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 contents/value?
  - Same concrete value, or same abstract value?
  - Same right now or same forever?
  - Interaction with inheritance (subclasses)
  - When are two collections equal?
    - Relationship to equality of elements? Order of elements?
    - What if a collection contains itself?
  - How to implement equality correctly and efficiently

Properties of equality

Reflexive

`a.equals(a) == true`

Symmetric

`a.equals(b)` ⇔ `b.equals(a)`

Transitive

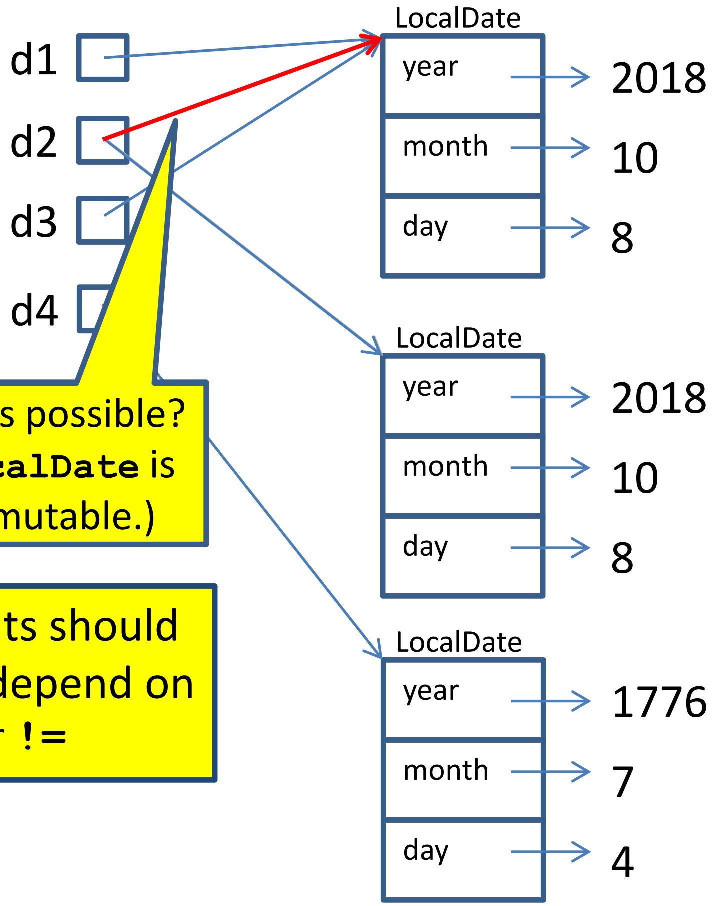
`a.equals(b)` ∧ `b.equals(c)` ⇒ `a.equals(c)`

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

Reference equality

- An object is equal only to itself
  - True if a and b refer to (point to) the same object
  - In Java: **`a == b`**

```
LocalDate d1 = LocalDate.now();
LocalDate d2 = LocalDate.now();
// T/F: d1 == d2 ?
// T/F: d1.equals(d2) ?
LocalDate d3 = d1;
// T/F: d1 == d3 ?
// T/F: d1.equals(d3) ?
LocalDate d4 = LocalDate.of(1776,
// T/F: d1 == d4 ?
// T/F: d1.equals(d4) ?
```



Is this possible? (**`LocalDate`** is immutable.)

Clients should not depend on == or **`!=`**

Reference equality

- An object is equal only to itself
  - True if a and b refer to (point to) the same object
  - In Java: **`a == b`**
- Reference equality is an equivalence relation
  - Reflexive
  - Symmetric
  - Transitive
- Reference equality is the *strongest* definition of equality
  - It is the *smallest* equivalence relation on objects
    - Why can't an equivalence relation be smaller than it?
  - 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;  // reference equality
  }
}
```

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)

Weak specification:
“The equals method implements an equivalence relation, and null does not equal anything else.”

- It is *consistent*: for any reference values x and y, multiple

The default implementation (reference equality) satisfies this specification.

- 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). …

Method specs and subtypes

- Subclasses can extend Object and override **`equals`**
- Subclasses must satisfy the spec of **`equals`**
  - Every Java class is a subclass of Object
- The spec of equals must be appropriate for every Java class
  - Subclasses may specify a stronger contract
  - Subclasses may not specify a weaker contract
- If Object.equals specified reference equality,
  every Java class would have to use reference equality

- When you write a spec, think about all possible subtypes
  - Write a weak, flexible spec to accommodate their needs
  - Balance needs of clients with needs of subtype implementors
  - Don’t enshrine implementation details in the spec

A class that needs less-strict equality

Here is a class that inherits Object.equals:

```
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
// We would like this to be true, so let's override equals
```

An incorrect equals method

Let's create an equals method that compares fields:

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

Overloads,
does not override,
`Object.equals(Objejct)`

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(o1.equals(o2));   // False! (oops)
```

Review:
Which implementation gets run?

