Equality

CSE 331: Software Design and Implementation, Autumn 2026

Your job is to deliver code that you know works.

TODO

- Homework 3: due Friday, November 6, 11:59pm
- Section Tuesday: equality and generics
- Mock midterm: Tuesday, November 10, 4:30 to 6:20pm
- Midterm: Friday, November 13, in lecture

The same queue twice

```
24  HelpQueue q1 = new FasterQueue();
25  q1.join("ada");
26  q1.join("bob");
27  q1.join("eve");
28  q1.next();
29
30  // A LinkedQueue: bob and eve join.
31  HelpQueue q2 = new LinkedQueue();
32  q2.join("bob");
33  q2.join("eve");
```

```
35  // Who is waiting in each?
36  System.out.println("q1.waiting(): " + q1.waiting());
37  System.out.println("q2.waiting(): " + q2.waiting());
38
39  // Are the two queues the same?
40  System.out.println("q1 == q2: " + (q1 == q2));
41  System.out.println("q1.equals(q2): " + q1.equals(q2));
```

- q1 is a FasterQueue, q2 a LinkedQueue: both hold bob, eve
- What do the four lines print? Write them down

The run

```
$ java Queues
q1.waiting(): [bob, eve]
q2.waiting(): [bob, eve]
q1 == q2: false
q1.equals(q2): false
```

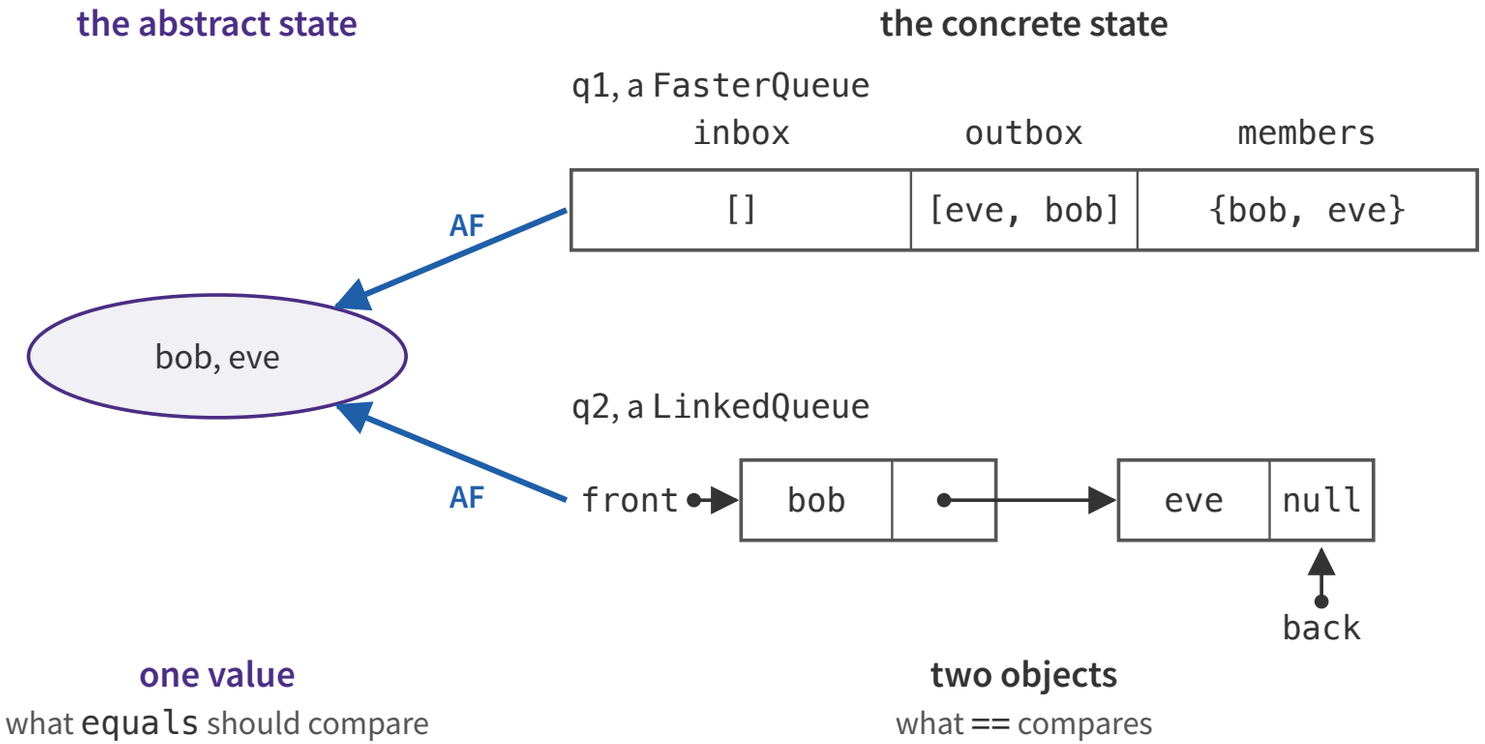
- The same students, front first: to a client, one queue
- `==`: two `new` expressions, two objects. Not our question
- `equals` printed `false` too. What did it compare?

