CSE 331
Software Design & Implementation

James Wilcox & Kevin Zatloukal
Fall 2022
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

• You will need to use these for HW5+

• Same concepts exist in other languages
What might we want?

- Sometimes want equivalence relation bigger than ==
  - Java takes OOP approach of letting classes *override equals*
  - (can also be defined by a Comparator)
Expected properties of equality

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

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

**Transitive**
\[ a \text{.equals}(b) \text{ and } b \text{.equals}(c) \Rightarrow a \text{.equals}(c) \]
- 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 *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
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
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 on non-null objects
  - consistency, but allow for mutation to change the answer
  - asymmetric with null (other way raises exception)
    - weird but useful
    - often see, e.g., “left”.equals(direction) – false for null
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
Overloading versus 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*
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
Java Method Calls

• Signature of the method to call is chosen at **compile time**
  – suppose class has `equals(Object)` and `equals(Duration)`
  – Java chooses “best” match to the argument’s **compile-time type**
  – if argument has type `Duration`, `equals(Duration)` is best match
  – if argument has any other type, `equals(Object)` is **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)
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(!)
d1.equals(o1); // true [using Object’s equals]
```
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]
But wait!

This doesn’t compile:

```java
public class Duration {
    ...
    public boolean equals(Object o) {
        return this.min==o.min && this.sec==o.sec;
    }
}
```
Really fixed now

```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;
    }
}
```

- 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)
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
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
```
Equality and mutation

Date class implements (only) observational equality

```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);
}
```

Violates rep invariant of a Set by mutating after insertion
Pitfalls of Mutability

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

Especially hard bugs to detect! (Be frightened!)
   – failure doesn’t show up on the line with the bug (e.g., setTime)
Easy to cause when modules don’t list everything they mutate
   – why we need @modifies
Benefits of Immutability

Seen so far:

1. No worries about **representation exposure**
   - mutable objects need copy-in & copy-out

2. No worries about **equals consistency violations**
   - (no good way to check for this at all!)

Some other languages have tools to prevent this
   - e.g., Python
   - I would include similar tools in any new language
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
  – concepts more general than Java

• Mutation and/or subtyping make things even murkier
  – more on this later…
  – good reason not to overuse/misuse either
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()
  ```

**Contrapositive**: `a.hashCode() != b.hashCode()` implies `!a.equals(b)`
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

- If hash codes are cheaper to compute, you could first check if those are the same before doing a full comparison – a pre-filter
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)
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;
    }
}

Recall: Duration example
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;
}
```