CSE 331
Software Design & Implementation

Kevin Zatloukal
Spring 2021
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
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 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
• Cast cannot fail
• Gets null case right too (null instanceof C always false)
Overloading vs Overriding

• Methods in Java are identified by the *signature*  
  – name + argument types

• Classes can have only one method with a given signature  
  – subclass method *overrides* superclass method with its own

• Classes can have many methods with the same name  
  – e.g., `List.add(Object)` and `List.add(int, Object)`  
  – this is called *overloading*
Java Method Calls

• Signature of the method to call is chosen at compile time
  – suppose class has equals( Object ) and equals( Duration )
  – x.equals( d1 ) becomes a call to equals( Duration ), best match
  – x.equals( o1 ) becomes a call to equals( Object ), only match

• Finding the method with that signature to call happens at run time
  – Java looks in the actual class of x ( at run time )
  – if it has a method with that signature, that method is called
  – otherwise, it continues looking in the superclass ( recursively )
DEMO
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...
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

The `Date` class implements (only) observational equality.

You can violate the representation 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! (Be frightened!) Easy to cause when modules don’t list everything they `mutate`

– why we need `@modifies`
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
**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**:
  ```java
  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?
public class Duration {
    private final int min; // RI: min>=0
    private final int sec; // RI: 0<=sec<60

    @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
class Duration {
    private int min;
    private int sec;

    public Duration(int min, int sec) {
        this.min = min;
        this.sec = sec;
    }

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

class MyDuration {
    public int hashCode() {
        return 6*min+sec/10;
    }
}
```

Must update hashCode – why?