
CSE 331

Software Design & Implementation

Hal Perkins

Spring 2012

==, equals(), and all that

(Slides by David Notkin and Mike Ernst)

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?

Properties of equality

for any useful notion of equality

- *Reflexive* $a.\text{equals}(a)$
 $3 \neq 3$ would be confusing
- *Symmetric* $a.\text{equals}(b) \Leftrightarrow b.\text{equals}(a)$
 $3 = 4 \wedge 4 \neq 3$ would be confusing
- *Transitive* $a.\text{equals}(b) \wedge b.\text{equals}(c)$
 $\Rightarrow a.\text{equals}(c)$
 $((1+2) = 3 \wedge 3 = (5-2)) \wedge$
 $((1+2) \neq (5-2))$ would be confusing

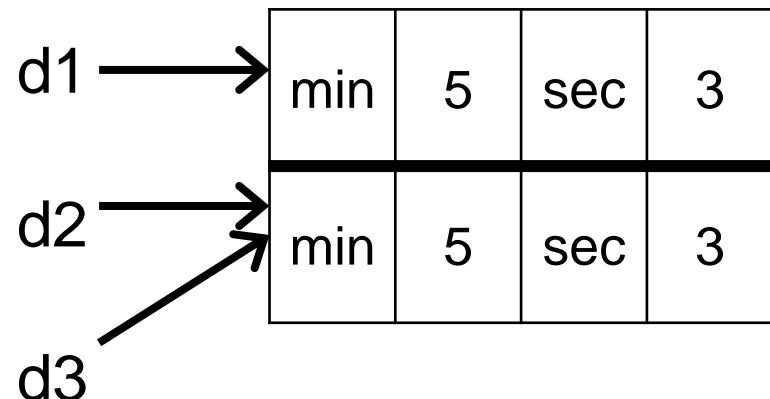
A relation that is reflexive, transitive, and symmetric is an *equivalence relation*

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

```
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) ?
```



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...

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.

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:

obj - the reference object with which to compare.

Returns:

true if this object is the same as the *obj* argument; false otherwise.

See Also:

`hashCode()`, `HashMap`

The `Object` contract

- Why so complicated?
- `Object` class is designed for inheritance
- Its specification will apply to all subtypes
 - In other words, all Java classes
- So, its specification must be flexible
 - Specification for `equals` cannot later be weakened
 - If `a.equals(b)` were specified to test `a==b`, then no class could change this and still be a true subtype of `Object`
 - Instead spec for `equals` enumerates basic properties that clients can rely on it to have in subtypes of `Object`
 - `a==b` is compatible with these properties, but so are other tests

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 – but
we likely
prefer it to
be **true**

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));
```

```
Object o1 = new Duration(10,5);  
Object o2 = new Duration(10,5);  
System.out.println(o1.equals(o2)); // False!
```

But oops

Overloading

- We have two `equals` methods:
 - `equals (Object)` in class `Object`
 - `equals (Duration)` in class `Duration`
- The one in `Duration` does **not** override the inherited one
 - it overloads it (different parameter type)
- If `d` has type `Duration`, `d.equals (Duration)` invokes the method in `Duration`
- If `o` has type `Object`, `o.equals (Duration)` invokes the `equals (Object)` method declared in `Object`
 - **Even if** the dynamic type of `o` is `Duration`!
 - `Object` does not have an `equals (Duration)` method. Method types are resolved using static types.
 - Dynamic types are used to select appropriate method at runtime (dynamic dispatch), but selected from possible methods with the correct static type.

@Override equals in Duration

```
@Override          // compiler warning if type mismatch
public boolean equals(Object o) {
    if (! (o instanceof Duration))    // Not equal if parameter
        return false;                // is not a Duration
    Duration d = (Duration) o;        // cast to treat o as
                                      // a Duration
    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
```

- ❑ **overriding** re-defines an inherited method from a superclass – same name & parameter list & return type
- ❑ **Durations** now have to be compared as **Durations** (or as **Objects**, but not as a mixture)

Equality and inheritance

Let's add a nanosecond field for fractional seconds

```
public class NanoDuration extends Duration {  
    private final int nano;  
    public NanoDuration(int min, int sec, int nano) {  
        super(min, sec);  
        this.nano = nano;  
    }  
}
```

Inherited `equals()` from `Duration` ignores `nano` so `Duration` instances with different `nanos` will be `equal`

equals: account for nano

```
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
```

Let's get symmetry

```
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;
}
```

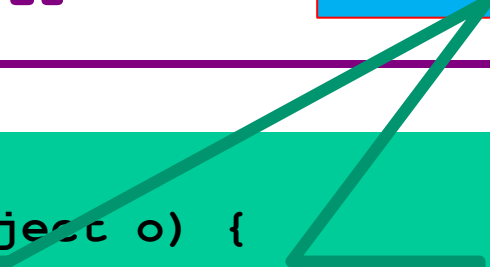
But this is not transitive!

Oops!

