Equality and mutability

6.170 Lecture 17

October 24, 2000

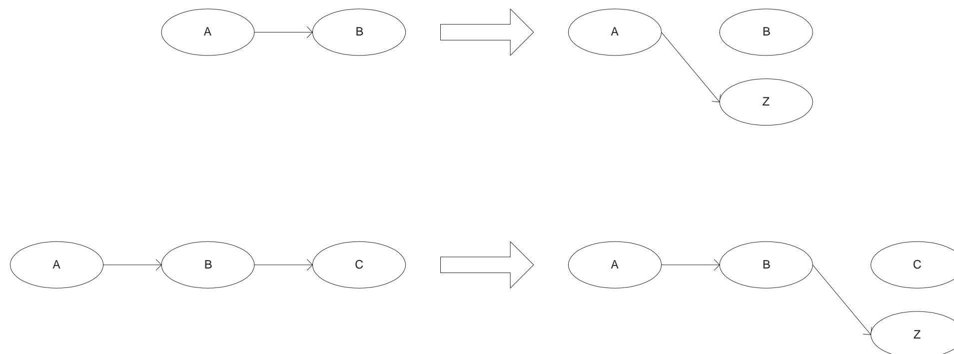
Contents

1 Immutability **167**
2 Abstraction function refresher **168**
3 Behavioral equivalence **170**
4 Abstract objects and abstract values **171**
5 Equality testing **172**
6 Relationships among tests; hashtable keys **173**

Reading: Section 5.4 of *Program Development in Java* by Barbara Liskov

1 Immutability

A *mutable* abstraction can change, taking on new abstract values. (Here, *abstraction* means an abstract object — an instance of an abstraction that models immutable things.) The abstract value can change by making a field refer to a different object or by mutating an object referred to by a field, as illustrated in the figure. Either of these mutations changes the abstract value of `A`.



An *immutable* abstraction never changes: it always has the same abstract value. Similarly, an immutable representation cannot change; typically either the representation is in terms of an immutable datatype or the representation is never modified. (There may be beneficial side effects — see book section 5.6 — but any such are invisible to the client. That is one way that a mutable representation can be used for an immutable datatype.)

The key reasons to use immutable representations and datatypes are efficiency and simplicity. An immutable representation may use less space for multiple objects than a mutable representation, because sharing is legal among immutable datatypes. This permits both sharing a representation among multiple objects and interning, which arranges to create only a single object representing an

abstract value. (Interning can also save time by permitting use of Java's `==` operator rather than a call to the `equals` method.) Sharing can be problematic for mutable representations; making the representation immutable eliminates this worry and may simplify code and ease reasoning about it.

