Item 9: Always override hashCode when you override equals

A common source of bugs is the failure to override the hashCode method. **You must override hashCode in every class that overrides equals.** Failure to do so will result in a violation of the general contract for Object.hashCode, which will prevent your class from functioning properly in conjunction with all hash-based collections, including HashMap, HashSet, and Hashtable.

Here is the contract, copied from the Object specification [JavaSE6]:

- Whenever it is invoked on the same object more than once during an execution of an application, the hashCode method must consistently return the same integer, provided no information used in equals comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.
- If two objects are equal according to the equals(Object) method, then calling the hashCode method on each of the two objects must produce the same integer result.
- It is not required that if two objects are unequal according to the equals(Object) method, then calling the hashCode method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hash tables.

The key provision that is violated when you fail to override hashCode is the second one: equal objects must have equal hash codes. Two distinct instances may be logically equal according to a class's equals method, but to Object's hashCode method, they're just two objects with nothing much in common. Therefore Object's hashCode method returns two seemingly random numbers instead of two equal numbers as required by the contract.

For example, consider the following simplistic PhoneNumber class, whose equals method is constructed according to the recipe in Item 8:

```
public final class PhoneNumber {
    private final short areaCode;
    private final short prefix;
    private final short lineNumber;

    public PhoneNumber(int areaCode, int prefix,
                       int lineNumber) {
        rangeCheck(areaCode,    999, "area code");
        rangeCheck(prefix,      999, "prefix");
        rangeCheck(lineNumber, 9999, "line number");
```

```
        this.areaCode  = (short) areaCode;
        this.prefix  = (short) prefix;
        this.lineNumber = (short) lineNumber;
    }

    private static void rangeCheck(int arg, int max,
                                   String name) {
        if (arg < 0 || arg > max)
           throw new IllegalArgumentException(name +": " + arg);
    }

    @Override public boolean equals(Object o) {
        if (o == this)
            return true;
        if (!(o instanceof PhoneNumber))
            return false;
        PhoneNumber pn = (PhoneNumber)o;
        return pn.lineNumber == lineNumber
            && pn.prefix  == prefix
            && pn.areaCode  == areaCode;
    }

    // Broken - no hashCode method!

    ... // Remainder omitted
}
```

Suppose you attempt to use this class with a `HashMap`:

```
Map<PhoneNumber, String> m
    = new HashMap<PhoneNumber, String>();
m.put(new PhoneNumber(707, 867, 5309), "Jenny");
```

At this point, you might expect `m.get(new PhoneNumber(707, 867, 5309))` to return `"Jenny"`, but it returns `null`. Notice that two `PhoneNumber` instances are involved: one is used for insertion into the `HashMap`, and a second, equal, instance is used for (attempted) retrieval. The `PhoneNumber` class's failure to override `hashCode` causes the two equal instances to have unequal hash codes, in violation of the `hashCode` contract. Therefore the `get` method is likely to look for the phone number in a different hash bucket from the one in which it was stored by the `put` method. Even if the two instances happen to hash to the same bucket, the `get` method will almost certainly return `null`, as `HashMap` has an optimization that caches the hash code associated with each entry and doesn't bother checking for object equality if the hash codes don't match.

Fixing this problem is as simple as providing a proper `hashCode` method for the `PhoneNumber` class. So what should a `hashCode` method look like? It's trivial to write one that is legal but not good. This one, for example, is always legal but should never be used:

```
// The worst possible legal hash function - never use!
@Override public int hashCode() { return 42; }
```

It's legal because it ensures that equal objects have the same hash code. It's atrocious because it ensures that *every* object has the same hash code. Therefore, every object hashes to the same bucket, and hash tables degenerate to linked lists. Programs that should run in linear time instead run in quadratic time. For large hash tables, this is the difference between working and not working.

