Reading 15: Equality

Software in 6.005

Safe from bugs	Easy to understand	Ready for change
Correct today and correct in the unknown future.	Communicating clearly with future programmers, including future you.	Designed to accommodate change without rewriting.

Objectives

- Understand equality defined in terms of the abstraction function, an equivalence relation, and observations.
- Differentiate between reference equality and object equality.
- Differentiate between strict observational and behavioral equality for mutable types.
- Understand the Object contract and be able to implement equality correctly for mutable and immutable types.

Introduction

In the previous readings we've developed a rigorous notion of *data abstraction* by creating types that are characterized by their operations, not by their representation. For an abstract data type, the *abstraction function* explains how to interpret a concrete representation value as a value of the abstract type, and we saw how the choice of abstraction function determines how to write the code implementing each of the ADT's operations.

In this reading we turn to how we define the notion of *equality* of values in a data type: the abstraction function will give us a way to cleanly define the equality operation on an ADT.

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. (This isn't strictly true of all subatomic particles, but true enough of large objects like snowflakes and baseballs and people.) So two physical objects are never truly "equal" to each other; they only have degrees of similarity.

In the world of human language, however, and in the world of mathematical concepts, you can have multiple names for the same thing. So it's natural to ask when two expressions represent the same thing: 1+2, √9, and 3 are alternative expressions for the same ideal mathematical value.

Three Ways to Regard Equality

Formally, we can regard equality in several ways.

Using an abstraction function . Recall that an abstraction function f: R → A maps concrete instances of a data type to their corresponding abstract values. To use f as a definition for equality, we would say that a equals b if and only if f(a)=f(b).

Using a relation . An *equivalence* is a relation E ⊆ T x T that is:

- reflexive: E(t,t) ∀ t ∈ T
- symmetric: E(t,u) ⇒ E(u,t)
- transitive: E(t,u) ∧ E(u,v) ⇒ E(t,v)

To use E as a definition for equality, we would say that a equals b if and only if E(a,b).

These two notions are equivalent. An equivalence relation induces an abstraction function (the relation partitions T, so f maps each element to its partition class). The relation induced by an abstraction function is an equivalence relation (check for yourself that the three properties hold).

A third way we can talk about the equality between abstract values is in terms of what an outsider (a client) can observe about them:

Using observation . We can say that two objects are equal when they cannot be distinguished by observation – every operation we can apply produces the same result for both objects. Consider the set expressions {1,2} and {2,1}. Using the observer operations available for sets, cardinality |…| and membership ∈, these expressions are indistinguishable:

- |{1,2}| = 2 and |{2,1}| = 2
- 1 ∈ {1,2} is true, and 1 ∈ {2,1} is true
- 2 ∈ {1,2} is true, and 2 ∈ {2,1} is true
- 3 ∈ {1,2} is false, and 3 ∈ {2,1} is false
- … and so on

In terms of abstract data types, "observation" means calling operations on the objects. So two objects are equal if and only if they cannot be distinguished by calling any operations of the abstract data type.

Example: Duration

Here's a simple example of an immutable ADT.

```
public class Duration {
    private final int mins;
    private final int secs;
    // rep invariant:
    //    mins >= 0, secs >= 0
    // abstraction function:
    //    represents a span of time of mins minutes and secs seconds

    /** Make a duration lasting for m minutes and s seconds. */
    public Duration(int m, int s) {
        mins = m; secs = s;
    }
    /** @return length of this duration in seconds */
    public long getLength() {
        return mins*60 + secs;
    }
}
```

Now which of the following values should be considered equal?

```
Duration d1 = new Duration (1, 2);
Duration d2 = new Duration (1, 3);
Duration d3 = new Duration (0, 62);
Duration d4 = new Duration (1, 2);
```

Think in terms of both the abstraction-function definition of equality, and the observational equality definition.

== vs. equals()

Like many languages, Java has two different operations for testing equality, with different semantics.

- The `==` operator compares references. More precisely, it tests *referential* equality. Two references are == if they point to the same storage in memory. In terms of the snapshot diagrams we've been drawing, two references are `==` if their arrows point to the same object bubble.
- The `equals()` operation compares object contents – in other words, *object* equality, in the sense that we've been talking about in this reading. The equals operation has to be defined appropriately for every abstract data type.

For comparison, here are the equality operators in several languages:

	referential equality	*object equality*
Java	`==`	`equals()`

Objective C	`==`	`isEqual:`
C#	`==`	`Equals()`
Python	`is`	`==`
Javascript	`==`	n/a

Note that `==` unfortunately flips its meaning between Java and Python. Don't let that confuse you: `==` in Java just tests reference identity, it doesn't compare object contents.

