Java theory and practice: Hashing it out

Defining hashCode() and equals() effectively and correctly

Brian Goetz May 27, 2003

Every Java object has a `hashCode()` and an `equals()` method. Many classes override the default implementations of these methods to provide a higher degree of semantic comparability between object instances. In this installment of *Java theory and practice*, Java developer Brian Goetz shows you the rules and guidelines you should follow when creating Java classes in order to define `hashCode()` and `equals()` effectively and appropriately.

View more content in this series

While the Java language does not provide direct support for associative arrays -- arrays that can take any object as an index -- the presence of the `hashCode()` method in the root `Object` class clearly anticipates the ubiquitous use of `HashMap` (and its predecessor, `Hashtable`). Under ideal conditions, hash-based containers offer both efficient insertion and efficient retrieval; supporting hashing directly in the object model facilitates the development and use of hash-based containers.

Defining equality

The `Object` class has two methods for making inferences about an object's identity: `equals()` and `hashCode()`. In general, if you override one of these methods, you must override both, as there are important relationships between them that must be maintained. In particular, if two objects are equal according to the `equals()` method, they must have the same `hashCode()` value (although the reverse is not generally true).

The semantics of `equals()` for a given class are left to the implementer; defining what `equals()` means for a given class is part of the design work for that class. The default implementation, provided by `Object`, is simply reference equality:

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

Under this default implementation, two references are equal *only* if they refer to the exact same object. Similarly, the default implementation of `hashCode()` provided by `Object` is derived by mapping the memory address of the object to an integer value. Because on some architectures

the address space is larger than the range of values for `int`, it is possible that two distinct objects could have the same `hashCode()`. If you override `hashCode()`, you can still use the `System.identityHashCode()` method to access this default value.

Overriding equals() -- a simple example

An identity-based implementation for `equals()` and `hashCode()` is a sensible default, but for some classes, it is desirable to relax the definition of equality somewhat. For example, the `Integer` class defines `equals()` similarly to this:

```
public boolean equals(Object obj) {
  return (obj instanceof Integer
          && intValue() == ((Integer) obj).intValue());
}
```

Under this definition, two `Integer` objects are equal *only* if they contain the same integer value. This, along with `Integer` being immutable, makes it practical to use an `Integer` as a key in a `HashMap`. This value-based approach to equality is used by all the primitive wrapper classes in the Java class library, such as `Integer`, `Float`, `Character`, and `Boolean`, as well as `String` (two `String` objects are equal if they contain the same sequence of characters). Because these classes are immutable and implement `hashCode()` and `equals()` sensibly, they all make good hash keys.

Why override equals() and hashCode()?

What would happen if `Integer` did not override `equals()` and `hashCode()`? Nothing, if we never used an `Integer` as a key in a `HashMap` or other hash-based collection. However, if we were to use such an `Integer` object for a key in a `HashMap`, we would not be able to reliably retrieve the associated value, unless we used the exact same `Integer` instance in the `get()` call as we did in the `put()` call. This would require ensuring that we only use a single instance of the `Integer` object corresponding to a particular integer value throughout our program. Needless to say, this approach would be inconvenient and error prone.

The interface contract for `Object` requires that if two objects are equal according to `equals()`, then they must have the same `hashCode()` value. Why does our root object class need `hashCode()`, when its discriminating ability is entirely subsumed by that of `equals()`? The `hashCode()` method exists purely for efficiency. The Java platform architects anticipated the importance of hash-based collection classes -- such as `Hashtable`, `HashMap`, and `HashSet` -- in typical Java applications, and comparing against many objects with `equals()` can be computationally expensive. Having every Java object support `hashCode()` allows for efficient storage and retrieval using hash-based collections.

Requirements for implementing equals() and hashCode()

There are some restrictions placed on the behavior of `equals()` and `hashCode()`, which are enumerated in the documentation for `Object`. In particular, the `equals()` method must exhibit the following properties:

- Symmetry: For two references, `a` and `b`, `a.equals(b)` if and only if `b.equals(a)`

- Reflexivity: For all non-null references, `a.equals(a)`
- Transitivity: If `a.equals(b)` and `b.equals(c)`, then `a.equals(c)`
- Consistency with `hashCode()`: Two equal objects must have the same `hashCode()` value

The specification for `Object` offers a vague guideline that `equals()` and `hashCode()` be *consistent* -- that their results will be the same for subsequent invocations, provided that "no information used in equals comparison on the object is modified." This sounds sort of like "the result of the calculation shouldn't change, unless it does." This vague statement is generally interpreted to mean that equality and hash value calculations should be a deterministic function of an object's state and nothing else.

What should equality mean?

The requirements for `equals()` and `hashCode()` imposed by the Object class specification are fairly simple to follow. Deciding whether, and how, to override `equals()` requires a little more judgment. In the case of simple immutable value classes, such as `Integer` (and in fact for nearly all immutable classes), the choice is fairly obvious -- equality should be based on the equality of the underlying object state. In the case of `Integer`, the object's only state is the underlying integer value.

For mutable objects, the answer is not always so clear. Should `equals()` and `hashCode()` be based on the object's identity (like the default implementation) or the object's state (like Integer and String)? There's no easy answer -- it depends on the intended use of the class. For containers like `List` and `Map`, one could have made a reasonable argument either way. Most classes in the Java class library, including container classes, err on the side of providing an `equals()` and `hashCode()` implementation based on the object state.

If an object's `hashCode()` value can change based on its state, then we must be careful when using such objects as keys in hash-based collections to ensure that we don't allow their state to change when they are being used as hash keys. All hash-based collections assume that an object's hash value does not change while it is in use as a key in the collection. If a key's hash code were to change while it was in a collection, some unpredictable and confusing consequences could follow. This is usually not a problem in practice -- it is not common practice to use a mutable object like a `List` as a key in a `HashMap`.

