L11: Equality

Today

- Review of visitor pattern, step-by-step
- Overloading vs. overriding
- Equality
- the Object contract

Recommended Reading

From David Flanagan, Java in a Nutshell, 5th edition
(available through MIT Libraries O'Reilly Safari)

- Sec 3.5 Subclasses and inheritance (discusses runtime method dispatch and overriding)
- Sec 3.10 Nested types (discusses anonymous classes)

From Joshua Bloch, Effective Java

- Item 8: Obey the general contract when overriding equals
- Item 9: Always override hashCode when you override equals

Visitor Pattern, Revisited

Let's look at another example of a visitor pattern, and trace through what it does at runtime so we can understand it better. Along the way we'll also touch on an important issue in object-oriented programming languages: how the language determines which method to call. For this example, we'll see how Java uses actual (runtime) types of objects, and declared (compile-time) types of variables, to make those decisions.

To keep the example simple, we'll use a recursive datatypewith only two variants, our friend ImList.

\[
\text{ImList}\langle E \rangle = \text{Empty} + \text{Cons(first:E, rest:ImList}\langle E \rangle)
\]

In fact we can simplify the ImList interface to the point where its only method is a visitor acceptor:

```java
// ImList represents an immutable sequence of elements of type E
public interface ImList\langle E \rangle {
    public <R> R accept(ImListVisitor\langle E,R \rangle visitor);
}
```

The visitor pattern will let clients unpack and operate on the list, so it's the only observer method we really need. Furthermore, we'll see that the visitor pattern’s static checking actually prevents mistakes like calling first() on an empty list, which would otherwise be possible if the ImList interface had first() and rest() observer methods.

The ImListVisitor interface has a method for each of the variants of ImList:

```java
// ImListVisitor represents a function ImList\langle E \rangle -> R
public static interface ImListVisitor\langle E,R \rangle {
    public R on(Empty\langle E \rangle nil);
    public R on(Cons\langle E \rangle list);
}
```
Note that the visitor needs two type parameters now – E for the type of the elements of the list, and R for the return type of the visitor. We’ve also chosen to overload the method name on(), rather than putting the name of the variant in the method name (onEmpty and onCons). Java decides which of the two on() methods to call depending on the compile-time type of the parameter passed to on().

Now we’ll implement the two variant classes that implement ImList:

```java
// Empty represents the empty list
public class Empty<E> implements ImList<E> {
    public Empty() { }
    public <R> R accept(ImListVisitor<E,R> visitor) {
        return visitor.on(this);
    }
}

// Cons represents an element followed by the rest of the list
public class Cons<E> implements ImList<E> {
    public final E first;
    public final ImList<E> rest;
    public Cons(E first, ImList<E> rest) {
        this.first = first;
        this.rest = rest;
    }
    public <R> R accept(ImListVisitor<E,R> visitor) {
        return visitor.on(this);
    }
}
```

That’s the datatype. Now let’s define an operation over the datatype:

```java
size: ImList<E> → int
size(Empty) = 0
size(Cons(first,rest)) = 1 + size(rest)
```

And implement it using a visitor:

```java
// size: return the number of elements in a list
public static <E> int size(ImList<E> list) {
    return list.accept(new ImListVisitor<E,Integer>() {
        public Integer on(Empty<E> nil) { return 0; }
        public Integer on(Cons<E> list) { return 1 + size(list.rest); }
    });
}
```

Look carefully at this code. It’s constructing a visitor object as an instance of an anonymous class, created on the spot to implement the ImListVisitor interface. (See the readings for this lecture for more information about the syntax and semantics of anonymous classes.) The visitor’s return type is specified as Integer, because size() is supposed to return int. (Type parameters in Java must be satisfied with object types, so primitive types have to be replaced with their object type equivalents.) Finally, the two on() methods implement the two cases of the size operation.
Notice now that, even in this simple code, we have two different method implementations for accept() and two different method implementations for on(). Let’s keep it straight by drawing an inheritance hierarchy for this code:

[Diagrams showing inheritance hierarchy]

Walking Along with a Visitor

Now let’s walk through some client code that calls the size operation, so we can see how the visitor moves through the ImList data structure, and how accept() and on() cooperate to produce the final result. Here’s the code we’ll use:

```java
public static void main(String[] args) {
    ImList<Integer> nil = new Empty<Integer>();
    ImList<Integer> x = new Cons<Integer>(5, nil);
    int n = size(x);
    System.out.println(n);
}
```

After the first two lines of main() have executed, a snapshot diagram of the state of the system looks like this:

[Snapshot diagram showing ImList state]

Here we’ve drawn not only the heap (allocated object instances like Cons and Empty, shown on the right) but also the call stack, the stack that the Java runtime uses to keep track of which method calls are still waiting for results (shown on the left). Right now, only the call to main() is on the call stack.

