Equality

In the physical world

- Every object is distinct -- at some level, even two snowflakes are different
  - Even if the distinction is just the position they occupy in space
- So two physical objects are never truly “equal” to each other, only degrees of similarity

In the world of human (or mathematical) language

- You can have multiple names for the same object
- So it’s natural to ask when two expressions represent the same object
  - 1+2
  - 3
  - $\sqrt{9}$
  - $(\lambda x \mid x+1) \ 2$
Two types of equality

Referential Equality
- Two objects are the same
- If an object is mutable → referential equality is common (more later)

Observational Equality
- If two objects looks the same in all observations, they should be the same
- Interesting representation for immutable objects
Two types of equality in Java

Referential Equality
- Two objects are the same
- If an object is mutable → referential equality is common (more later)
- == compares object locations
  - two references are == if they point to the same storage in memory

Observational Equality
- If two objects looks the same in all observations, they should be the same
- Interesting representation for immutable objects
- equals() compares object contents
  - In other words, object equality
Why Equality is important?

Reduce Storage
- Ability to coalesce multiple objects that are the same
- Ex: Multiple strings that represent the same string can be stored once

Reduce Computation
- Ex: If the method has performed the same computation before, don’t need to redo the computation

Reduce Synchronization
- Multiple threads can create/reference multiple objects instead of needing to synchronize to create a single one

Reduce Complexity
- Only need to check equality even though the representation might be different.
Two Views of Observational Equality

As a relation

An equivalence is a relation $R \subseteq T \times T$ that is:

- reflexive: $R(t, t)$ for all $t \in T$
- symmetric: $R(t, u) \Rightarrow R(u, t)$
- transitive: $R(t, u) \land R(u, v) \Rightarrow R(t, v)$

To use $R$ as a definition for equality: $a$ equals $b$ if and only if $R(a, b)$

As an interpretation function

An interpretation function $f: E \rightarrow V$ maps expressions to abstract values

To use $f$ as a definition for equality of expressions: $a$ equals $b$ if and only if $f(a) = f(b)$

These notions are equivalent

- an equivalence relation induces an interpretation function (the relation partitions $T$, so $f$ maps each element to its partition class)
- the relation induced by an interpretation function is an equivalence relation (check the three properties)
Two Views of Observational Equality

Two possible definitions

- Two expressions are equal when they denote the same abstract value
- Two expressions are equal when they cannot be distinguished by observation

Example

- consider the set expressions \{1,2\} and \{2,1\}
- the unique abstract set they both denote contains exactly 1 and 2
- under the observers |...| and \(\in\), they’re indistinguishable
  - \(|\{1,2\}| = 2\) and \(|\{2,1\}| = 2\)
  - \(1 \in \{1,2\}\) is true, and \(1 \in \{2,1\}\) is true
  - \(2 \in \{1,2\}\) is true, and \(2 \in \{2,1\}\) is true
  - \(3 \in \{1,2\}\) is false, and \(3 \in \{2,1\}\) is false
- So both approaches produce the same results in the world of mathematical expressions
Equality in Immutable ADTs

Mapping these viewpoints to abstract data types

- Interpretation function is the abstraction function
  - i.e., two rep objects r1 and r2 are equal iff the abstraction function A maps them to the same abstract value, i.e. A(r1) = A(r2)

- “Indistinguishable by observation” means calling methods
  - i.e., two immutable objects are equal iff they cannot be distinguished by calling methods on the objects
Equals – Simple concept but…..

➢ Our first try at Duration.equals():

```java
public class Duration {

    ...

    public boolean equals (Duration that) {
        return this.getSecs() == that.getSecs();
    }

}
```

© Robert Miller 2007
public class Duration {
    // abstract duration is an integer representing
    // total number of seconds
    // A(r) = r.getSecs()
    private final int mins;
    private final int secs;
    public Duration(int m, int s) {
        mins = m;
        secs = s;
    }
    public int getSecs() {
        return mins*60 + secs;
    }
}

Duration d1 = new Duration (1, 2);
Duration d2 = new Duration (1, 3);
Duration d3 = new Duration (0, 62);

Which of these should be considered equal?
equals() must be an equivalence

Implementation of equals() must satisfy the three properties
- Reflexive: a.equals(a) for all non-null references a
- Symmetric: a.equals(b) \implies b.equals(a)
- Transitive: a.equals(b) \land b.equals(c) \implies a.equals(c)

Suprisingly easy to get this wrong
- What if b is null?
  - Better return false if we want symmetry
- What if we wanted a tolerance in comparing Durations:

```java
public class Duration {
    private static final int CLOCK_SKEW = 5; // seconds
    public boolean equals(Duration that) {
        return Math.abs(this.getSecs() - that.getSecs()) <= CLOCK_SKEW;
    }
}
```
- Which property is violated?

