanoDuration}(1, 2, 4); \\
d_1.\text{equals}(d_2); &\quad // \text{true} \\
d_2.\text{equals}(d_3); &\quad // \text{true} \\
d_1.\text{equals}(d_3); &\quad // \text{false!}
\end{align*}
\]
No perfect solution

- *Effective Java* says not to (re)override `equals` like this
  - (unless superclass is non-instantiable)
  - generally good advice
  - but there is one way to satisfy `equals` contract (see below)

- Two far-from-perfect approaches on next two slides:
  1. Don’t make `NanoDuration` a subclass of `Duration`
  2. Change `Duration`’s `equals` so only `Duration` objects that are not (proper) subclasses of `Duration` are equal
Option 1: avoid subclassing

Choose composition over subclassing (EJ Item 81)

– often good advice in general (we’ll discuss more later on)
– many programmers overuse subclassing

```java
public class NanoDuration {
    private final Duration duration;
    private final int nano;
    ...
}
```

Solves some problems:

– clients can choose which type of equality to use

Introduces others:

– can’t use NanoDurations where Durations are expected (since it is not a subtype)
Option 2: the getClass trick

Check if o is a Duration and not a subtype:

@Overrides
public boolean equals(Object o) { // in Duration
    if (o == null)
        return false;
    if (!o.getClass().equals(getClass()))
        return false;
    Duration d = (Duration) o;
    return d.min == min && d.sec == sec;
}

But this breaks CountedDuration!
    – subclasses do not “act like” instances of superclass because behavior of equals changes with subclasses
    – generally considered wrong to “break” subtyping like this
Subclassing summary

• Subtypes should be useable wherever the type is used
  – Liskov substitution principle

• Unresolvable tension between
  – what we want for equality:  treat subclasses differently
  – what we want for subtyping:  treat subclasses the same

• No perfect solution for all cases
  – choose whether you want subtyping or not
    • in former case, don’t override equals (make it final)
    • in latter case, can use composition instead
    • (this matches the advice in Effective Java)
  – (unfortunately) must make one choice for all subtypes
hashCode

Another method in Object:

    public int hashCode()

“Returns a hash code value for the object. This method is supported for the benefit of hashtables such as those provided by java.util.HashMap.”

Contract (again essential for correct overriding):

- **Self-consistent**: o.hashCode() is fixed (unless o is mutated)
- **Consistent with equality**:
  
  a.equals(b) implies a.hashCode() == b.hashCode()

_Want_ !a.equals(b) implies a.hashCode() != b.hashCode()

- but not actually in contract and (not true in most implementations)
Think of it as a pre-filter

• If two objects are equal, they *must* have the same hash code
  – up to implementers of `equals` and `hashCode` to satisfy this
  – if you override `equals`, you *must* override `hashCode`

• If objects have same hash code, they *may or may not* be equal
  – “usually not” leads to better performance
  – `hashCode` in `Object` tries to (but may not) give every object a different hash code

• Hash codes are usually cheap[er] to compute, so check first if you “usually expect not equal” – a pre-filter
Asides

• Hash codes are used for hash tables
  – common implementation of collection ADTs
  – see CSE332
  – libraries won’t work if your classes break relevant contracts

• Cheaper pre-filtering is a more general idea
  – Example: Are two large video files the exact same video?
    • Quick pre-filter: Are the files the same size?
Doing it

- So: we have to override `hashCode` in `Duration`
  - Must obey contract
  - Aim for non-equals objects usually having different results

- Correct but expect poor performance:
  ```java
  public int hashCode() { return 1; }
  ```

- A bit better:
  ```java
  public int hashCode() { return min; }
  ```

- Better:
  ```java
  public int hashCode() { return min ^ sec; }
  ```

- Best
  ```java
  public int hashCode() { return 60*min+sec; }
  ```
Correctness depends on `equals`

Suppose we change the spec for Duration’s `equals`:

```java
public boolean equals(Object o) {
    if (!(o instanceof Duration))
        return false;
    Duration d = (Duration) o;
    return min == d.min && sec/10 == d.sec/10;
}
```

Must update `hashCode` – why?

```java
public int hashCode() {
    return 6*min+sec/10;
}
```
Equality, mutation, and time

If two objects are equal now, will they always be equal?
- in mathematics, “yes”
- in Java, “you choose”
- Object contract doesn't specify

For immutable objects:
- abstract value never changes
- equality should be forever (even if rep changes)

For mutable objects, either:
- use reference equality (never changes)
- not forever: mutation changes abstract value hence equals

Common source of bugs: mutating an object in a data structure
Examples

StringBuilder is mutable and sticks with reference-equality:

```java
StringBuilder s1 = new StringBuilder("hello");
StringBuilder s2 = new StringBuilder("hello");
s1.equals(s1); // true
s1.equals(s2); // false
```

By contrast:

```java
Date d1 = new Date(0); // Jan 1, 1970 00:00:00 GMT
Date d2 = new Date(0);
d1.equals(d2); // true
d2.setTime(1);
d1.equals(d2); // false
```
Behavioral and observational equivalence

Two objects are “behaviorally equivalent” if there is no sequence of operations (excluding $==$) that can distinguish them.

Two objects are “observationally equivalent” if there is no sequence of observer operations that can distinguish them.

- excludes mutators and $==$
Equality and mutation

Date class implements (only) observational equality

Can violate rep invariant of a Set by mutating after insertion

```java
Set<Date> s = new HashSet<Date>();
Date d1 = new Date(0);
Date d2 = new Date(1000);
s.add(d1);
s.add(d2);
d2.setTime(0);
for (Date d : s) { // prints two of same date
    System.out.println(d);
}
```
Pitfalls of observational equivalence

Have to make do with caveats in specs:

“Note: Great care must be exercised if mutable objects are used as set elements. The behavior of a set is not specified if the value of an object is changed in a manner that affects equals comparisons while the object is an element in the set.”

Same problem applies to keys in maps

Same problem applies to mutations that change hash codes when using HashSet or HashMap

Especially hard bugs to detect!
Easy to cause when modules don’t list everything they mutate
– why we need @modifies
Another container wrinkle: self-containment

equals and hashCode on containers are recursive:

class ArrayList<E> {
    public int hashCode() {
        int code = 1;
        for (Object o : list)
            code = 31*code + (o==null ? 0 : o.hashCode());
        return code;
    }
}

This causes an infinite loop:
List<Object> lst = new ArrayList<Object>();
lst.add(lst);
lst.hashCode();
Summary

• Different notions of equality:
  – reference equality stronger than
  – behavioral equality stronger than
  – observational equality

• Java’s `equals` has an elaborate specification, but does not require any one of the above notions
  – also requires consistency with `hashCode`
  – concepts more general than Java

• Mutation and/or subtyping make things even less satisfying
  – good reason not to overuse/misuse either