As programmers in any of these languages, we can't change the meaning of the referential equality operator. In Java, `==` always means referential equality. But when we define a new data type, it's our responsibility to decide what object equality means for values of the data type, and implement the `equals()` operation appropriately.

Equality of Immutable Types

The `equals()` method is defined by `Object`, and its default implementation looks like this:

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

In other words, the default meaning of `equals()` is the same as referential equality. For immutable data types, this is almost always wrong. So you have to **override** the `equals()` method, replacing it with your own implementation.

Here's our first try for `Duration`:

```
public class Duration {
    ...
    // Problematic definition of equals()
    public boolean equals(Duration that) {
        return this.getLength() == that.getLength();
    }
}
```

There's a subtle problem here. Why doesn't this work? Let's try this code:

```
Duration d1 = new Duration (1, 2);
Duration d2 = new Duration (1, 2);
Object o2 = d2;
d1.equals(d2) → true
d1.equals(o2) → false
```

You can see this code in action. You'll see that even though `d2` and `o2` end up referring to the very same object in memory, you still get different results for them from `equals()`.

What's going on? It turns out that `Duration` has **overloaded** the `equals()` method, because the method signature was not identical to `Object`'s. We actually have two `equals()` methods in `Duration`: an implicit `equals(Object)` inherited from `Object`, and the new `equals(Duration)`.

```
public class Duration extends Object {
    // explicit method that we declared:
    public boolean equals (Duration that) {
        return this.getLength() == that.getLength();
    }
    // implicit method inherited from Object:
    public boolean equals (Object that) {
        return this == that;
    }
}
```

We've seen overloading since the very beginning of the course in static checking. Recall from the Java Tutorials that the compiler selects between overloaded operations using the compile-time type of the parameters. For example, when you use the `/` operator, the compiler chooses either integer division or float division based on whether the arguments are ints or floats. The same compile-time selection happens here. If we pass an `Object` reference, as in `d1.equals(o2)`, we end up calling the `equals(Object)` implementation. If we pass a `Duration` reference, as in `d1.equals(d2)`, we end up calling the `equals(Duration)` version. This happens even though `o2` and `d2` both point to the same object at runtime! Equality has become inconsistent.

It's easy to make a mistake in the method signature, and overload a method when you meant to override it. This is such a common error that Java has a language feature, the annotation `@Override`, which you should use whenever your intention is to override a method in your superclass. With this annotation, the Java compiler will check that a method with the same signature actually exists in the superclass, and give you a compiler error if you've made a mistake in the signature.

So here's the right way to implement `Duration`'s `equals()` method:

```
@Override
public boolean equals (Object thatObject) {
    if (!(thatObject instanceof Duration)) return false;
    Duration thatDuration = (Duration) thatObject;
    return this.getLength() == thatDuration.getLength();
}
```

This fixes the problem:

```
Duration d1 = new Duration(1, 2);
Duration d2 = new Duration(1, 2);
Object o2 = d2;
d1.equals(d2) → true
d1.equals(o2) → true
```

You can see this code in action in the Online Python Tutor.

instanceof

The `instanceof` operator tests whether an object is an instance of a particular type. Using `instanceof` is dynamic type checking, not the static type checking we vastly prefer. In general, using `instanceof` in object-oriented programming is a bad smell. In 6.005 — and this is another of our rules that holds true in most good Java programming — **`instanceof` is disallowed anywhere except for implementing `equals`**. This prohibition also includes other ways of inspecting objects' runtime types. For example, `getClass` is also disallowed.

We'll see examples of when you might be tempted to use `instanceof`, and how to write alternatives that are safer from bugs and more ready for change, in a future reading.

The Object Contract

The specification of the `Object` class is so important that it is often referred to as *the `Object` Contract*. The contract can be found in the method specifications for the `Object` class. Here we will focus on the contract for `equals`. When you override the `equals` method, you must adhere to its general contract. It states that:

- `equals` must define an equivalence relation – that is, a relation that is reflexive, symmetric, and transitive;
- `equals` must be consistent: repeated calls to the method must yield the same result provided no information used in `equals` comparisons on the

object is modified;
- for a non-null reference `x`, `x.equals(null)` should return false;
- `hashCode` must produce the same result for two objects that are deemed equal by the `equals` method.

Breaking the Equivalence Relation

Let's start with the equivalence relation. We have to make sure that the definition of equality implemented by `equals()` is actually an equivalence relation as defined earlier: reflexive, symmetric, and transitive. If it isn't, then operations that depend on equality (like sets, searching) will behave erratically and unpredictably. You don't want to program with a data type in which sometimes `a` equals `b`, but `b` doesn't equal `a`. Subtle and painful bugs will result.

