CSE 331
Software Design & Implementation

Hal Perkins

Winter 2023

Identity, `equals`, and `hashCode`

Administrivia

HW5 part 1 due Thursday night

- Specs and tests only; no graph implementation, but skeleton code & tests must compile cleanly and generate proper JavaDocs
- Trade ideas with your colleagues – but then need to pick something
- Script vs JUnit tests:
  - Prefer script tests to JUnit if either could do a particular test
    - Don't duplicate same test as a script test and also JUnit
- Test granularity and "how many":
  - "Enough" to give you high confidence things are ok when all pass
  - Each test should ideally check one new thing
    - Section worksheet from last Thur. should be useful perspective
    - (A few "kitchen sink" tests to check long sequences of operations are ok in addition, but not as the core tests)
- ☞ Commit/push work regularly as parts are done – **don't** wait until due date to commit everything all at once
  - Write useful commit messages as you go
  - Provides backup and allows you to revert changes if needed

Midterm exam

- Exam is next Tuesday, 5-6 pm (+ a bit extra if we misjudge things and need a few more minutes)
  - Gowen 201 & 301 – who goes where TBA
  - If absolutely unavoidable conflicts, mail cse331-staff[at]cs ***now*** so we can discuss
- Review session Sun. CSE2 G20 (Amazon aud.), ~1 hour
  - 2 pm or 3 pm start: preference?
  - Chance to get questions answered, not a scripted review presentation
- Topics: everything (lectures, sections, hw assignments, readings) up to and including equals/hashCode (these slides)
- Closed book, but you can bring one 5x8 notecard with whatever *handwritten* notes you want on both sides
  - We'll have lots of blank notecards and hw1/hw2 solutions to hand out at the end of class today

If something isn't right….

- If something isn't going right or you are having problems, please get in touch with the course staff as soon as you can (cse331-staff[at]cs)
  - Try not to wait until the day something is due
  - We had an awful lot of messages in the 36 hours before both hw3 and hw4 late day cutoffs and it's hard to help effectively at the last minute (and we would like to help because we know people are stressed when we get messages like that)
  - So if you're sick, or have fallen way behind for some reason, or something else is going on, don't "tough it out" and hope for the best until it's so late that it's much harder to solve the problem. Thanks.

Object equality

A simple idea??

- Two objects are equal if they have “the same value”

A subtle idea: intuition can be misleading

- Same object/reference or same contents/value?
- 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 correctly and efficiently?

Expected properties of equality

Reflexive `a.equals(a) == true`

- Confusing if an object does not equal itself

Symmetric `a.equals(b)` ⇔ `b.equals(a)`

- Confusing if order-of-arguments matters

Transitive `a.equals(b)` ∧ `b.equals(c)` ⇒ `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

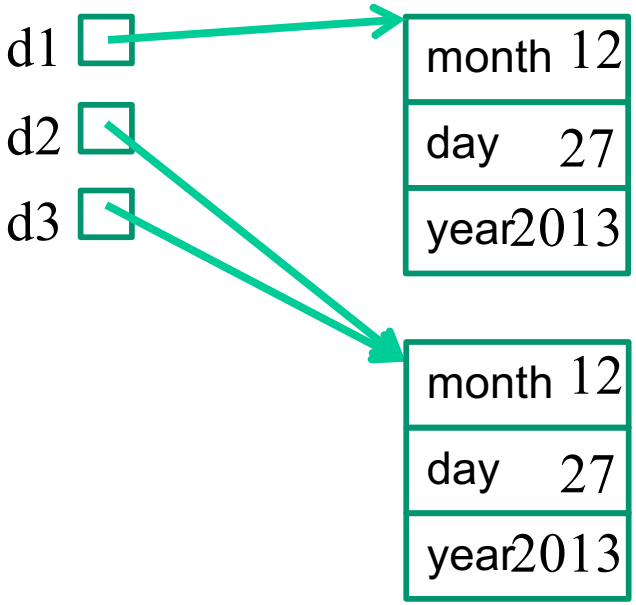
- Reference equality means an object is equal only to itself
  - `a == b` only if `a` and `b` refer to (point to) the same object

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

- Reference equality is the *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
  - The *strongest* definition of equality

What might we want?

```
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) ?
```



- Sometimes want equivalence relation bigger (weaker) than ==
  - Java lets classes *override* **`equals`**

`Object.equals` method

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

- Implements reference equality
- Subclasses can override to implement a different equality
- But Java library includes a *contract* (specification) **`equals`** should satisfy that is much more elaborate
  - 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)`

Indicates whether some other object is "equal to" this one.

