CSE 331
Software Design & Implementation

Kevin Zatloukal
Spring 2020
Identity, equals, and hashCode
Overview

• Using the libraries reduces bugs in most cases
  – take advantage of code already inspected & tested

• 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., `List` may not work properly if `equals` is wrong
  – e.g., `HashSet` may not work properly if `hashCode` is wrong
hashCode

Another method in `Object`:

```java
public int hashCode()
```

“Returns a hash code value for the object. This method is supported for the benefit of hash tables 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
  – contrapositive: if they have different hash codes, then they must not be equal

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

• Contract for `hashCode` requires only
  – (self-)consistency
  – consistent with equals

• Java’s `hashCode` must be consistent with `equals`
  – if you override `equals`, you **must** override `hashCode`

• Good performance of hash tables requires that non-equal objects *usually* have different hash codes
  – does not need to be perfect
Object.equals method

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

• Correct and idiomatic Java
• Gets null case right (null instanceof C always false)
• Cast cannot fail
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!

```java
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

Duration \( d_1 = \text{new NanoDuration}(1, 2, 3); \)
Duration \( d_2 = \text{new Duration}(1, 2); \)
Duration \( d_3 = \text{new NanoDuration}(1, 2, 4); \)
d1.equals(d2);  // true
d2.equals(d3);  // true
d1.equals(d3);  // false!

<table>
<thead>
<tr>
<th>NanoDuration</th>
<th>Duration</th>
<th>NanoDuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>min 1</td>
<td>min 1</td>
<td>min 1</td>
</tr>
<tr>
<td>sec 2</td>
<td>sec 2</td>
<td>sec 2</td>
</tr>
<tr>
<td>nano 3</td>
<td></td>
<td>nano 4</td>
</tr>
</tbody>
</table>
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 less-than-perfect approaches on next two slides:
  1. Don’t make `NanoDuration` a subclass of `Duration`
     – fact that equals should be different is a hint it’s not a subtype
  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 (Effective Java)
- 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 NanoDuration where Duration are expected (since it is not a subtype)
Option 2: the `getClass` trick

Check if `o` is a `Duration` and **not** a subtype:

```java
@Overrides
public boolean equals(Object o) {
    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 \textit{should} be useable wherever the type is used
  – Liskov substitution principle

• Unresolvable tension between
  – what we want for equality: \textit{treat subclasses differently}
  – what we want for subtyping: \textit{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 still use composition instead
    • this matches the advice in \textit{Effective Java} and from us (later)
  – almost always best to avoid getClass trick
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:

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

By contrast:

    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 $\equiv$) that can distinguish them.

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

- excludes mutators and $\equiv$
Equality and mutation

The \texttt{Date} class implements (only) observational equality.

Can violate rep invariant of a \texttt{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! (Be frightened!)
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
  – requires consistency with `hashCode`
  – concepts more general than Java

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