1. Resolve overloading at compile time
   - Let R be the compile-time type of the receiver
   - Choose the most specific, applicable, accessible operation in R
     - Accessible operations: Visible (`public`, `private`, `protected`)
     - Applicable operations: Those whose parameter types are supertypes of the argument types
     - Most specific: its parameter types are subtypes of the corresponding parameter types for other applicable ops
       - If no most specific exists, compile-time error

   This picks a method family or signature
2. Resolve overriding at run time (dynamic dispatch)
   - Run the implementation in the run-time type of the receiver
     - Might be inherited from a superclass

Overloading resolution

Implementation without overriding:

```
class Object {
  boolean equals(Object o) {
    return this == o;
  }
}
class Duration extends Object {
  boolean equals(Duration d) {
    if (d == null)
      return false;
    return d.min == min
        && d.sec == sec;
   }
}
```

```
Duration d1
  = new Duration(10,5);
Duration d2
  = new Duration(10,5);
Object o1 = d1;
Object o2 = d2;
d1.equals(d1);   true
d1.equals(d2);   true
d1.equals(o1);   true
d1.equals(o2);   false
o1.equals(d1);   true
o1.equals(d2);   false
o1.equals(o1);   true
o1.equals(o2);   false
```

`Duration.equals`

`Object.equals`

A correct equals method for Duration

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

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

`instanceof` evaluates to false if value is null

Rare use of cast that is idiomatic

Reflexive, symmetric, and transitive for all values

Equality and inheritance

Count how many objects have ever been created:

```
public class CountedDuration extends Duration {
    public static numCountedDurations = 0;
    public CountedDuration(int min, int sec) {
        super(min, sec);
        numCountedDurations++;
    }
}
```

Does not override equals; inherits from **`Duration`**

Any combination of **`Duration`** and **`CountedDuration`** objects can be compared

- Equal if same values in **`min`** and **`sec`** fields
- Works because **`o instanceof Duration`** is **`true`** when **`o`** is an instance of **`CountedDuration`**

`numCountedDurations` is not part of the abstract state

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 override `equals`. Why?

If we inherit `equals()` from `Duration`, `nano` will be ignored, and objects with different `nano`s will be equal.

Symmetry bug

A first attempt at an equals method for NanoDuration (using the rules we have learned so far):

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

This is not symmetric!

```
Duration nd = new NanoDuration(5, 10, 15);
Duration d = new Duration(5, 10);
System.out.println(nd.equals(d)); // false
System.out.println(d.equals(nd)); // true
```

Two ways to fix the symmetry bug:

1. Change to **`true`**
2. Change to **`false`**

Proposed symmetry fix #1:

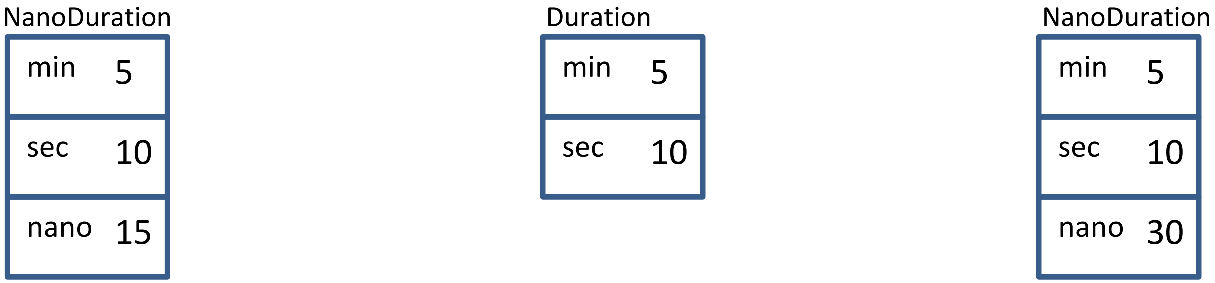
`nd.equals(d)` ⇒ `true` (ignore `nano`)

```
class NanoDuration extends Duration {
  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 n = (NanoDuration) o;
    return super.equals(n) && nano == n.nano;
  }
  … }
```

This is not transitive!