© Robert Miller 2007
public class Duration {
    ... 
    public boolean equals (Duration that) {
        return this.getSecs() == that.getSecs();
    }
}

Duration d1 = new Duration (1, 2);
Duration d2 = new Duration (1, 2);
Object o2 = new Duration(1,2);

d1.equals(d2) → true
o2.equals(d2) → true
d2.equals(o2) → false
Overriding vs. Overloading

What’s going on?
➢ Duration has overloaded the equals() method inherited from Object, instead of overriding it

```java
public class Object {
    public boolean equals (Object that) { return this == that; }
}
public class Duration {
    public boolean equals (Duration that) {
        return this.getSecs() == that.getSecs();
    }
}
```

© Robert Miller 2007

Implicitly added by Java
Fixing Equals()

@Override // compile-time error if doesn’t match a superclass method
public boolean equals (Object othat) {
    if (! (othat instanceof Duration)) return false; // handles null othat
    Duration that = (Duration) othat;
    return this.getSeconds() == that.getSeconds();
}

➤ This fixes the immediate problem:

Duration d1 = new Duration(1, 2);
Object o2 = new Duration(1, 2);

d1.equals(o2) → true
o2.equals(d1) → true
Subclassing

Suppose we subclass Duration

```java
public class PreciseDuration extends Duration {
    private final int millisecs;
    public PreciseDuration(int m, int s, int ms) {
        super(m, s);
        millisecs = ms;
    }
    public int getMillisecs() {
        return super.getSeconds() * 1000 + millisecs;
    }
}
```

How should equality be defined for PreciseDuration?

• Can we simply use the equals() inherited from Duration? No, because it ignores milliseconds.
Subclassing and Equals()

Can we simply override equals() in the same way?

```java
public class PreciseDuration extends Duration {
    ...
    @Override
    public boolean equals (Object othat) {
        if (! (othat instanceof PreciseDuration)) return false;
        PreciseDuration that = (PreciseDuration) othat;
        return this.getMillisecs() == that.getMillisecs();
    }
}
```

Duration d1 = new PreciseDuration(1, 2, 3);
PreciseDuration pd2 = new PreciseDuration(1, 2, 3);

d1.equals(pd2) → true
pd2.equals(d1) → true
Another Try

➢ Suppose we use the superclass definition of equals() in all cases except for comparing two PreciseDuration objects

```java
public class PreciseDuration extends Duration {
    ...
    @Override
    public boolean equals(Object othat) {
        if (! (othat instanceof PreciseDuration)) return super.equals(othat);
        PreciseDuration that = (PreciseDuration) othat;
        return this.getMillisecs() == that.getMillisecs();
    }
}
```

Duration `d1 = new Duration(1, 2);`
PreciseDuration `pd2 = new PreciseDuration(1, 2, 3);`

d1.equals(pd2) → true
pd2.equals(d1) → true
Subclassing and Equals()

Can we simply override equals() in the same way?

```java
public class PreciseDuration extends Duration {
    ...
    @Override
    public boolean equals (Object othat) {
        if (! (othat instanceof PreciseDuration)) return false;
        PreciseDuration that = (PreciseDuration) othat;
        return this.getMillisecs() == that.getMillisecs();
    }
}
```

Duration `d1 = new Duration(1, 2);`
PreciseDuration `pd2 = new PreciseDuration(1, 2, 3);`

d1.equals(pd2) → true
pd2.equals(d1) → false

Not Symmetric!
Another Try

➤ Suppose we use the superclass definition of equals() in all cases except for comparing two PreciseDuration objects

```java
public class PreciseDuration extends Duration {
    ...
    @Override
    public boolean equals(Object othat) {
        if (! (othat instanceof PreciseDuration)) return super.equals(othat);
        PreciseDuration that = (PreciseDuration) othat;
        return this.getMillisecs() == that.getMillisecs();
    }
}
```

PreciseDuration pd1 = new PreciseDuration(1, 2, 3);
Duration d2 = new Duration(1, 2);
PreciseDuration pd3 = new PreciseDuration(1, 2, 10);

pd1.equals(d2) → true
d2.equals(pd3) → true
Not Transitive!
pd1.equals(pd3) → false
Solving the Subclassing Problem

No really satisfactory solution

- Standard approach: superclass equality should reject all subclass objects
  - i.e., instead of
    ```java
    if (! (othat instanceof PreciseDuration)) return false;
    ```
  use:
    ```java
    if (! (othat.getClass().equals(getClass())))) return false;
    ```

But this is inflexible

- e.g., doesn’t permit a subclass that doesn’t add any new abstract values

Better solution

- avoid inheritance, and use composition instead
- see Bloch, Effective Java, Item 14

© Robert Miller 2007
Immutable objects can change internal state without losing their immutability

- As long as the state change has no observable effect on the object’s abstract value
- A change like this is called a **beneficent side-effect**
- Often done for performance reasons

**Examples**

- Caching results: e.g. Expr.eval() might store the result of the evaluation in the root of the tree, so that it doesn’t have to do it again
- Rebalancing: e.g., a splay tree is a binary search tree that moves the most recently accessed element to the top of the tree

**State changes caused by beneficent side effects shouldn’t affect equality**
Hash Tables and HashCode

A hash table is a representation for a mapping:

- an abstract data type that maps keys to values.
- Hash tables offer constant time lookup, so they tend to perform better than trees or lists.