The **`equals`** method implements an equivalence relation:

- It is *reflexive*: for any reference value **`x`**, **`x.equals(x)`** should return **`true`**.
- It is *symmetric*: for any reference values **`x`** and **`y`**, **`x.equals(y)`** should return **`true`** if and only if **`y.equals(x)`** returns **`true`**.
- It is *transitive*: for any reference values **`x`**, **`y`**, and **`z`**, if **`x.equals(y)`** returns **`true`** and **`y.equals(z)`** returns **`true`**, then **`x.equals(z)`** should return **`true`**.
- It is *consistent*: for any reference values **`x`** and **`y`**, multiple invocations of **`x.equals(y)`** consistently return **`true`** or consistently return **`false`**, provided no information used in equals comparisons on the object is modified.
- 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
  - Complete enough for real software

- So:
  - Equivalence relation
  - Consistency, but allow for mutation to change the answer
  - Asymmetric with **`null`** (other way raises exception)
  - Final detail: argument of **`null`** must return **`false`**

A class that needs less-strict equality

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

```
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…

How about this?

```
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 types of params)
- A method *overrides* a superclass method only if it has the *same* name and *exactly the same* argument types

So `Duration`'s `boolean equals(Duration d)` does ***not*** override `Object`'s `boolean equals(Object d)` it *overloads* it

- 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

```
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]
```

Example fixed (mostly)

```
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]
```

Java overridding a little more generally

- Won't go through all the *overloading-resolution* rules here, but…
- In short, Java:
  - Uses (compile-time) types to pick the *signature* (method name, # parameters, and their types) at compile-time based on static (declared) type of receiver plus # of parameters & types
    - 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 which class or subclass implementation with that signature to actually use
    - 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:

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

Really fixed now

```
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* #10)

Satisfies the contract

```
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

- Better style: always use `@Override` annotation when overriding

```
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

Okay, so are we done?

- Done:
  - Understanding the **`equals`** contract
  - Implementing **`equals`** correctly for **`Duration`**
    - Overriding
    - Satisfying the contract [for all types of arguments]

- Alas, matters can get worse for subclasses of **`Duration`**
  - No perfect solution, so understand the trade-offs…

- But first, we need to look at a closely related function: **`hashCode`**

`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 (also essential for correct overriding):

- Self-consistent:

  `o.hashCode() == o.hashCode()`

  ...so long as `o` doesn't change between the calls

- Consistent with equality:

  `a.equals(b)` $\Rightarrow$ `a.hashCode() == b.hashCode()`

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 two objects have the same hash code, they *might or might 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 can check that first if you "usually expect not equal" – a pre-filter

Asides

- Hash codes are used for hash tables (and hash sets, …)
  - A common collection implementation
  - See CSE332 (and most versions of CSE333!)
  - Libraries won't work if your classes break relevant **`equals`** / **`hashCode`** 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?

`hashCode` implementations

- 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:

```
public int hashCode() { return 1; }
```

- Correct but expect better-but-still-possibly-poor performance:

```
public int hashCode() { return min; }
```

- Better (changes in either field will likely change **`hashCode`**):

```
public int hashCode() { return min ^ sec; }
```

(**`^`** operator is exclusive-or – mashes bits of **`min`** and **`sec`** together)

Correctness depends on `equals`

Suppose we change the spec for **`Duration`**'s **`equals`** (and change rep inv. so it no longer requires **`sec`** < 60.):

```
// true if o and this represent same # of seconds
public boolean equals(Object o) {
  if (! (o instanceof Duration))
    return false;
  Duration d = (Duration) o;
  return 60*min+sec == 60*d.min+d.sec;
}
```

May need to update **`hashCode`** – why? (Always check!)

- This works:

```
public int hashCode() {
  return 60*min+sec;
}
```

Okay, so are we really done now?

- Done:
  - Understanding the **`equals`** contract
  - Implementing **`equals`** correctly for **`Duration`**
    - Overriding
    - Satisfying the contract [for all types of arguments]
  - **`hashCode`** and its relation to **`equals`**

- Alas, matters can get worse for subclasses of **`Duration`**
  - No perfect solution (even without worrying about **`hashCode`**), so understand the 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 good

- **`CountedDuration`** does not override **`equals`**
- Will (implicitly) treat any **`CountedDuration`** like a **`Duration`** when checking **`equals`**
- 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`**

Now `NanoDuration` [not so good!]

- If we don't override **`equals`** in **`NanoDuration`**, then objects with different **`nano`** fields will be equal

- So using everything we have learned:

```
@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

```
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 - uses NanoDuration.equals
d2.equals(d1); // true - uses Duration.equals
```