An example of a simple mutable class that defines `equals()` and `hashCode()` based on its state is `Point`. Two `Point` objects are equal if they refer to the same `(x, y)` coordinates, and the hash value of a `Point` is derived from the IEEE 754-bit representation of the `x` and `y` coordinate values.

For more complex classes, the behavior of `equals()` and `hashCode()` may even be imposed by the specification of a superclass or interface. For example, the `List` interface requires that a `List` object is equal to another object if and only if the other object is also a `List` and they contain the same elements (defined by `Object.equals()` on the elements) in the same order. The requirements for `hashCode()` are defined with even more specificity -- the `hashCode()` value of a list must conform to the following calculation:

```
hashCode = 1;
Iterator i = list.iterator();
while (i.hasNext()) {
    Object obj = i.next();
    hashCode = 31*hashCode + (obj==null ? 0 : obj.hashCode());
}
```

Not only is the hash value dependent on the contents of the list, but the specific algorithm for combining the hash values of the individual elements is specified as well. (The `String` class specifies a similar algorithm to be used for computing the hash value of a `String`.)

Writing your own equals() and hashCode() methods

Overriding the default `equals()` method is fairly easy, but overriding an already overridden `equals()` method can be extremely tricky to do without violating either the symmetry or transitivity requirement. When overriding `equals()`, you should always include some Javadoc comments on `equals()` to help those who might want to extend your class do so correctly.

As a simple example, consider the following class:

```
class A {
  final B someNonNullField;
  C someOtherField;
  int someNonStateField;
}
```

How would we write the `equals()` method for this class? This way is suitable for many situations:

```
public boolean equals(Object other) {
  // Not strictly necessary, but often a good optimization
  if (this == other)
    return true;
  if (!(other instanceof A))
    return false;
  A otherA = (A) other;
  return
    (someNonNullField.equals(otherA.someNonNullField))
      && ((someOtherField == null)
          ? otherA.someOtherField == null
          : someOtherField.equals(otherA.someOtherField)));
}
```

Now that we've defined `equals()`, we have to define `hashCode()` in a compatible manner. One compatible, but not all that useful, way to define `hashCode()` is like this:

```
public int hashCode() { return 0; }
```

This approach will yield horrible performance for `HashMap`s with a large number of entries, but it does conform to the specification. A more sensible implementation of `hashCode()` for `A` would be like this:

```
public int hashCode() {
  int hash = 1;
  hash = hash * 31 + someNonNullField.hashCode();
  hash = hash * 31
              + (someOtherField == null ? 0 : someOtherField.hashCode());
  return hash;
}
```

Note that both of these implementations delegate a portion of the computation to the `equals()` or `hashCode()` method of the state fields of the class. Depending on your class, you may also want to delegate part of the computation to the `equals()` or `hashCode()` function of the superclass. For primitive fields, there are helper functions in the associated wrapper classes that can help in creating hash values, such as `Float.floatToIntBits`.

Writing an `equals()` method is not without pitfalls. In general, it is impractical to cleanly override `equals()` when extending an instantiable class that itself overrides `equals()`, and writing an `equals()` method that is intended to be overridden (such as in an abstract class) is done differently than writing an `equals()` method for a concrete class. See *Effective Java Programming Language Guide*, Item 7 for some examples and more details about why this is so.

Room for improvement?

Building hashing into the root object class of the Java class library was a very sensible design compromise -- it makes using hash-based containers so much easier and more efficient. However, several criticisms have been made of the approach to and implementation of hashing and equality in the Java class library. The hash-based containers in `java.util` are very convenient and easy to use, but may not be suitable for applications that require very high performance. While most of these will never be changed, it is worthwhile to keep in mind when you're designing applications that rely heavily on the efficiency of hash-based containers. These criticisms include:

- Too small a hash range. Using `int`, instead of `long`, for the return type of `hashCode()` increases the possibility of hash collisions.
- Bad distribution of hash values. The hash values for short strings and small integers are themselves small integers, and are close to the hash values of other "nearby" objects. A more well-behaved hash function would distribute the hash values more evenly across the hash range.
- No defined hashing operations. While some classes, such as `String` and `List`, define a hash algorithm to be used in combining the hash values of its constituent elements into a single hash value, the language specification does not define any approved means of combining the hash values of multiple objects into a new hash value. The trick used by `List`, `String`, or the example class `A` discussed earlier in Writing your own equals() and hashCode() methods are simple, but far from mathematically ideal. Nor does the class library offer convenience implementations of any hashing algorithm that would simplify the creation of more sophisticated `hashCode()` implementations.
- Difficulty writing `equals()` when extending an instantiable class that already overrides `equals()`. The "obvious" ways to define `equals()` when extending an instantiable class that already overrides `equals()` all fail to meet the symmetry or transitivity requirements of the `equals()` method. This means that you must understand the structure and implementation

details of classes you are extending when overriding `equals()`, and may even need to expose private fields in the base class as protected to do so, which violates principles of good object-oriented design.

Summary

By defining `equals()` and `hashCode()` consistently, you can improve the usability of your classes as keys in hash-based collections. There are two approaches to defining equality and hash value: identity-based, which is the default provided by `Object`, and state-based, which requires overriding both `equals()` and `hashCode()`. If an object's hash value can change when its state changes, be sure you don't allow its state to change while it is being used as a hash key.

Related topics

- Series: Java theory and practice
- Effective Java: Programming Language Guide
- Hashtables and key objects
- Hashing Concepts and the Java Programming Language
- Java development on developerWorks