Equality

Michael Ernst

CSE 331
University of Washington

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 value?
  - Same concrete value, or same abstract value?
  - Is equality temporary or forever?
  - How does equality behave in the presence of inheritance?
  - Is equality of collections related to equality of elements?
    - What about self-containment?
  - How can implement equality efficiently?

Properties of equality

Equality is an *equivalence relation*; that is:

- *Reflexive* **`a.equals(a)`**

  $3 \neq 3$ would be wrong

- *Symmetric* **`a.equals(b)`** $\Leftrightarrow$ **`b.equals(a)`**

  $3 = 4 \wedge 4 \neq 3$ would be wrong

- *Transitive* **`a.equals(b)`** $\wedge$ **`b.equals(c)`**
  $\Rightarrow$ **`a.equals(c)`**

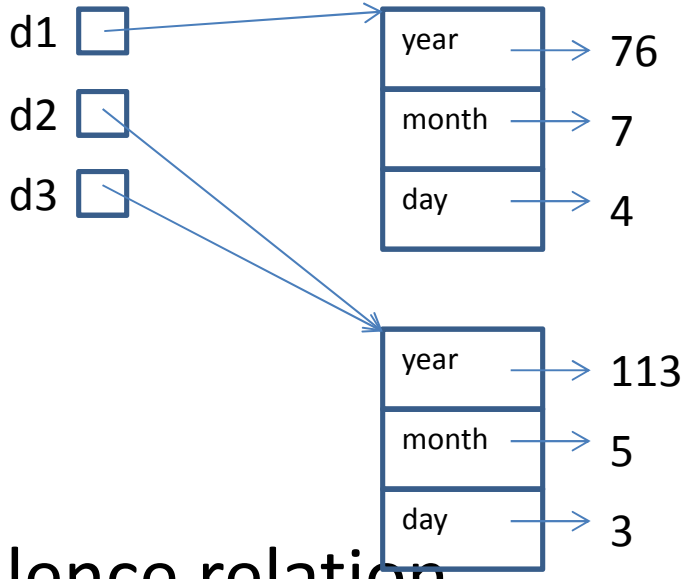
  $((1+2) = 3 \wedge 3 = (5-2)) \wedge$
  $((1+2) \neq (5-2))$ would be wrong

Reference equality

- a == b
  - True if a and b point to the same object

```
Date d1 = new Date(76,7,4);
Date d2 = new Date(113,5,3);
Date d3 = d2;

   // T/F: d1 == d2 ?
   // T/F: d1 == d3 ?
   // T/F: d2 == d3 ?
   // T/F: d1.equals(d2) ?
   // T/F: d2.equals(d3) ?
```



- Reference equality is an equivalence relation
- Reference equality is the strongest definition of equality
  - Weaker definitions can be useful

`Object.equals` method

The **`Object.equals`** implementation is very simple:

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

Yet its specification is much more elaborate. Why?

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

- Object class is designed for inheritance
- Its specification will apply to all subtypes
  - In other words, all Java classes
  - The specification for equals cannot later be weakened
- 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 true subtype of Object
  - Instead the spec for equals enumerates basic properties that clients can rely on in subtypes of Object
  - a==b is compatible with these properties, but so are other tests

Properties of equals

Equality is reflexive

a.equals(a) is true

Equality is symmetric

a.equals(b) $\Leftrightarrow$ b.equals(a)

Equality is transitive

a.equals(b) and b.equals(c) $\Rightarrow$ a.equals(c)

No object equals **`null`**

a.equals(null) = false

(other conditions omitted for now)

Default implementation (reference equality) satisfies these conditions

Beyond reference equality

We often want to compare 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 would like this to be true
```

An incorrect equals method

Let's create an equals method that compares fields:

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

This is an equivalence relation (reflexive, symmetric, and transitive) for **`Duration`** objects

```
Duration d1 = new Duration(10,5);
Duration d2 = new Duration(10,5);
System.out.println(d1.equals(d2));   // True!
```

What is an example of code for which this fails?