Transitivity bug

```
Duration nd1 = new NanoDuration(5, 10, 15);
Duration d = new Duration(5, 10);
Duration nd3 = new NanoDuration(5, 10, 30);
System.out.println(nd1.equals(d2)); // true
System.out.println(d2.equals(nd3)); // true
System.out.println(nd1.equals(nd3)); // false!
```



NanoDuration

min	5
sec	10
nano	15

Duration

min	5
sec	10

NanoDuration

min	5
sec	10
nano	30

Transitivity bug

- The bug is in the specification: "A **`Duration`** equals a **`NanoDuration`** if their **`min`** and **`sec`** fields correspond, ignoring the **`nano`** field" (not an equivalence relation)
  - We coded without a specification
  - This is a *design* problem, and needs a design solution
  - Lesson: write and review your spec before coding
- Must use symmetry fix #2: **`d.equals(nd)`** ⇒ **`false`**
  Solution: no **`Duration`** equals any **`NanoDuration`**
  (But, **`Duration.equals`** cannot special-case **`NanoDuration`**)
  1. Change **`Duration.equals`** so it does not consider *any* subclass to be equal
  2. Change **`NanoDuration`** so it is not a subclass of **`Duration`**

Checking exact class, instead of instanceof

Duration can avoid comparing against an instance of a subtype:

```
@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;
}
```

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
  - **`Duration`** objects never equal **`CountedDuration`** objects
  - Consider an **`ArithmeticDuration`** class that adds operators but no new fields

Another solution: avoid subtyping

Use composition instead:

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

NanoDurations and Durations are now unrelated

- Unrelated objects are never equal
- Solves some but not all problems, and introduces others
  - Can’t use a NanoDuration where a Duration is expected (not a Java subtype)
  - Tedious, error-prone implementation with lots of forwarding methods

A base class reduces code duplication

- Can avoid some method redefinition by having **`Duration`** and **`NanoDuration`** both extend a common abstract class
  - Or implement the same interface
  - Leave overriding **`equals`** to the two subclasses

- Still no subtyping or substitution of **`NanoDuration`** for **`Duration`**
- Requires advance planning, or willingness to change **`Duration`** when you discover the need for **`NanoDuration`**

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:

> "`Timestamp`s are not **`Date`**s. Ignore that **`extends Date`** 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
}
```

- A hash code is an efficient prefilter 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 how hash tables are implemented, 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)` ⇒ **`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 poor prefiltering 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:

```
// Returns 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 min*60 + sec == d.min*60 + d.sec; // same # of sec.
}
```

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

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

Don't use arithmetic to compute hashes! Typical implementation of **`hashCode`** (few exceptions):
`Objects.hash(field1, field2, field3);`

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 (even if rep changes)
- For mutable objects, equality can either:
  - Compare abstract values (field-by-field comparison),
  - Or be eternal (reference equality).
  - 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.

Date is mutable and takes the "abstract value" approach:

```
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 (excluding ==) 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" if there is no sequence of *observer* operations that can distinguish them

  - Excluding mutators
  - Excluding == (permitting == would require reference equality)
- Two **`String`**s, **`Date`**s, or **`StringBuffer`**s 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 of the same Date
    System.out.println(d);
}
```

Pitfalls of observational equivalence

Equality for set elements would ideally be behavioral

Java makes no such guarantee (or requirement)

So we 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 [or hash codes] while the object is an element in the set."

Same problem applies to keys in maps

Libraries choose not to copy-in for performance and to preserve object identity

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("zaphod", "yoda"));
List<String> enemies = ...;   // any other list, say with “xenu”
Set<List<String>> h = new HashSet<>();
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 with “false” for contains()
```

More container wrinkles: self-containment

The **`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 client code causes an infinite loop in **`hashCode`**:

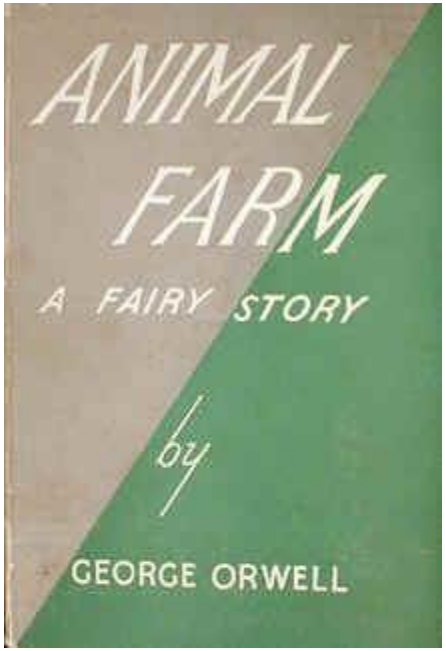
```
List<Object> lst = new LinkedList<Object>();
lst.add(lst);
int code = lst.hashCode();
```

Summary:
All equals are not equal!

- reference equality
- behavioral equality
- observational equality


stronger

weaker



Summary: Java specifics

- Mixes different types of equality
  - Objects are treated differently than collections
- Extendable specifications
  - Subtypes can be less strict
- Only enforced by the specification
- Speed hack
  - **`hashCode`**

Summary: object-oriented Issues

- Inheritance
  - Subtypes inheriting equals 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