Here's an example of how an innocent attempt to make equality more flexible can go wrong. Suppose we wanted to allow for a tolerance in comparing `Duration` objects, because different computers may have slightly unsynchronized clocks:

```
private static final int CLOCK_SKEW = 5; // seconds

@Override
public boolean equals (Object thatObject) {
    if (!(thatObject instanceof Duration)) return false;
    Duration thatDuration = (Duration) thatObject;
    return Math.abs(this.getLength() - thatDuration.getLength()) <= CLOCK_SKEW;
}
```

Which property of the equivalence relation is violated?

Breaking Hash Tables

To understand the part of the contract relating to the `hashCode` method, you'll need to have some idea of how hash tables work. Two very common collection implementations, `HashSet` and `HashMap`, use a hash table data structure, and depend on the `hashCode` method to be implemented correctly for the objects stored in the set and used as keys in the map.

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. Keys don't have to be ordered, or have any particular property, except for offering `equals` and `hashCode`.

Here's how a hash table works. It contains an array that is initialized to a size corresponding to the number of elements that we expect to be inserted. When a key and a value are presented for insertion, we compute the hashcode of the key, and convert it into an index in the array's range (e.g., by a modulo division). The value is then inserted at that index.

The rep invariant of a hash table includes the fundamental constraint that keys are in the slots determined by their hash codes.

Hashcodes are designed so that the keys will be spread evenly over the indices. But occasionally a conflict occurs, and two keys are placed at the same index. So rather than holding a single value at an index, a hash table actually holds a list of key/value pairs, usually called a *hash bucket*. A key/value pair is implemented in Java simply as an object with two fields. On insertion, you add a pair to the list in the array slot determined by the hash code. For lookup, you hash the key, find the right slot, and then examine each of the pairs until one is found whose key equals the query key.

Now it should be clear why the `Object` contract requires equal objects to have the same hashcode. If two equal objects had distinct hashcodes, they might be placed in different slots. So if you attempt to lookup a value using a key equal to the one with which it was inserted, the lookup may fail.

`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 memory address of this */; }
}
```

For references `a` and `b`, if `a == b`, then the address of a `==` the address of b. So the `Object` contract is satisfied.

But immutable objects need a different implementation of `hashCode()`. For `Duration`, since we haven't overridden the default `hashCode()` yet, we're currently breaking the `Object` contract:

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

`d1` and `d2` are `equal()`, but they have different hash codes. So we need to fix that.

A simple and drastic way to ensure that the contract is met is for `hashCode` to always return some constant value, so every object's hash code is the same. This satisfies the `Object` contract, but it would have a disastrous performance effect, since every key will be stored in the same slot, and every lookup will degenerate to a linear search along a long list.

The standard way to construct a more reasonable hash code that still satisfies the contract is to compute a hash code for each component of the object that is used in the determination of equality (usually by calling the `hashCode` method of each component), and then combining these, throwing in a few arithmetic operations. For `Duration`, this is easy, because the abstract value of the class is already an integer value:

```
@Override
public int hashCode() {
    return (int) getLength();
}
```

Josh Bloch's fantastic book, *Effective Java*, explains this issue in more detail, and gives some strategies for writing decent hash code functions. The advice is summarized in a good StackOverflow post. Recent versions of Java now have a utility method `Objects.hash()` that makes it easier to implement a hash code involving multiple fields.

Note, however, that as long as you satisfy the requirement that equal objects have the same hash code value, then the particular hashing technique you use doesn't make a difference to the correctness of your code. It may affect its performance, by creating unnecessary collisions between different objects, but even a poorly-performing hash function is better than one that breaks the contract.

Most crucially, note that if you don't override `hashCode` at all, you'll get the one from `Object`, which is based on the address of the object. If you have overridden `equals`, this will mean that you will have almost certainly violated the contract. So as a general rule:

> **Always override `hashCode` when you override `equals`.**

Many years ago in (a precursor to 6.005 confusingly numbered) 6.170, a student spent hours tracking down a bug in a project that amounted to nothing more than misspelling `hashCode` as `hashcode`. This created a new method that didn't override the `hashCode` method of `Object` at all, and strange things happened. Use `@Override`!

Equality of Mutable Types

We've been focusing on equality of immutable objects so far in this reading. What about mutable objects?

Recall our definition: two objects are equal when they cannot be distinguished by observation. With mutable objects, there are two ways to interpret this:

- when they cannot be distinguished by observation *that doesn't change the state of the objects*, i.e., by calling only observer, producer, and creator methods. This is often strictly 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. This interpretation allows calling any methods on the two objects, 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, because there aren't any mutator methods.

For mutable objects, it's tempting to implement strict observational equality. Java uses observational equality for most of its mutable data types, in fact. If two distinct `List` objects contain the same sequence of elements, then `equals()` reports that they are equal.

But using observational equality leads to subtle bugs, and in fact allows us to easily break the rep invariants of other collection data structures. Suppose we make a `List`, and then drop it into a `Set`:

```
List<String> list = new ArrayList<>();
list.add("a");

Set<List<String>> set = new HashSet<List<String>>();
set.add(list);
```

We can check that the set contains the list we put in it, and it does:

```
set.contains(list) → true
```

But now we mutate the list:

```
list.add("goodbye");
```

And it no longer appears in the set!