```
Object o1 = new Duration(10,5);
Object o2 = new Duration(10,5);
System.out.println(d1.equals(d2));   // False! (oops)
```

Error: Duration.equals overloaded Object.equals

The Duration equals method incorrectly *overloaded* the Object one

- instead of *overriding* as intended

General rule:

- A method family contains multiple implementations of the same signature (name + parameter types)
- The method family is determined at compile time based on static types
- Which implementation from the method family to run is determined at run time based on dynamic types (classes)

Overloading: two unrelated methods with the same name, **`equals`**:

`equals(Object)` in class **`Object`**
`equals(Duration)` in class **`Duration`**

- The compiler chooses which method to invoke based on compile-time types of all arguments (including the receiver)
- If **`x`** has compile-time type **`Duration`**, **`x.equals(Duration)`** invokes **`Duration`** .equals
- If **`x`** has complie-time type **`Object`**, **`x.equals(Duration)`** invokes **`Object.equals`**

What is the result of each of these?

```
d1.equals(d2); // True
o1.equals(o2); // False
d1.equals(o2);
o1.equals(d2);
```

Overriding: If same name and parameter types, then multiple implementations of the same method family

- The run-time system chooses which implementation to run based on the run-time type of the recevver (only)

A correct equals method for Duration

Compiler warns if the signature does not match (i.e., if this overloads and creates a new method family)

```
@Override
public boolean equals(Object o) {
    if (! (o instanceof Duration))
        return false;
    Duration d = (Duration) o;
    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

Let's add a nano-second 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;
    }
}
```

We need to overload equals.

If we inherit equals() from Duration, nano will be ignored,

and objects with different nanos will be equal.

Symmetry bug

A first attempt at an equals method for NanoDuration:

```
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);
System.out.println(d1.equals(d2)); // false
System.out.println(d2.equals(d1)); // true
```

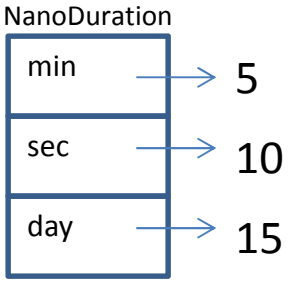
Symmetry fix for NanoDuration.equals

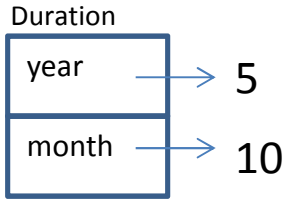
```
public boolean equals(Object o) {
  if (! (o instanceof Duration))
    return false;
  // if o is a normal Duration, compare
  without nano
  if (! (o instanceof NanoDuration))
    return super.equals(o);
  NanoDuration nd = (NanoDuration) o;
  return super.equals(nd) && nano == nd.nano;
}
```

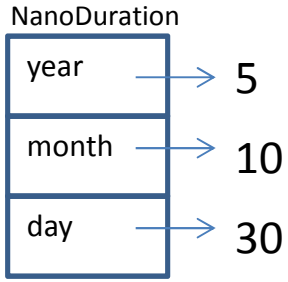
This is not transitive!

Transitivity bug

```
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!
```







Checking exact class, instead of instanceof

Duration can avoid comparing against an instance of a subtype:

```
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;
}
```

Previously:

```
if (! (o instanceof Duration))
        return false;
```

Problems:

- 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

Another solution: avoid inheritance

Can use composition instead:

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

NanoDurations and Durations are unrelated

- There is no presumption that NanoDurations and Durations may be equal
- Solves some but not all problems, and introduces others
  - Example: can't use NanoDurations where Durations are expected (not a Java subtype)

Date and Timestamp in Java

public class Timestamp extends Date

> "A thin wrapper around java.util.Date that ... adds the ability to hold the SQL TIMESTAMP nanos value and provides formatting and parsing operations ..."

Caveat 1

> "The Timestamp.equals(Object) method is not symmetric with respect to the java.util.Date.equals(Object) method."

Caveat 2

> "Also, the hashcode method uses the underlying java.util.Date implementation and therefore does not include nanos in its computation."

Date and Timestamp in Java

Caveat 3

> "Due to the differences between the Timestamp class and the java.util.Date class mentioned above, it is recommended that code not view Timestamp values generically as an instance of java.util.Date. The inheritance relationship between Timestamp and java.util.Date really denotes implementation inheritance, and not type inheritance."

Translation:

> "Timestamps are not Dates. Ignore that **`extends Dates`** bit in the class declaration."

Timestamp: overloading error

public boolean equals(Timestamp ts)

"Tests to see if this Timestamp object is equal to the given Timestamp object."

public boolean equals(Object ts)

"Tests to see if this Timestamp object is equal to the given object. This version of the method equals has been added to fix the incorrect signature of Timestamp.equals(Timestamp) and to preserve backward compatibility with existing class files. Note: This method is not symmetric with respect to the equals(Object) method in the base class."

A special case: uninstantiable types

- No equality problem if superclass cannot be instantiated!
  - For example, suppose Duration were abstract
  - Then no troublesome comparisons can arise between Duration and NanoDuration instances
- This may be why this problem is not very intuitive
  - In real life, “superclasses” can't be instantiated
  - We have specific apples and oranges, never unspecialized Fruit

Efficiency of equality

Equality tests can be slow

- E.g., compare two text documents or video files

It can be useful to quickly prefilter

- Example: are the files same length?
- If not, they are not equal
- If so, then they might be equal
  - They need to be compared

```
if (file1.length() != file2.length()) {
  return false;
} else {
  ... // do full equality check
}
```

Hash codes are efficient prefilters for equality

- Do objects have same hash code?
- If not, they are not equal
- If so, then they might be equal
  - They need to be compared

Aside: another use for `hashCode`

- Compute an index for an object in a hash table
- This is a special case of prefiltering for equality!
  - If you know about hash tables, think about this until you understand why.
  - If you don't know about hash tables, ignore this.

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:

- 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()`**

Many possible `hashCode` implementations

```
public class Duration {
  public int hashCode() {
    return 1;          // always safe, but no prefiltering
  }
}
public class Duration {
  public int hashCode() {
    return min;        // safe, but inefficient for Durations
  }                    // that differ in sec field only
}
public class Duration {
  public int hashCode() {
    return min+sec;    // safe, and changes in any field
  }                    // will tend to change the hash code
}
```

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, the answer is “yes”
  - In Java, the answer is “you choose”
  - The Object contract doesn't specify this (why not?)
- For immutable objects
  - Abstract value never changes
  - Equality is automatically forever
- For mutable objects, equality can either:
  - Compare abstract values (field-by-field comparison),
  - Or be eternal.
  - Can't do 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.

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 the same content are behaviorally equivalent; two Dates or StringBuffers with the same content are not
- Two objects are "observationally equivalent" ifthere 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 wiith "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:

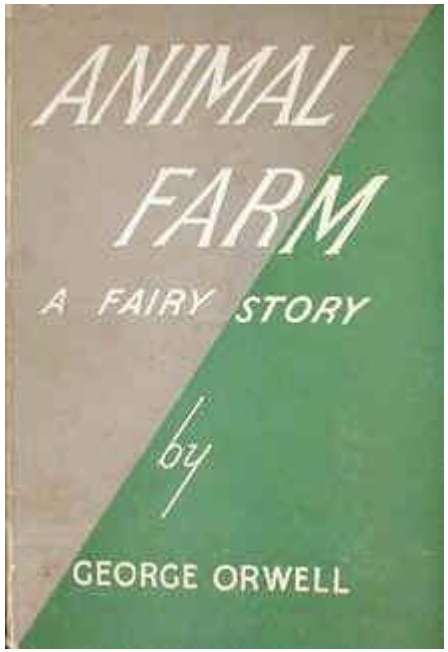
```
class ArrayList<E> {
  public int hashCode() {
     int code = 1;
     for (Object o : list)
       code = 31*code + (o==null ? 0 : o.hashCode());
    return code;
  }
```

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 are treated differently than 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