Programming: object equality

The basic intuition is simple: two objects are equal if they are indistinguishable (have the same value).

But our intuitions are incomplete in subtle ways:
- Must the objects be the same object or “just” indistinguishable?
- What is an object’s value? How do we interpret “the bits”?
- What does it mean for two collections of objects to be equal?
  - Does each need to hold the same objects? In the same order? What if a collection contains itself?
  - Who decides? The programming language designer? You?
- If a program uses inheritance, does equality change?
- Is equality always an efficient operation? Is equality temporary or forever?

Equality: hard in reality as well

- Using DNA, which of two identical twins committed a crime?
- “My grandfather’s axe”: after repeatedly replacing an axe’s head and handle, is it still the same axe?
- If you are flying next to someone on an airplane, are you on the same flight? The same airline?
- And then there are really hard questions like social equality, gender equality, race equality, equal opportunity, etc!

Properties of equality:

- Reflexive: $a.equals(a)$
  - $3 = 3$ would be confusing
- Symmetric: $a.equals(b) \iff b.equals(a)$
  - $3 = 4$ and $4 \neq 3$ would be confusing
- Transitive: $a.equals(b) \land b.equals(c) \Rightarrow a.equals(c)
  - $((1+2) = 3 \land 3 = (5-2)) \land ((1+2) \neq (5-2))$ would be confusing

A relation that is reflexive, transitive, and symmetric is an equivalence relation.

Reference equality

- The simplest and strongest (most restrictive) definition is reference equality.
- $a == b$ if and only if $a$ and $b$ refer (point) to the same object.
- Easy to show that this definition ensures $==$ is an equivalence relation.

Object.equals method

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

- This implements reference equality.
- What about the specification of Object.equals?
  - It’s a bit more complicated...

Duration $d1 = new Duration(5,3)$
Duration $d2 = new Duration(5,3)$
Duration $d3 = p2$;

// T/F: $d1 == d2$?
// T/F: $d1 == d3$?
// T/F: $d2 == d3$?
// T/F: $d1.equals(d2)$?
// T/F: $d2.equals(d3)$?
public boolean equals(Object obj)
Indicates whether some other object is "equal to" this one. The
equals method implements an equivalence relation:

- 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.

The equals method for class Object implements the most
discriminating possible equivalence relation on objects; that is,
for any reference values x and y, this method returns true if
and only if x and y refer to the same object (x==y has the
value true).

Parameters & Returns & See Also

The Object contract

- Why complicated? Because the Object class is
designed for inheritance
- Its specification will apply to all subtypes — that is, all
Java subclasses — so its specification must be flexible
  - If a.equals(b) were specified to test a == b, then no
class could change this and still be a subtype of Object
- Instead the specification gives the basic properties that
dients can rely on it to have in all subtypes of Object
- Object's implementation of equals as a == b
satisfies these properties but the specification is more
flexible

Comparing objects less strictly

public class Duration {
  private final int min;
  private final int sec;
  public Duration(int min, int sec) {
    this.min = min;
    this.sec = sec;
  }
  Duration d1 = new Duration(10,5);
  Duration d2 = new Duration(10,5);
  System.out.println(d1.equals(d2));  // False!
}

An obvious improvement

public boolean equals(Duration d) {
  return d.min == min && d.sec == sec;
}

- This defines an equivalence relation for Duration
objects (proof by partial example and handwaving)
- Duration d1 = new Duration(10,5);
- Duration d2 = new Duration(10,5);
- System.out.println(d1.equals(d2));  // False!

- Duration objects now have to be compared as Durations (or as
  Object, but not as a mixture)

overloading

- Defining equals in the Duration class creates a
  method that is invoked upon executing
d.equals(...) where d is a declared instance of
Duration
- This co-exists with equals in the Object class that
  is invoked upon executing o.equals(...) where o is
  a declared instance of Object — even if it refers
to an instance of Duration
- This gives two equals methods that can be invoked
  on instances of Duration with different results

@Override equals in Duration

@Override  // compiler warning if type mismatch
public boolean equals(Object o) {
  if (!(o instanceof Duration))  // Parameter must also be
    return false;                  //    a Duration instance
  Duration d = (Duration) o;    // cast to treat o as
    return d.min == min && d.sec == sec;
}