Local variables and method parameters, like nil and x in main, are part of the stack, since each fresh method call gets fresh local variables. But fields of objects, like first and rest, are found in the heap. (For more on this diagram notation, see the Lecture 1 addendum about snapshot diagrams.)

We’ve also written the declared type (compile-time type) of the nil and x variables. This reminds us what Java knows about the variable at compile time. Would Java allow us to write x.first, or would that cause a static type error? What about Python? Would it be legal there?

Now the call to size(x) occurs, and we get a new frame on the call stack:
The body of size() makes a new instance of its anonymous ImListVisitor, and then passes it to accept:

Which implementation of accept() is being called here? The code in size() called list.accept(), but its list variable has type ImList, which is an interface, and doesn’t actually implement the method. The rule is that **var.method() uses the runtime type of the object that var points to determine which implementation of method() is actually called.** This is called runtime type dispatch, since it’s using the runtime type of the object to “dispatch” the call to the method that should handle it. Since list points to a Cons object, the call list.accept() is handled by Cons.accept(). (What about Python? Does the same rule work there?)

Note also that the stack frame for accept(), unlike the other stack frames, includes a **this** variable. (What does that variable mean? Why didn’t the stack frames for main() and size() have a this variable? If you don’t understand these questions, go back and read Understanding Instance and Class Members in the Java tutorial.) A crucial fact here is that the compile-time type of this is Cons, because we’re now inside Cons.accept(). Java now has proof, which it can use even at compile time, that the ImList we’re using is a Cons object.

The Cons.accept() method now calls visitor.on(this). The runtime type dispatch rule says we should look at the runtime type of the object that visitor points to, which is the anonymous ImListVisitor class. So far so good. But now we have an ambiguity: the visitor class has two implementations for the on() method. on() is **overloaded** within the same class. Which one is called? The rule for overloaded methods is that a **call to an overloaded method method(x, y, z) uses the compile-time types of the expressions passed as parameters x, y, z to determine which overloaded method to call.** Since the compile-time type of this is Cons, visitor.on(this) is handled by on(Cons).

The key difference here is that the choice of method for overloading is always determined at compile time, while the choice of method for interface implementation is always determined at runtime.

Overloading appears in many contexts in programming languages, by the way, not always in named methods like on(). For example, the + operator is overloaded in Java; it combines two integers differently than it combines two Strings, or a String and an integer. The types of its operands choose a different implementation of + at compile time. Compare these:

\[ 5 + 5 \]
“5” + “5”

5 + “5”

5 + (Object) “5” // what does this mean? Does Java allow it?

Now our snapshot looks like this:

We’re now inside the Cons-specific case of the size function. Notice what’s happened here: the list variable that we originally had in size(), which had an abstract compile-time type of ImList, has been narrowed down to a list variable with the more specific compile-time type of Cons. Java now has proof that the list is actually a Cons object. We didn’t use runtime checks like instanceof or downcasting to make that happen. Instead the partnership of accept() and on(), using runtime type dispatch, was responsible for checking the runtime type of the ImList object and getting us into the right piece of code for it, with a variable whose compile-time type now gives us the ability to access first and rest, with safety that Java can guarantee at compile time.

Continuing, the Cons case of size makes a recursive call to size() on the rest of the list, so we get another frame on top of the stack:

size() creates another visitor instance and hands it to accept():
Which accept() is called here? Why?
Then accept() calls on():

Which on() is called here?
Finally we’re in the Empty case of the size function, which returns 0. We return from each of the methods on the stack until we finally get back to main() with our result.

Note that this is a bit wasteful – how many instances of the visitor will size() end up creating for an n-element list? Do they really need to be distinct? How might we change size() to avoid all this redundancy?
Can you define these functions using visitors?

