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 (pt 2) – HW7

• 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)

```java
Date d1 = new Date(12,10,2021);
Date d2 = new Date(12,10,2021);
Date d3 = d2;
// d1==d2 ?
// d2==d3 ?
// d1.equals(d2) ?
// d2.equals(d3) ?
```
Expected properties of equality

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

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

**Transitive**
\[ a \text{.equals}(b) \&\& b \text{.equals}(c) \Rightarrow a \text{.equals}(c) \]
- 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)
    - final detail: argument of null must return false
    - 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...
How about this?

```java
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

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
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]
```
Example fixed (mostly)

```java
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]
```

// true
// true [overriding]
// true [overriding]
// true [overriding]
But wait!

This doesn’t actually compile:

```java
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)
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