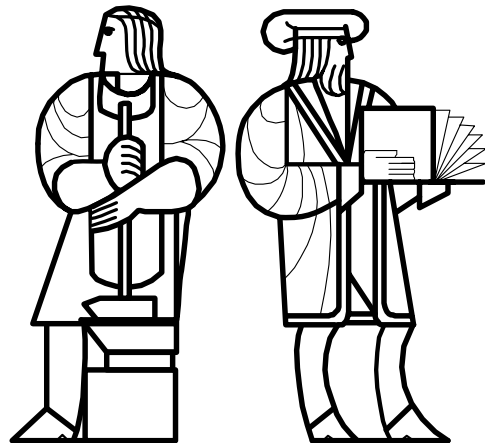


6.170 Lecture 11

Equality



MIT EECS

object equality

A simple idea – we have intuitions about equality:

Two objects are equal if they have the same value

Two objects are equal if they are indistinguishable

Many Subtle Issues

The concept of Equality

Is equality temporary or forever?

How Java treats Equality

Is equality same everywhere?

How can we make equality an efficient operation?

How OO Programming impact this simple concept

How does equality behave in the presence of inheritance?

Is equality of collections related to equality of elements?

What happens to equality given self-containment and other non-hierarchical forms?

Maintaining a simple concept in a complex, malleable system

How do our ideas of equality play out in large systems?

Unintended consequences

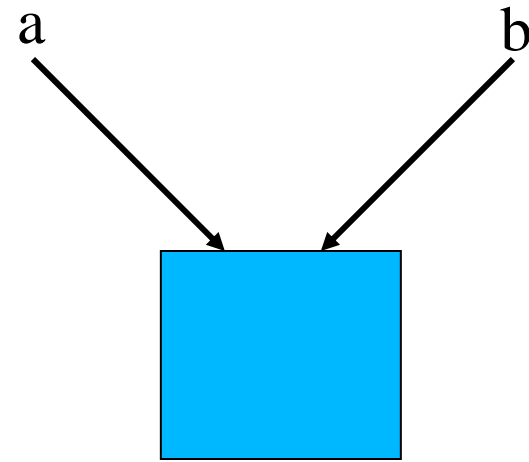
Referential Equality

$a == b$

True if both a and b points to the same object

Strongest definition of equality.

There are other weaker definitions of equality that are also important.



Object.equals method

The Object.equals method is very simple:

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

Yet its specification is much more elaborate. Why?

Equals Specification

Equals

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:

- [Boolean.hashCode\(\)](#), [Hashtable](#)

the Object Contract

Object class is designed for inheritance

Its specification will apply to all subtypes

In other words, all Java classes

So 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

basic specification for equals

Equality is reflexive

$a.equals(a)$ is true

Equality is symmetric

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

Equality is transitive

$a.equals(b) \text{ and } b.equals(c) \implies a.equals(c)$

No object equals null

$a.equals(\text{null}) = \text{false}$

(other conditions omitted for now)

default Object.equals implementation

Equality is reflexive

$$a == a$$

Equality is symmetric

$$a == b \iff b == a$$

Equality is transitive

$$a == b \text{ and } b == c \implies a == c$$

No object equals null

$$(a == \text{null}) = \text{false for any non-null reference}$$

Default implementation (“referential equality”) works fine

beyond referential equality

Often want to compare objects less strictly

```
public class Duration {  
    private final int day;  
    private final int sec;  
    public Duration(int day, int sec) {  
        this.day = day; this.sec = sec;  
    }  
}
```

```
Duration d1 = new Duration(10,5);
```

```
Duration d2 = new Duration(10,5);
```

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

```
// But maybe we would like this to be true - why not?
```

an equals method

Let's try adding an equals method that compares fields:

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

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

This is reflexive, symmetric, transitive for Duration objects
But it doesn't override the Object.equals method

the problem

Here's the problem:

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

This will make our life complicated

Example: Durations may behave strangely in Collections

Avoid overloading methods

If same name but different arguments, then Java considers it unrelated

Occasional uses for optimization, especially when the semantics are identical

try again

Here's an alternative equals method for Duration:

```
@Override // compiler will warn if type mismatch
public boolean equals(Object o) {
    if (!(o instanceof Duration))
        return false;
    Duration d = (Duration) o;
    return d.day == day && 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 day, int sec, int nano) {  
        super(day, sec);  
        this.nano = nano;  
    }  
    // What should we do about equals?  
    // If we inherit from Duration, nano will be ignored  
    // and objects with different nanos will be equal  
}
```