Hashcodes map objects to “buckets”

- Store Object $\rightarrow$ calculate the hashcode, find the bucket, store
- Lookup $\rightarrow$ calculate the hashcode, find the bucket, search within it

Good Hashcodes uniformly distribute objects to buckets

- # of buckets $>>$ objects in the bucket $\Rightarrow$ lookup and store are liner time

HashCode method provides the hashcode for an object
equals and hashCode Contract

equals() and hashCode() have a specific relationship

\[ a.equals(b) \implies a.hashCode() == b.hashCode() \]

- Note that the converse is not necessarily true – a and b can have the same hash code even if they are different objects

This relationship is required by Object’s specification

- Without it, hash sets and hash maps don’t work correctly
Enforcing the Object Contract

Object's default hashCode() implementation is consistent with its default equals()

```
public class Object {
    public boolean equals(Object that) { return this == that; }
    public int hashCode() { return /* the address of this */; }
}
```

- For references a, b, if a == b, then the address of a == the address of b

But immutable objects need a different hashCode()

Duration d1 = new Duration(1,2);
Duration d2 = new Duration(1,2);
d1.equals(d2) → true
d1.hashCode() → 2392
d2.hashCode() → 4823

Using a hashmap will violate observational equality

© Robert Miller 2007
Override hashCode()

Always override hashCode() when you override equals()

➢ Your hash code should be computed from the same parts of the abstract value that equals() compares

```java
public class Duration {
    @Override
    public boolean equals(Object _that) {
        if (_that == null || !this.getClass().equals(_that.getClass()))
            return false;
        Duration that = (Duration) _that;
        return this.getSecs() == that.getSecs();
    }

    @Override
    public int hashCode() {
        return this.getSecs();
    }
}

➢ See Bloch, Effective Java, item 8
```

© Robert Miller 2007
More on Equality of Mutable Objects

Recall our definition of observational equality
➢ Two objects are equal when they cannot be distinguished by observation

With mutable objects…
➢ ... when they cannot be distinguished by observation *that doesn’t change the current state of the program*
  • i.e., by calling only observer, producer, and creator methods
  • This is often called **observational equality**, since it tests whether the two objects “look” the same, in the current state of the program
➢ ...when they cannot be distinguished by *any* observation, even state changes
  • i.e., allow calling any methods, including mutators
  • This is often called **behavioral equality**, since it tests whether the two objects will “behave” the same, in this and all future states

For immutable objects, observational and behavioral equality are identical
Consistency over Time

Consistency property

- Equality shouldn’t change over time
- if \(a.equals(b)\) now, then \(a.equals(b)\) later too

Here’s why

```java
List<String> list = makeList("hello");
Set<List<String>> set = new HashSet<List<String>>();
set.add(list);
set.contains(list) \rightarrow \text{true}

list.add("goodbye");
set.contains(list) \rightarrow \text{false!}
for (List<String> l : set) { set.contains(l) \rightarrow \text{false!} }
```
Breaking the Rep

What’s going on?

- List<String> is a mutable object

- In the standard Java implementation of collection classes like List, mutations affect the result of equals() and hashCode()
  - i.e., equals() and hashCode() implement observational equality

- When the list is first put into the HashSet, it is stored in the hash bucket corresponding to its hashCode() result at that time

- When the list is subsequently mutated, its hashCode() changes, but HashSet doesn’t realize it should be moved to a different bucket

- So it can never be found again
True Confessions

Quote from the specification of java.util.Set

Ê 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. A special case of this prohibition is that it is not permissible for a set to contain itself as an element.

Java library is inconsistent about equals()

Ê Collections use observational equality
Ê Other classes (like StringBuffer) use behavioral equality

Lesson: equals() should implement behavioral equality

Ê Mutable objects should just inherit equals() and hashCode() from Object
Ê For observational equality (whether two objects “look” the same in the current state), define a new method, e.g. similar()
The Full Object Contract

- equals must define an equivalence relation (reflexive, symmetric, transitive)

- equals must be consistent with itself (repeated calls must yield the same result unless the objects are mutated)
  - the Object contract permits equals() to implement observational equality, but that doesn’t mean you should do it

- x.equals(null) should return false (assuming x is non-null)

- equals and hashCode must be consistent with each other
  - a.equals(b) implies a.hashCode() == b.hashCode()
Summary

Equivalence relations
➢ equals() should be reflexive, symmetric, and transitive

Abstraction function as a basis for equality
➢ Two objects with the same abstract value should be equals()

Object contract
➢ a.equals(b) implies a.hashCode() == b.hashCode()

Observational vs. Behavioral vs. Referential equality

Consistency over time
➢ Using behavioral equality avoids messy issues with putting objects in collections