What `equals` compared

```
public boolean equals(Object obj) {
    return (this == obj);
}
```

- `HelpQueue` specifies no `equals`; neither class declares one
- So each inherits `Object`'s, the class every class extends
- `Object`'s `equals` is `==`: the one check `Object` can make
- Nobody has said what `equals` should mean for a help queue

Two objects, one abstract value



- Different fields, and both stand for the queue bob, eve
- For any abstract data type: equal means the same abstract value

Equality for an abstract data type

Definition: equality for an abstract data type

Two objects of an abstract data type are **equal** when they stand for the same abstract value.

`equals` is the method that decides it. Like every method of the type, its specification is about the abstract state, the value an object stands for, and its body works on the fields.

- `Object` cannot know what a class stands for
- A window stands for itself: `==` is its `equals`
- A help queue stands for a value: the class writes its own
  - Unless its objects can change: a case for the end

`Object`'s specification of `equals`

- *Reflexive*: `x.equals(x)` is `true`
- *Symmetric*: `x.equals(y)` if and only if `y.equals(x)`
- *Transitive*: `x.equals(y)` and `y.equals(z)` give `x.equals(z)`
- *Consistent*: the same answer, while nothing `equals` uses changes
- `x.equals(null)` is `false`
- Every class is a subtype of `Object`: every `equals` has to keep these
- The first three: an **equivalence relation**, like `=` on numbers
- All five come free when `equals` asks for the same abstract value

`Duration`: one class, one number

```
66  public static void main(String[] args) {
67      Duration d1 = new Duration(1, 39);
68      Duration d2 = new Duration(0, 99);
69      Object o = d2;
70      Set<Duration> set = new HashSet<Duration>();
71      set.add(d1);
72      System.out.println("d1 == d2: " + (d1 == d2));
73      System.out.println("d1.equals(d2): " + d1.equals(d2));
74      System.out.println("d1.equals(o): " + d1.equals(o));
75      System.out.println("set.contains(d2): " + set.contains(d2));
76  }
```

- The RI lets seconds run past 59: 1:39 and 0:99 are one span
- Four questions, four lines: write down what each prints

Run 1: no `equals` yet

```
$ java Duration
d1 == d2: false
d1.equals(d2): false
d1.equals(o): false
set.contains(d2): false
```

- `==` and `Object`'s `equals`: one object? No, two
- `o` is `d2` under a second name: the same question
- A `HashSet` decides with `equals`, and with `hashCode`, later

A first try: compare lengths

```java
/** @return whether d is the same length as this */
public boolean equals(Duration d) {
    return seconds() == d.seconds();
}
```

```
$ java Duration
d1 == d2: false
d1.equals(d2): true
d1.equals(o): false
set.contains(d2): false
```

- `d1.equals(d2)` says `true`: lengths compared
- `o` is `d2`, and `d1.equals(o)` says `false`!
- The set still cannot find `d2`. The third line is the clue

Half sheet, question 1 (3 minutes)

```
$ java Duration
d1 == d2: false
d1.equals(d2): true
d1.equals(o): false
set.contains(d2): false
```

- The first try: `equals(Duration d)`, and no other `equals` declared
- `d1.equals(d2)` printed `true`, and `d1.equals(o)` printed `false`
  - `o` is a second name for `d2`, declared an `Object`
- Which `equals` ran in each call, and why?