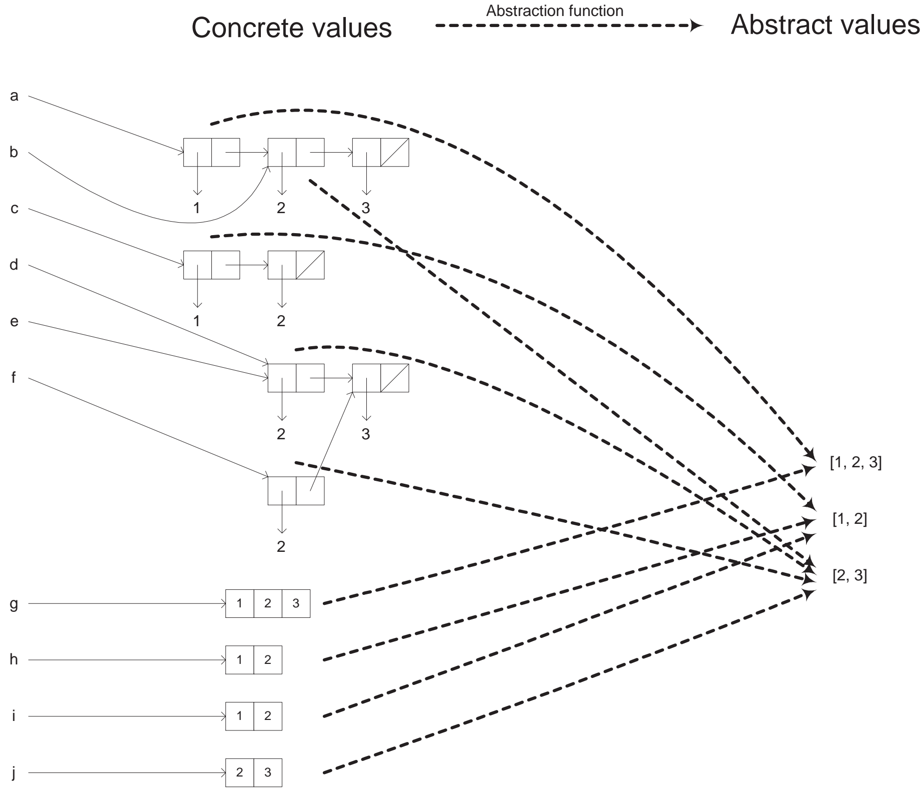
An immutable representation may also be more time-efficient. It is always safe to have multiple references to a given immutable object, since the object cannot change. Making a copy is as simple as returning a new reference to the existing object. Augmenting a data structure can be very simple, reusing the old object and simply adding a small amount of new information. (Consider Scheme's `cons` procedure; it reuses existing data structure and so runs in constant time. The same should be true of your implementations of `Path` (on problem set 2 and the final project). Immutable objects can be passed to unknown methods with impunity; those methods cannot modify them, so the client need not protect itself by making a copy first.

On the other hand, an immutable representation may also be less time-efficient. An update can force the entire object to be recopied, because the existing object cannot be updated in place. This is particularly wasteful when the old version is thrown away and only the new one is used thereafter.

When deciding whether to make a data structure mutable or immutable, consider the expected use, and arrange for operations to be convenient and efficient for clients in the usual case. The choice of mutability determines whether an abstraction provides producer-style or mutator-style operations; changing that decision after the fact is difficult and time-consuming, requiring changes to all clients.

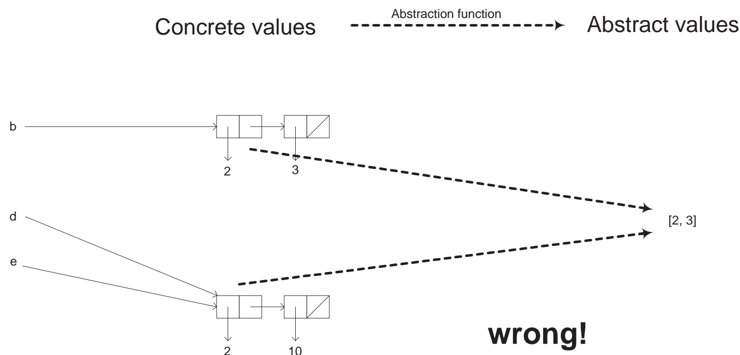
2 Abstraction function refresher

Recall that an abstraction function maps concrete values to abstract values — for instance, `LinkedList` or `Array` values to (mathematical) lists.



Adding an element to an immutable list results in a new concrete value and possibly a new abstract value. Executing `a.addFirst(new Integer(0))` results in creation of a new list link (a cons cell) which is mapped to new abstract value [0, 1, 2, 3]. Abstract immutable values cannot be modified; the old abstract value [1, 2, 3] remains as is (in this case, because it is also the abstract value for `g`) or is removed (if there are no more references to it).

Mutability clouds this picture, however. (From here on we will focus on the relationship between `b`, `d`, and `e`; sharing of portions of representations, as occurs elsewhere in the diagram, is a complicated issue which we will defer for now.) How should a modification to a concrete object be reflected in the figure? For instance, suppose that the second element of list `d` is changed to 10:



The diagram must be updated to reflect the fact that the abstract value of `e` has changed, but that of `b` has not. In other words, a simple, local mutation does not have a simple, local effect on the diagram. This is undesirable because we would like to be able to perform simple operations

either at the abstract or the concrete level and understand how they affect the rest of the system, at either level.