Equals for the subclass

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;  
}
```

```
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
```

This has a problem – it is not symmetric!

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;  
}
```

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!
```

What is the solution?

Can check exact class in Duration, rather than just use instanceof

But then can't do any minor subclassing, for example to make an ArithmeticDuration class that offers no new fields, just a few new operators

checking exact class

Here's how in Duration we could avoid ever comparing against an instance of a subtype:

```
public boolean equals(Object o) {  
    if (o == null || !o.getClass().equals(getClass()))  
        return false;  
    Duration d = (Duration) o;  
    return d.day == day && d.sec == sec;  
}
```

But now every subtype must override equals

Even if it wants the identical definition

Hard to compare subtypes to one another

another solution: avoid inheritance

Can use composition:

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

Now there is no presumption that NanoDurations and Durations may be equal

an important special case

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

equality and efficiency

Equality tests can be slow

E.g. testing if two text documents are equal

Or testing for equality between millions of objects

Useful to be able to quickly prefilter

E.g. are documents same length?

If not, they are not equal

If so, then they are worth testing for equality

Hash codes are efficient prefilters for equality

Do objects have same hash code?

If not, they are not equal

If so, then they are worth testing for equality

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`.”

[Hashtables are explained in the Javadocs, Bloch, 6.046.]

The general contract of `hashCode` is:

“Whenever it is invoked on the same object, the `hashCode` method must consistently return the same integer, provided no information used in equals comparisons on the object is modified.”

“If two objects are equal according to the `equals(Object)` method, then calling the `hashCode` method on each of the two objects must produce the same integer result.”

hashCode for Duration?

Any of these hashCode implementations would work.

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

equals and hashCode

Suppose that day and sec do not have to be equal for Duration to be equal – suffices for same total number of seconds to be represented

```
public boolean equals(Object o) {  
    if (!(o instanceof Duration))  
        return false;  
    Duration d = (Duration) o;  
    return 24*60*60*day+sec ==  
        24*60*60*d.day+d.sec;  
}
```

Now we have to update our hash codes as well, or we will get inconsistent behavior. This works:

```
public int hashCode() {  
    return 24*60*60*day+sec;  
}
```

equality and time

If two objects are equal now, will they always be equal?

In mathematics, answer is “yes”

For Java, answer is “you choose”

Object contract doesn't nail this down

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 referential (==) equality, which is the only way to guarantee eternal equality for mutable objects. Compare:

```
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
```

two types of 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 StringBufferers with same content are not

Two objects are “observationally equivalent” if:

There is no sequence of *observer* operations that can distinguish them (that is, we exclude mutators)

Two Strings, Dates, or StringBufferers with same content are observationally equivalent

We exclude == (limits equality to referential equality)

approaches to equality

Liskov approach

equals method is always behavioral equivalence

- referential equality for mutable objects
- value equality for immutable objects

similar method is observational equivalence

- value equality for both mutable and immutable object

Java approach

Mixed! Read specs, especially in the Collections

Collection equality is observational equality

equality and mutation

```
Set<Date> s = new HashSet<Date>();  
    Date d1 = new Date(0);  
    Date d2 = new Date(0);  
    s.add(d1);  
    s.add(d2);  
  
    for (Date d : s) {  
        System.out.println(d);  
    }
```

equality and mutation

```
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) {
        System.out.println(d);
    }
```

Date class implements observational equality

Can therefore violate rep invariant of a Set container by mutating after insertion

caveats in specs

Equality for set elements would ideally be behavioral

But no guarantee (or requirement) that this is so in Java

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

self-containment

equals and hashCode methods on containers are recursive, e.g. hashCode for List<E>:

```
int code = 1;
for (Object o : list) {
    code = 31*code + (o==null ? 0 : o.hashCode());
}
```

Then if we try something like this:

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

We get an infinite loop

Summary: Equality

All Equals are not Equal!

Referential Equality

Behavioral Equality

Observational Equality

Summary: Java Issues

A Mixed Approach

Objects different from collections

Extendable specifications

Objects, subtypes can be less strict

Only enforced by the specification

Speed Hack

HashCode

Summary: Object Oriented Issues

Issues with Inheriting

Subtypes inheriting equal can break the spec. Many subtle issues showed-up.

Forcing all subtypes to implement is cumbersome

Mutable objects

Much more difficult to deal with

Observational Equality

Can break Referential 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 doing a lot of testing