An overload is not an override

```
$ javac Duration.java
Duration.java:46: error: method does not override or implement
    a method from a supertype
    @Override
    ^
1 error
```

- Java tells methods apart by **name and parameter types**
- `equals(Duration)` is a second `equals`: an **overload**
- A call picks one by the **declared type** of its argument
  - `d2` is a Duration: ours. `o` is an Object: `Object`'s
- `HashSet` calls `equals(Object)`, so ours never ran

The override

```
52  /** @return whether o is a Duration of the same length as this */
53  @Override
54  public boolean equals(Object o) {
55      if (o instanceof Duration d) {
56          return seconds() == d.seconds();
57      }
58      return false;
59  }
```

```
$ java Duration
d1 == d2: false
d1.equals(d2): true
d1.equals(o): true
set.contains(d2): false
```

- `instanceof Duration d`: `null` fails it; `d` is `o`, as a Duration
- `seconds()`, not the fields: an `int` that stands in for the abstract value
- The two calls agree now: an override runs whatever the declared type
- The set still says no, and never got as far as `equals`

Hash codes

- A `HashSet` files each element by its **hash code**, `hashCode()`
- `contains` looks only in the argument's bucket: `d1` and `d2` differ

`Object`'s documentation: equal objects must have equal hash codes

```
61   @Override
62   public int hashCode() {
63       return seconds();
64   }
```

```
$ java Duration
d1 == d2: false
d1.equals(d2): true
d1.equals(o): true
set.contains(d2): true
```

- Any number computed from the abstract value alone keeps the rule

Rule: `equals` and `hashCode` together

Rule: equals and hashCode together

Write `equals(Object)` to compare abstract values, and write `hashCode` from the same abstract value, in the same class.

An `equals` whose parameter is anything but an `Object` is an overload, which `@Override` catches. An `equals` without a `hashCode` is a mistake that `javac` accepts: the program runs, and a `HashSet` loses track of things.

- `Duration` has no subclass so far. Let's give it one

`NanoDuration`, as a subclass

```
/** @return whether o is a NanoDuration of the same length as this */
@Override
public boolean equals(Object o) {
    if (o instanceof NanoDuration n) {
        return nanos() == n.nanos();
    }
    return false;
}
```

- A Duration plus nanoseconds; `equals` written from its value
- `d` is 5:10, `nd` is 5:10 and 15 ns: predict both lines

The run

```
$ java NanoDuration
d.equals(nd): true
nd.equals(d): false
```

- The two disagree about one pair of objects

Half sheet, question 2 (3 minutes)

```java
/** @return whether o is a NanoDuration of the same length as this */
@Override
public boolean equals(Object o) {
    if (o instanceof NanoDuration n) {
        return nanos() == n.nanos();
    }
    return false;
}
```

- `NanoDuration extends Duration`, with this `equals`
- `d.equals(nd)` printed `true`, and `nd.equals(d)` printed `false`
- Which of `Object`'s promises is broken, and who made it?
- What does *declare only behavioral subtypes* say to do instead?

Symmetry breaks

```
$ java NanoDuration
d.equals(nd): true
nd.equals(d): false
```

- `Duration`'s asks `instanceof Duration`: `nd` is one, same seconds
- `NanoDuration`'s asks `instanceof NanoDuration`: `d` is not
- The object's class picks the override, by dynamic dispatch
  - The argument's declared type picked the overload, at compile time
- Symmetry, which `Object` promised, is broken

The fix most people reach for

```java
/**
 * @return whether o is a NanoDuration of the same length as this,
 *     or a Duration of the same length in whole seconds
 */
@Override
public boolean equals(Object o) {
    if (o instanceof NanoDuration n) {
        return nanos() == n.nanos();
    }
    return super.equals(o);
}
```

```
$ java NanoDuration
d.equals(nd1): true
nd1.equals(d): true
d.equals(nd2): true
nd1.equals(nd2): false
```

- nd1 is 5:10 and 15 ns, d is 5:10, nd2 is 5:10 and 30 ns
- Symmetry is back, and transitivity is gone

No answer exists

- A `Duration` stands for a span of whole seconds: `d` is 5:10
- A `NanoDuration` stands for a span of nanoseconds: `nd1`, `nd2`
- Suppose `d` and `nd1` were the same value
  - Then `d` and `nd2` are too: `Duration`'s `equals` goes by whole seconds
  - Then `nd1` and `nd2` are, by transitivity. They differ by 15 ns
- No `equals` keeps all five promises while it does both:
  - calls `nd1` equal to `d`, and tells `nd1` from `nd2`
- **Not a behavioral subtype**: do not write `extends Duration`