The problem is that variables `b`, `d`, and `e` represent the same abstract value, but they do not represent the same abstract object (nor the same concrete object). In particular, changes to `d` or `e` are visible to the other, but not to `b`. This violates the intuition behind an abstraction function: there are two concrete objects with the same abstract value but different behavior.

3 Behavioral equivalence

When two objects are *behaviorally equivalent*, then either one can be used in place of the other. That is, in any context (say, a computer program or fragment), uses of one of the objects can be replaced by uses of the other one without any change in behavior. This principle is sometimes known as *Leibniz's Law* or *substitution of equals for equals*: if two things are really equal, we should not be able to distinguish between them.

To determine whether two objects are behaviorally equivalent, it is sufficient to ask whether there is a context which distinguishes the objects — that is, which behaves differently depending on which objects is used.

Suppose we wish to determine whether (global) arrays `a` and `b` are behaviorally equivalent:

```
class TestArrays {
  int[] a = { 1, 2, 3 };
  int[] b = { 1, 2, 3 };

  int test1(int[] x) {
    return x.length + x[1];
  }

  int test2(int[] x) {
    b[1] = 0;
    return x.length + x[1];
  }
}
```

The two arrays behave identically when passed to `test1`, which returns 5 when executed on either array. However, `test2` distinguishes the arrays; it returns 5 for `a` but 3 for `b`. Thus, `a` and `b` are not behaviorally equivalent.

Another test distinguishes the two arrays if they are in the same context, even if they are not directly available to the test routines. Consider the context `test3(␣, a, b)`, where `test3` is defined as follows:

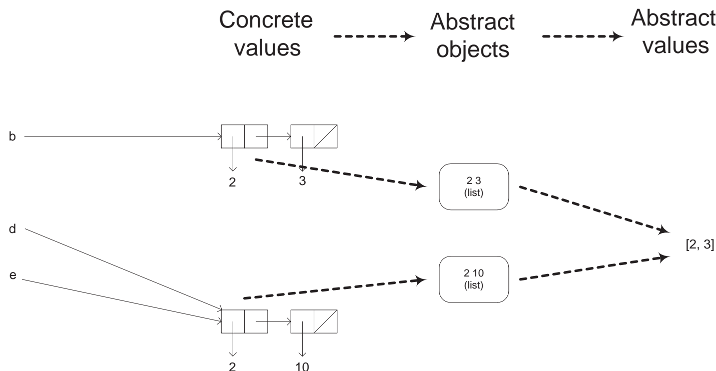
```
int test3(int[] x, int[] y, int[] z) {
  z[1] = 0;
  return x.length + x[1];
}
```

Because `test3(a, a, b)` and `test3(b, a, b)` have different behavior, `a` and `b` are not behaviorally equivalent.

The same sort of test can be done for a variety of datatypes, including user-defined ones.

4 Abstract objects and abstract values

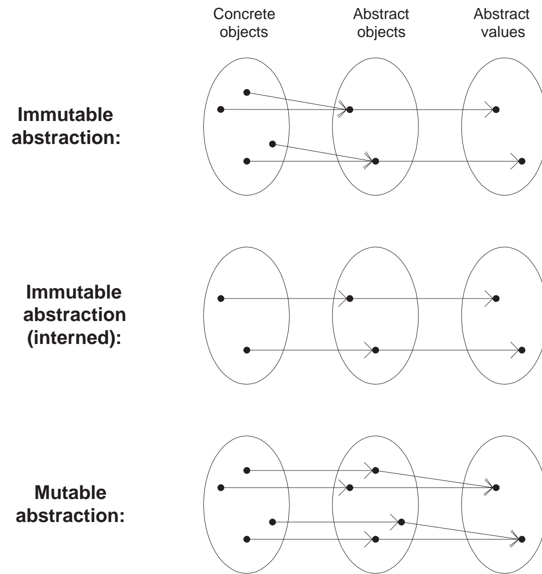
To formalize the notion of behavioral equivalence, we will break the abstraction function into two parts and add a level for abstract objects to the diagram.