```
set.contains(list) → false!
```

It's worse than that, in fact: when we iterate over the members of the set, we still find the list in there, but `contains()` says it's not there!

```
for (List<String> l : set) {
    set.contains(l) → false!
}
```

If the set's own iterator and its own `contains()` method disagree about whether an element is in the set, then the set clearly is broken. You can see this code in action on Online Python Tutor.

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()`. 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.

When `equals()` and `hashCode()` can be affected by mutation, we can break the rep invariant of a hash table that uses that object as a key.

Here's a telling quote from the specification of `java.util.Set`:

> 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.

The Java library is unfortunately inconsistent about its interpretation of `equals()` for mutable classes. Collections use observational equality, but other mutable classes (like `StringBuilder`) use behavioral equality.

The lesson we should draw from this example is that **`equals()` should implement behavioral equality**. In general, that means that two references should be `equals()` if and only if they are aliases for the same object. So mutable objects should just inherit `equals()` and `hashCode()` from `Object`. For clients that need a notion of observational equality (whether two mutable objects "look" the same in the current state), it's better to define a new method, e.g., `similar()`.

The Final Rule for equals() and hashCode()

For immutable types:

- `equals()` should compare abstract values. This is the same as saying `equals()` should provide behavioral equality.
- `hashCode()` should map the abstract value to an integer.

So immutable types must override both `equals()` and `hashCode()`.

For mutable types:

- `equals()` should compare references, just like `==`. Again, this is the same as saying `equals()` should provide behavioral equality.
- `hashCode()` should map the reference into an integer.

So mutable types should not override `equals()` and `hashCode()` at all, and should simply use the default implementations provided by `Object`. Java doesn't follow this rule for its collections, unfortunately, leading to the pitfalls that we saw above.

Autoboxing and Equality

One more instructive pitfall in Java. We've talked about primitive types and their object type equivalents – for example, `int` and `Integer`. The object type implements `equals()` in the correct way, so that if you create two `Integer` objects with the same value, they'll be `equals()` to each other:

```
Integer x = new Integer(3);
Integer y = new Integer(3);
x.equals(y) → true
```

But there's a subtle problem here; `==` is overloaded. For reference types like `Integer`, it implements referential equality:

```
x == y // returns false
```

But for primitive types like `int`, `==` implements behavioral equality:

```
(int)x == (int)y // returns true
```

So you can't really use `Integer` interchangeably with `int`. The fact that Java automatically converts between `int` and `Integer` (this is called *autoboxing* and *autounboxing*) can lead to subtle bugs! You have to be aware what the compile-time types of your expressions are. Consider this:

```
Map<String, Integer> a = new HashMap(), b = new HashMap();
a.put("c", 130); // put ints into the map
b.put("c", 130);
a.get("c") == b.get("c") → ?? // what do we get out of the map?
```

You can see this code in action on Online Python Tutor.

Summary

- Equality should be an equivalence relation (reflexive, symmetric, transitive).
- Equality and hash code must be consistent with each other, so that data structures that use hash tables (like `HashSet` and `HashMap`) work properly.
- The abstraction function is the basis for equality in immutable data types.
- Reference equality is the basis for equality in mutable data types; this is the only way to ensure consistency over time and avoid breaking rep invariants of hash tables.

Equality is one part of implementing an abstract data type, and we've already seen how important ADTs are to achieving our three primary objectives. Let's look at equality in particular:

- **Safe from bugs**. Correct implementation of equality and hash codes is necessary for use with collection data types like sets and maps. It's also highly desirable for writing tests. Since every object in Java inherits the `Object` implementations, immutable types must override them.
- **Easy to understand**. Clients and other programmers who read our specs will expect our types to implement an appropriate equality operation, and will be surprised and confused if we do not.
- **Ready for change**. Correctly-implemented equality for *immutable* types separates equality of reference from equality of abstract value, hiding from clients our decisions about whether values are shared. Choosing behavioral rather than observational equality for *mutable* types helps avoid

unexpected aliasing bugs.