Object d1 = new Duration(10,5);
Object d2 = new Duration(10,5);
System.out.println(d1.equals(d2));  // True
Equality and inheritance

- Add a nanosecond field for fractional seconds
  ```java
  public class NanoDuration extends Duration {
      private final int nano;
      public NanoDuration(int min, int sec, int nano) {
          super(min, sec);
          this.nano = nano;
      }
  }
  ```

- Inheriting `equals()` from `Duration` ignores `nano`, so `Duration` instances with different `nanos` will be equal.

equals: account for nano

```java
public boolean equals(Object o) {
    if (! (o instanceof NanoDuration))
        return false;
    NanoDuration nd = (NanoDuration) o;
    return super.equals(nd) && nano == nd.nano;
}
```

But this is not symmetric! Oops!

Duration d1 = new NanoDuration(5,10,15);
Duration d2 = new Duration(5,10);
System.out.println(d1.equals(d2)); // false
System.out.println(d2.equals(d1)); // true

Fix in `Duration`

```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 is not transitive! Oops!

Duration d1 = new NanoDuration(5,10,15);
Duration d2 = new NanoDuration(5,10);
Duration d3 = new NanoDuration(5,10,30);
System.out.println(d1.equals(d2)); // true
System.out.println(d2.equals(d3)); // true
System.out.println(d1.equals(d3)); // false!

Let's get symmetry

```java
public boolean equals(Object o) {
    if (! (o instanceof NanoDuration))
        return false;
    if (! o.getClass().equals(getClass()))
        return false;
    NanoDuration nd = (NanoDuration) o;
    return super.equals(nd) && nano == nd.nano;
}
```

But this is not symmetric!

Duration d1 = new NanoDuration(5,10,15);
Duration d2 = new Duration(5,10);
System.out.println(d1.equals(d2)); // false
System.out.println(d2.equals(d1)); // true

Oops!

Check exact class instead of `instanceOf`

Equivalent change in `NanoDuration`

General issues

- Every subtype must override `equals` — even if it wants the identical definition
- Take care when comparing subtypes to one another
  - On your own: Consider an `ArithmeticDuration` class that adds operators but no new fields

Another solution: avoid inheritance

- Use composition instead
  ```java
  public class NanoDuration {
      private final Duration duration;
      private final int nano;
      // ...
  }
  ```

- Now instances of `NanoDuration` and of `Duration` are unrelated — there is no presumption that they can be `equal` or `unequal` or even compared to one another...
- Solves some problems, introduces others — for example, can't use `NanoDuration`s where `Duration`s are expected (because one is not a subtype of the other)
Efficiency of equality

- Equality tests can be slow: Are two objects with millions of sub-objects equal? Are two video files equal?
- It is often useful to quickly pre-filter – for example, if (video1.length() != video2.length()) return false; else do full equality check.
- Java requires each class to define a standard pre-filter – a `hashCode()` method that produces a single hash value (a 32-bit signed integer) from an instance of the class.
- If two objects have different hash codes, they are guaranteed to be different.
- If they have the same hash code, they may be equal objects and should be checked in full. 
  Unless you define `hashCode()` improperly!!!

Duration `hashCode` implementations

```java
public int hashCode() {
    return 1;          // always safe, no pre-filtering
}
public int hashCode() {
    return min;        // safe, inefficient for Durations differing only in sec field
}
public int hashCode() {  
    return min+sec;    // safe and efficient
}
public int hashCode() {  
    return new Random().nextInt(50000); // danger! danger!
}
```

Equality, mutation, and time

- If two objects are equal now, will they always be equal?
  - In mathematics, “yes.”
  - In Java, “you choose” – the `Object` contract doesn’t specify this (but why not?)
  - For immutable objects, equality is inherently forever.
    - The object’s abstract value never changes (much more on “abstract value” in the ADT lectures) – very roughly, these are the values the client of a class uses (not the representation used internally).
  - For mutable objects, equality can either:
    - Compare abstract values field-by-field or
    - Be eternal (how can a class with mutable instances have eternal equality?)
    - But not both.

Next steps

- Assignment 1
  - Due Friday 11:59PM
- Assignment 2
  - Out Friday
  - Due in two parts, see calendar
- Lectures
  - Abstract data types (F, M)