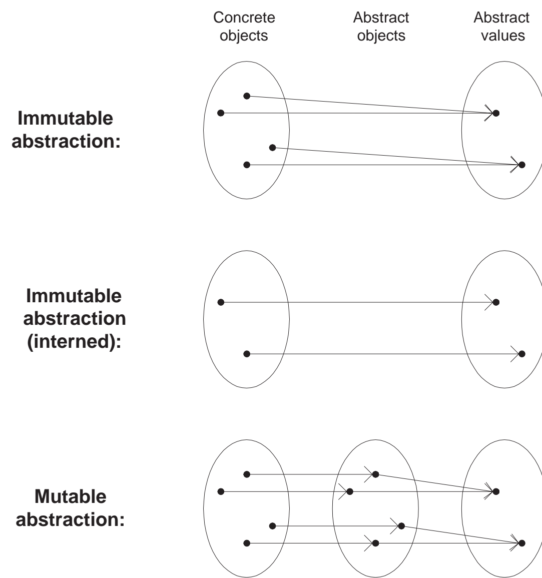
With a two-part abstraction function, it takes two steps to determine the abstract value for a particular concrete value. However, the figure now clearly indicates sharing relationships (much as a snapshot (object diagram) does, and as an object model can in many cases (though sometimes it requires extra textual constraints). More importantly, state changes (whether creating a new value or modifying an old one) have a simple interpretation in the figure.

Behavioral equivalence applies to abstract objects; two abstract objects are different if they can be distinguished by a sequence of operations. In general, an immutable type may have many instances of a concrete representation, but there is only one abstract object for a particular value; each abstract object represents a different abstract value. Interning arranges that there is only one concrete object for each abstract object. Mutable objects typically have only one concrete object per abstract object, but multiple abstract objects may represent the same abstract value. (It is possible for multiple concrete objects to represent the same abstract object; however, such an implementation is very intricate and, for that reason, rarely done.)

The following figure illustrates the relationship among concrete objects, abstract objects, and abstract values for the cases of immutable abstractions, interned immutable abstractions, and mutable abstractions. The ovals enclose collections of objects of the given type.



Actually, immutable objects should only be considered in terms of their values, never in terms of object identity (at the concrete or abstract levels), so the picture can be more simply drawn in the following way.



This captures the intuition that it is only mutation that complicates understanding of abstraction functions and equality.

5 Equality testing

With the introduction of abstract objects, there are three possibilities for the semantics of comparison. Two objects can be tested for being the same concrete object (this is called *object equality* or *object identity*), for representing the same abstract object, or for representing the same abstract value. (It is also possible to test whether two potentially different concrete objects have the same

contents. This may be equivalent to one of the other tests; when it is not, it is generally not very interesting.)

The Java `==` operator tests object equality. Unfortunately, it does not define operators that are specified to report abstract object equality or abstract value equality. This is unfortunate, because clients sometimes wish to use a particular variety of equality, but without a standard name, it is impossible to write code that will work for all types.

Java does specify a `equals` method; however, the specification of `equals` is carefully written to allow implementations to check object equality, abstract object equality, or abstract value equality. Thus, clients can't be sure just what they are getting when they call `equals`. Reading the specifications of `equals` for the Java 1.3 collection classes clarifies that `equals` is intended to check abstract value equality in those cases. However, for other classes, Java's `equals` still checks abstract object equality (and collections used abstract object equality for `equals` in Java 1.2).