Fixing symmetry in NanoDuration

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

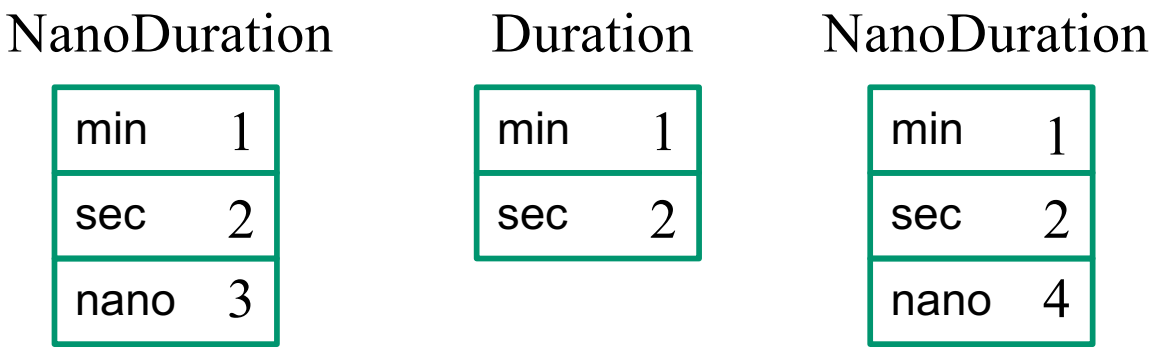
```
public boolean equals(Object o) {
  if (! (o instanceof Duration))
    return false;
  // if o is a normal Duration, compare without nano
  if (! (o instanceof NanoDuration))
    return super.equals(o);
  NanoDuration nd = (NanoDuration) o;
  return super.equals(nd) && nano == nd.nano;
}
```

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

- Transitivity…

The transitivity bug

```
Duration d1 = new NanoDuration(1, 2, 3);
Duration d2 = new Duration(1, 2);
Duration d3 = new NanoDuration(1, 2, 4);
d1.equals(d2);  // true
d2.equals(d3);  // true
d1.equals(d3);  // false!
```



No great solution

- *Effective Java* says not to (re)override **`equals`** like this
  - Unless superclass is non-instantiable (e.g., abstract)
  - "Don't do it" a non-solution given the equality we want for **`NanoDuration`** objects

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

Avoid subclassing

Choose composition over subclassing

- Often good advice: many programmers overuse (abuse) subclassing [see future lecture on proper subtyping]

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

`NanoDuration` and **`Duration`** now unrelated

- No presumption they can be compared to one another

Solves some problems, introduces others

- Can't use **`NanoDuration`**s where **`Duration`**s are expected (not a subtype)
- No inheritance, so need explicit *forwarding* methods

Slight alternative

- Can avoid some method redefinition by having **`Duration`** and **`NanoDuration`** both extend a common abstract class
  - Or implement the same interface
  - Leave overriding **`equals`** to the two subclasses

- Keeps **`NanoDuration`** and **`Duration`** from being used "like each other"

- But requires advance planning or willingness to change **`Duration`** when you discover the need for **`NanoDuration`**

The `getClass` trick

Different run-time class checking to satisfy the **`equals`** contract:

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

But now **`Duration`** objects never equal **`CountedDuration`** objects

- 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

- Due to subtleties, no perfect solution to how to design and implement **`NanoDuration`**

- Unresolvable tension between
  - "What we want for equality"
  - "What we want for subtyping"

- And: We need to be sure to have properly defined **`hashCode`** methods to satisfy the contracts related to **`equals`**

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:

- Stick with reference equality
- "No" equality is not forever – compare abstract fields
  - Mutation changes abstract value, hence what-object-equals

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` is mutable and takes the "abstract value" approach:

```
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

- This is "eternal" equality
- Two **`String`**s with the same content are behaviorally equiv.

Two objects are "observationally equivalent" if there is no sequence of *observer* operations that can distinguish them

- Excludes mutators (and ==)
- Two **`String`**s, **`Date`**s, or **`StringBuffer`**s with the same content are observationally equivalent

Equality and mutation

`Date` class implements (only) observational equality

Can therefore violate rep invariant of a `Set` by mutating after insertion

```
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

Equality for set elements would ideally be behavioral
Java makes no such guarantee (or requirement)

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`**

(Libraries choose not to copy-in for performance and to preserve object identity)

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 client code causes an infinite loop:

```
List<Object> lst = new ArrayList<Object>();
lst.add(lst);
lst.hashCode();
```

Summary: not all equals are equal

- 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 of the above notions
  - But does require consistency with **`hashCode`**

- The concepts are more general than Java

- Mutation and/or subtyping make things even less satisfying
  - Good reason not to overuse/misuse either