The way out: hold a `Duration`

```
47  /** @return whether o is a NanoDuration of the same length as this */
48  @Override
49  public boolean equals(Object o) {
50      if (o instanceof NanoDuration n) {
51          return nanos() == n.nanos();
52      }
53      return false;
54  }
```

```
$ java NanoDuration
d.equals(nd): false
nd.equals(d): false
```

- **Composition**, as Square held a Rectangle: a `Duration` in a field
- Both say no: `Duration`'s `equals` stays as it was, as it usually must
- `getClass()`, an object's exact class, would keep it. We do not use it

Back to the queues

`List`'s documentation: equal lists "contain the same elements in the same order"

```
81  /**
82   * @return whether o is a HelpQueue with the same students
83   *     waiting, in the same order
84   */
85  boolean equals(Object o);
86
87  /** @return the hash code of the list waiting returns */
88  int hashCode();
```

- A LinkedQueue and a FasterQueue holding bob, eve should be equal
- The promise goes in `HelpQueue`'s specification, for every class

Both classes write them

```
156  @Override
157  public boolean equals(Object o) {
158      if (o instanceof HelpQueue q) {
159          return waiting().equals(q.waiting());
160      }
161      return false;
162  }
163
164  @Override
165  public int hashCode() {
166      return waiting().hashCode();
167  }
```

- `instanceof HelpQueue`: the specification says any HelpQueue
- `waiting()` stands in for the abstract value; `List` compares it in order
- `FasterQueue`'s two methods are the same, line for line

The run

```
36        // Does each say it equals the other?
37        System.out.println("q1.equals(q2): " + q1.equals(q2));
38        System.out.println("q2.equals(q1): " + q2.equals(q1));
39
40        // Does a set holding q2 find q1?
41        Set<HelpQueue> set = new HashSet<HelpQueue>();
42        set.add(q2);
43        System.out.println("set.contains(q1): " + set.contains(q1));
44
45        // Then dan joins q2. Are the two still equal, and does the
46        // set still find q2?
47        q2.join("dan");
48        System.out.println("q1.equals(q2): " + q1.equals(q2));
49        System.out.println("set.contains(q2): " + set.contains(q2));
50    }
```

```
$ java EqualQueues
q1.equals(q2): true
q2.equals(q1): true
set.contains(q1): true
q1.equals(q2): false
set.contains(q2): false
```

- The first three lines: equal across two classes, and the set finds q1

One class forgets

```
$ java EqualQueues
q1.equals(q2): true
q2.equals(q1): false
```

- A copy: LinkedQueue keeps `Object`'s methods. `javac` accepts it
- NanoDuration's symmetry break, between two implementations
- A client keeping queues in a `HashSet` can now be wrong
- The promise rests on **every** implementation, written or not yet

A queue that changes

- `q2.join("dan")`: `q1.equals(q2)` is `false`, rightly
- `set.contains(q2)` is `false`, and `q2` is the set's only element!
  - `add` filed `q2` under the hash code of bob, eve
  - dan joined; the set was not told, and does not ask again
  - `contains` looks under the hash code of bob, eve, dan: nothing

Set's documentation: "Great care must be exercised if mutable objects are used as set elements."

The cost, and the ways around it

- An `equals` from the abstract value costs this on a class that changes
- `Duration` and `String` never change: a set never loses track of one
- `StringBuilder` changes, and keeps `Object`'s `equals`
- `List` took neither way, and every set of lists carries the warning
- A help queue took `List`'s choice, so `Set`'s warning applies to it
  - A queue that sits in a `HashSet` must not change while it is there

Conclusion

- **When are two objects of an abstract data type equal?**
- **When they stand for the same abstract value.**
  - `equals` and `hashCode` together, from that value
  - `equals(Object)` with `@Override`: an overload is another method
  - A finer value is not added by extending: it gets its own class
  - Implementations of one type agree through its specification
  - A set loses track of a value that changes inside it

Coming up

- **Friday**: generics
- **Section Tuesday**: equality and generics
- Mock midterm: Tuesday, November 10, 4:30 to 6:20pm
- Midterm: Friday, November 13, in lecture
- Homework 3: due Friday, November 6, 11:59pm
- Hand in your half sheet at the door