Java doesn't provide an operation for checking abstract object equality. Typically it isn't nearly as interesting as the other varieties and is less frequently used. Additionally, if you know whether you are manipulating an immutable or a mutable object, you can test abstract object equality (which is the same as either abstract value equality or object equality, respectively, in the usual case) by using either `==` or `equals`. While this does work, it has a few disadvantages: clients may not know or care whether an abstraction is immutable or mutable; an abstraction may change from one to the other; and many-to-one mappings from concrete objects to abstract objects are possible.

The Liskov text *Program Development in Java* proposes a different scheme, with `equals` for abstract object equality and `similar` for abstract value equality. (More specifically, `equals` indicates behavioral equivalence in all contexts and `similar` indicates behavioral equivalence in mutation-free contexts.)

	(concrete) object equality	abstract object equality	abstract value equality
Java	`==`		`equals`
Liskov	`==`	`equals`	`similar`

The book's proposal is a good one and is internally self-consistent; the use of three different names makes design decisions clearer. Unfortunately, the text's approach is prescriptive rather than descriptive: it tells you how you ought to write programs and how the Java authors should have designed their system rather than describing the way that Java (including libraries and much other code) actually does work. The rest of the world has not yet come around to the `==/equals/similar` way of thinking, and you cannot change built-in classes or the expectations of other clients of your code. So beware when you care about interoperability with code written by people who have not taken 6.170. Remember that `equals` means something different to them and write your code accordingly.

In 6.170, you should adhere to the Liskov convention. One implication of this is that if two objects are ever `equals`, then the are always `equals`.

6 Relationships among tests; hashtable keys

The three equality tests satisfy the following implication relationships:

$$\begin{array}{c}\text{object}\\ \text{equality}\end{array} \Longrightarrow \begin{array}{c}\text{abstract}\\ \text{object}\\ \text{equality}\end{array} \Longrightarrow \begin{array}{c}\text{abstract}\\ \text{value}\\ \text{equality}\end{array}$$

Thus, implementations should also obey those implications.

Java furthermore demands that

$$\texttt{a==b} \implies \texttt{a.equals(b)} \implies \texttt{a.hashCode()==b.hashCode()}$$

Java does not check these implications, but incorrect behavior may result if they are violated.

Java uses hash codes as a quick-and-dirty test for equality: it may be cheaper to compute and compare hash codes (particularly if they can be computed once, then cached for future use) than to perform full equality tests. If the hash codes are different, then the objects are certainly different and no more checking need be done. If the hash codes are the same, then a complete equality check must be performed.

Because of the above implication, it is incorrect for `equals` to test abstract value equality and `hashCode` to work over the representation or the identity of the abstract object. Both should operate at the same level of abstraction. The fact that `equals` operates over abstract values (in the Java scheme, not the Liskov scheme) has an unfortunate consequence: since the value represented by an object can change without the object changing, an object's hash code can change. When hashtable keys are modified in such a way as to change their hash codes, hash tables can operate incorrectly, for instance reporting that the objects are no longer members.

The following code can throw an error:

```
ArrayList al = new ArrayList();
HashSet h = new HashSet();
h.put(al);
al.add(new Integer(0));
if (! h.contains(al)) {
  throw new Error("Element has disappeared from HashSet");
}
```

If you are lucky, it will throw an error every time, but it is not guaranteed to do so.

This is an example of *representation exposure*. Internally, `HashSet` objects contain a collection of buckets. Each object is put in a bucket according to its hash code, and to find a particular object, only one bucket, rather than the entire set, need be searched. The `HashSet` representation invariant states, "if key `k` is in this set, it is in the bucket indexed by `k.hashCode()`". Code that changes the hash code violates the representation invariant. As is always the case when there is representation exposure, we must perform global reasoning (i.e., verify that the set elements are never modified) to ensure that the representation invariant is not violated. If there is no representation exposure, then local reasoning suffices.

The moral of this story is: *never mutate hashtable (or hashset) keys.* It is best to use immutable keys; if you must use mutable keys, be sure that they are not side-effected by your program or any code it calls.