- `append(list1, list2)`: returns a list with the elements of list1 followed by the elements of list2
- `nth(n, list)`: returns the nth element of list
- `sum(List<Integer> list)`: returns the sum of the elements of the list
- `reverse(list)`: returns a list containing the elements of list in the reverse order

**Static Checking Revisited**

In practice, the visitor pattern is more often used for recursive tree-shaped datatypes, like the abstract syntax trees we saw in the previous two lectures. It isn’t typically used for list-like datatypes like ImList. Iterators and map/filter/reduce (also called list comprehensions in Python) are more common design patterns for writing functions over list-like datatypes. We’ll see map/filter/reduce in a future lecture.

But one reason we’re looking at the visitor pattern closely, even in this unusual setting, is that it offers a powerful demonstration of static checking. Code that would have needed runtime type checks, to make sure you’re not using a method or field that isn’t defined, has been replaced by code that can be proven safe from those bugs at compile time.

Without the visitor pattern (or in a dynamically-typed language like Python), the size method might have been written like this:

```java
if (list instanceof Cons) {
    return 1 + size(((Cons) list).rest);
} else {
    return 0;
}
```

What if we had forgotten the instanceof test – or gotten it wrong and written `list instanceof Empty` instead? Java would happily compile the code, and the bug would lie waiting until runtime.

The idea of static checking is to find and eliminate large classes of bugs in advance, before the program is even run. Static type checking eliminates type errors, the kinds of errors where an operation is used on the wrong kinds of values. Using instanceof and downcasting cuts around static checking, and leaves you vulnerable to type errors at runtime.

Static checking makes it much easier to build and maintain large systems. Most languages used for large-scale application development or operating system development – including C, C++, Java, Objective C – have some degree of static checking. In fact, when great software engineers find themselves building a large system in a dynamically-typed language, they may respond by putting static checking on top of it! Case in point: Google has an enormous amount of Javascript code, including for some very large and complex systems like GMail, Google Calendar, and Google Docs.

In order to manage this codebase, Google engineers developed a compiler for Javascript (https://developers.google.com/closure), which partly provides performance improvements, but also provides static type checking using type declarations in comments, e.g.:

```javascript
/**
 * Whether the array is empty.
 * @param {goog.array.ArrayLike} arr The array to test.
 * @return {boolean} true if empty.
 */
goog.array.isEmpty = function(arr) { ... }
```

Static checking is Google’s friend; make it yours too.
Equality

Now we turn to the other topic of this lecture: defining equality of values in a datatype.

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, actually, 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: \(1+2, \sqrt{9}\), 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 \(f: \mathbb{R} \rightarrow \mathbb{A}\) maps concrete instances of a datatype 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 \(E \subseteq T \times T\) that is:

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

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

These 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 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 \(\in\), these expressions are 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
- ... and so on

In terms of abstract datatypes, “observation” means calling methods on the objects. So two objects are equal if and only if they cannot be distinguished by calling methods on the objects.

Example

Here’s a simple example of an immutable ADT.

```java
public class Duration {
    private final int mins;
    private final int secs;
}
Now which of the following values should be considered equal?

```java
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()` operator compares object contents -- in other words, object equality, in the sense that we’ve been talking about in this lecture.

Here are the equality operators in several languages:

<table>
<thead>
<tr>
<th>Language</th>
<th>Referential equality</th>
<th>Object equality</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java</td>
<td><code>==</code></td>
<td><code>equals()</code></td>
</tr>
<tr>
<td>Objective C</td>
<td><code>==</code></td>
<td><code>isEqual:</code></td>
</tr>
<tr>
<td>C#</td>
<td><code>==</code></td>
<td><code>Equals()</code></td>
</tr>
<tr>
<td>Python</td>
<td><code>is</code></td>
<td><code>==</code></td>
</tr>
<tr>
<td>Javascript</td>
<td><code>==</code></td>
<td><code>n/a</code></td>
</tr>
</tbody>
</table>

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 datatype, it’s our responsibility to decide what object equality means for values of the datatype, and implement the `equals()` operation appropriately.
The equals() method is defined by Object, and its default implementation looks like this:

```java
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 datatypes, 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:

```java
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:

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

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

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

Recall from earlier in the lecture that the compiler selects between overloaded methods using the compile-time type of the parameters. So we get one method implementation when passing an Object reference, and a different method when passing a Duration reference, even when both point to the same object. Equality has become inconsistent.