```
Duration d1 = new NanoDuration(5,10,15);
Duration d2 = new Duration(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!
```

Fix in Duration

Replaces earlier version
`if (! (o instanceof Duration))
 return false;`



```
@Override  
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;  
}
```

- ❑ 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
 - Consider an **ArithmeticDuration** class that adds operators but no new fields (on your own)

Another solution: avoid inheritance

- Use composition instead

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

- **NanoDuration** and **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
 - Example: can't use **NanoDurations** where **Durations** are expected (not a (Java) subtype)

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!!!

specification for `Object.hashCode`

- `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`.”
- The general contract of `hashCode` is
 - Deterministic: `o.hashCode() == o.hashCode()`
... so long as `o` doesn't change between the calls
 - Consistent with equality
 - `a.equals(b) ⇒ a.hashCode() == b.hashCode()`
 - Change `equals()`? Must you update `hashCode()`?
 - ALMOST ALWAYS! I MEAN ALWAYS!

Aside: `hashCode` and hash tables

- Classic use of hashing is selecting an index for an object in a hash table (e.g., map, set)
 - $O(1)$ cost if done right
- Java libraries do this too, but in two distinct steps:
 - `hashCode` returns an `int` value that respects equality
 - Collections scale this value as needed
 - See CSE 332 for much more...

Duration hashCode implementations

Many possibilities...

```
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!  
}
```

Consistency of equals and hashCode

Suppose we change the spec for `Duration.equals`:

```
// Return true if o and this represent the same number 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;
}
```

We must update `hashCode`, or we will get inconsistent behavior. (Why?)
This works:

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

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
- For immutable objects, equality is inherently forever
 - The object's abstract value never changes (c.f. “abstract value” in the ADT lectures) – be sure `equal` does not depend on possibly changing internal values
- 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! (Since abstract value can change.)

examples

StringBuffer is mutable, and takes the “eternal” approach

```
StringBuffer s1 = new StringBuffer("hello");  
StringBuffer s2 = new StringBuffer("hello");  
System.out.println(s1.equals(s1)); // true  
System.out.println(s1.equals(s2)); // false
```

This is reference (==) equality, which is the only way to guarantee eternal equality for **mutable** objects. (Not a problem for immutable data)

By contrast:

```
Date d1 = new Date(0); // Jan 1, 1970 00:00:00 GMT  
Date d2 = new Date(0);  
System.out.println(d1.equals(d2)); // true  
d2.setTime(1); // a millisecond later  
System.out.println(d1.equals(d2)); // false
```


Behavioral and observational equivalence

Two objects are “**behaviorally equivalent**” if:

- There is no sequence of operations that can distinguish them

- This is “eternal” equality

- Two Strings with same content are behaviorally equivalent, two Dates or StringBuffers with same content are not

Two objects are “**observationally equivalent**” if:

- There is no sequence of observer operations that can distinguish them

- Excluding mutators

- Excluding == (permitting == would require reference equality)

- Two Strings, Dates, or StringBuffers with same content are observationally equivalent

Equality and mutation

Date class implements observational equality

Can therefore **violate rep invariant** of a Set container 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 identical Dates
    System.out.println(d);
}
```

Pitfalls of observational equivalence

Equality for set elements would ideally be behavioral

Java makes no such guarantee (or requirement)

So 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**

Mutation and hash codes

Sets assume **hash codes don't change**

Mutation and observational equivalence can break this assumption too:

```
List<String> friends =  
    new LinkedList<String>(Arrays.asList("yoda", "zaphod"));  
List<String> enemies = ...; // any other list, say with "xenu"  
Set<List<String>> h = new HashSet<List<String>>();  
h.add(friends);  
h.add(enemies);  
friends.add("weatherwax");  
System.out.println(h.contains(friends)); // probably false  
for (List<String> lst : h) {  
    System.out.println(lst.equals(friends));  
} // one "true" will be printed - inconsistent!
```

More container wrinkles: self-containment

equals and hashCode methods 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;  
    }  
}
```

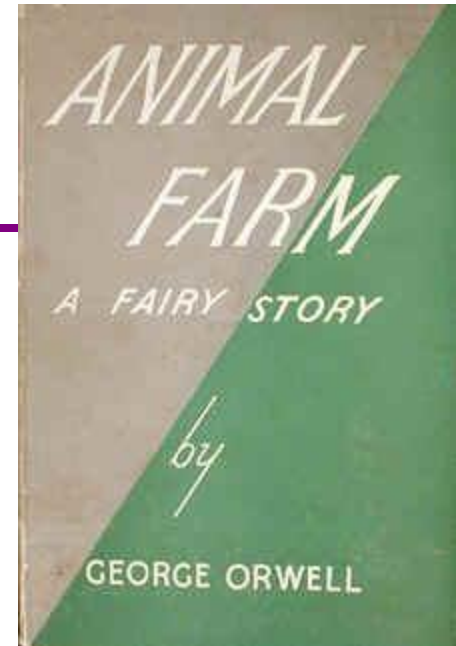
This causes an infinite loop:

```
List<Object> lst = new LinkedList<Object>();  
lst.add(lst);  
int code = lst.hashCode();
```

Summary:

All equals are not equal!

- reference equality
- behavioral equality
- observational equality



Summary: Java specifics

- Mixes different types of equality
 - Objects different from collections
- Extendable specifications
 - Objects, subtypes can be less strict
- Only enforced by the specification
- Speed hack
 - hashCode

Summary: object-oriented Issues

- Inheritance
 - Subtypes inheriting equal can break the spec.
Many subtle issues.
 - Forcing all subtypes to implement is cumbersome
- Mutable objects
 - Much more difficult to deal with
 - Observational equality
 - Can break reference equality in collections
- Abstract classes
 - If only the subclass is instantiated, we are ok...

Summary: software engineering

- Equality is such a simple concept
- But...
 - Programs are used in unintended ways
 - Programs are extended in unintended ways
- Many unintended consequences
- In equality, these are addressed using a combination of:
 - Flexibility
 - Carefully written specifications
 - Manual enforcement of the specifications
 - perhaps by reasoning and/or testing