It's easy to make a mistake in the method signature, and overload when you meant to override. 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:

```java
@Override // asserts that superclass has this method; compile-time error if not
public boolean equals (Object _that) {
    if (!(_that instanceof Duration)) return false;
    Duration that = (Duration) _that;
    return this.getLength() == that.getLength();
}
```
\texttt{return this.getLength()} == \texttt{that.getLength();}

This fixes the immediate problem:

\begin{verbatim}
Duration d1 = new Duration(1, 2);
Object o2 = new Duration(1,2);
d1.equals(o2) \rightarrow \text{true}
o2.equals(d1) \rightarrow ?? // is it symmetric?
\end{verbatim}

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 \texttt{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
era\textit{tically and unpredictably. You don’t want to program with a datatype 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 Durations, because different computers
may have slightly unsynchronized clocks:

\begin{verbatim}
private static final int CLOCK_SKEW = 5; // seconds

@Override
public boolean equals (Object _that) {
    if (!(\_that instanceof Duration)) return false;
    Duration that = (Duration) \_that;
    return Math.abs(this.getLength() - that.getLength()) <= CLOCK_SKEW;
}
\end{verbatim}

Which property of the equivalence relation is violated? Reflexivity, symmetry, or transitivity?

Hashing

To understand the part of the contract relating to the \texttt{hashCode} method, you’ll need to have some
idea of how hash tables work. To keep this lecture self-contained, we describe hash tables below. 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 \texttt{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 'hash buckets'), implemented in Java as objects from class 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 matches the given 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():

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

```java
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:

```java
@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 web (Stack Overflow and Wikipedia) also has advice
about writing hash code functions. Note, however, that as long as you satisfy the requirement that
\texttt{equals()} 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 \texttt{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:

\textbf{Always override \texttt{hashCode} when you override \texttt{equals}.}

Many years ago in 6.170, a student spent hours tracking down a bug in a project that amounted to
nothing more than mispelling \texttt{hashCode} as hashcode. This created a new method that didn’t override
the \texttt{hashCode} method of Object at all, and strange things happened. Use @Override!

\textbf{Equality of Mutable Objects}

We’ve been focusing on equality of immutable objects so far in this lecture. 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:

\begin{itemize}
  \item when they cannot be distinguished by observation \textit{that doesn’t change the state of the objects}, i.e.,
        by calling only observer, producer, and creator methods. This is often strictly called
        \textbf{observational equality}, since it tests whether the two objects “look” the same, in the
        current state of the program.
  \item when they cannot be distinguished by \textit{any} observation, even state changes. This
        interpretation allows calling any methods on the two objects, including mutators. This is
        often called \textbf{behavioral equality}, since it tests whether the two objects will “behave” the
        same, in this and all future states.
\end{itemize}

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 datatypes, 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:

\begin{verbatim}
List<String> list = Arrays.asList(new String[] { “a” });

Set<List<String>> set = new HashSet<List<String>>();
set.add(list);
\end{verbatim}

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

\texttt{set.contains(list)} → \texttt{true}

But now we mutate the list:

\texttt{list.add(“goodbye”)};

And it no longer appears in the set!

\texttt{set.contains(list)} → \texttt{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!

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

If the set’s iterator and its contains() method disagree about whether an element is in the set, then the set clearly is broken.

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.Mutable objects should just inherit equals() and hashCode() from Object. For clients that need a notion of observational equality (whether two 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()**

To summarize, 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:
```java
Integer x = new Integer(3);
Integer y = new Integer(3);
x.equals(y) // returns true
```

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

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

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

```java
(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:

```java
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") // but what do we get out of the map?
```

**Summary**

- the visitor pattern provides complete static checking for datatype traversals
- overloaded methods use compile-time method selection, while overridden methods use runtime type dispatch
- equality should be an equivalence relation (reflexive, symmetric, transitive)
- equality and hash code must be consistent with each other
- the abstraction function is the basis for equality in immutable datatypes
- reference equality is the basis for equality in mutable datatypes; the only way to ensure consistency over time and avoid breaking rep invariants of hash tables