A good hash function tends to produce unequal hash codes for unequal objects. This is exactly what is meant by the third provision of the `hashCode` contract. Ideally, a hash function should distribute any reasonable collection of unequal instances uniformly across all possible hash values. Achieving this ideal can be difficult. Luckily it's not too difficult to achieve a fair approximation. Here is a simple recipe:

1. Store some constant nonzero value, say, 17, in an `int` variable called `result`.
2. For each significant field `f` in your object (each field taken into account by the `equals` method, that is), do the following:
   a. Compute an `int` hash code `c` for the field:
      i. If the field is a `boolean`, compute `(f ? 1 : 0)`.
      ii. If the field is a `byte`, `char`, `short`, or `int`, compute `(int) f`.
      iii. If the field is a `long`, compute `(int) (f ^ (f >>> 32))`.
      iv. If the field is a `float`, compute `Float.floatToIntBits(f)`.
      v. If the field is a `double`, compute `Double.doubleToLongBits(f)`, and then hash the resulting `long` as in step 2.a.iii.
      vi. If the field is an object reference and this class's `equals` method compares the field by recursively invoking `equals`, recursively invoke `hashCode` on the field. If a more complex comparison is required, compute a "canonical representation" for this field and invoke `hashCode` on the canonical representation. If the value of the field is `null`, return `0` (or some other constant, but `0` is traditional).

vii. If the field is an array, treat it as if each element were a separate field. That is, compute a hash code for each significant element by applying these rules recursively, and combine these values per step 2.b. If every element in an array field is significant, you can use one of the `Arrays.hashCode` methods added in release 1.5.

b. Combine the hash code `c` computed in step 2.a into `result` as follows:

```
result = 31 * result + c;
```

3. Return `result`.
4. When you are finished writing the `hashCode` method, ask yourself whether equal instances have equal hash codes. Write unit tests to verify your intuition! If equal instances have unequal hash codes, figure out why and fix the problem.

You may exclude *redundant fields* from the hash code computation. In other words, you may ignore any field whose value can be computed from fields included in the computation. You *must* exclude any fields that are not used in `equals` comparisons, or you risk violating the second provision of the `hashCode` contract.

A nonzero initial value is used in step 1 so the hash value will be affected by initial fields whose hash value, as computed in step 2.a, is zero. If zero were used as the initial value in step 1, the overall hash value would be unaffected by any such initial fields, which could increase collisions. The value 17 is arbitrary.

The multiplication in step 2.b makes the result depend on the order of the fields, yielding a much better hash function if the class has multiple similar fields. For example, if the multiplication were omitted from a `String` hash function, all anagrams would have identical hash codes. The value 31 was chosen because it is an odd prime. If it were even and the multiplication overflowed, information would be lost, as multiplication by 2 is equivalent to shifting. The advantage of using a prime is less clear, but it is traditional. A nice property of 31 is that the multiplication can be replaced by a shift and a subtraction for better performance: `31 * i == (i << 5) - i`. Modern VMs do this sort of optimization automatically.

Let's apply the above recipe to the `PhoneNumber` class. There are three significant fields, all of type `short`:

```
@Override public int hashCode() {
    int result = 17;
    result = 31 * result + areaCode;
    result = 31 * result + prefix;
    result = 31 * result + lineNumber;
    return result;
}
```

Because this method returns the result of a simple deterministic computation whose only inputs are the three significant fields in a `PhoneNumber` instance, it is clear that equal `PhoneNumber` instances have equal hash codes. This method is, in fact, a perfectly good `hashCode` implementation for `PhoneNumber`, on a par with those in the Java platform libraries. It is simple, reasonably fast, and does a reasonable job of dispersing unequal phone numbers into different hash buckets.

If a class is immutable and the cost of computing the hash code is significant, you might consider caching the hash code in the object rather than recalculating it each time it is requested. If you believe that most objects of this type will be used as hash keys, then you should calculate the hash code when the instance is created. Otherwise, you might choose to *lazily initialize* it the first time `hashCode` is invoked (Item 71). It is not clear that our `PhoneNumber` class merits this treatment, but just to show you how it's done:

```
// Lazily initialized, cached hashCode
private volatile int hashCode;  // (See Item 71)

@Override public int hashCode() {
    int result = hashCode;
    if (result == 0) {
        result = 17;
        result = 31 * result + areaCode;
        result = 31 * result + prefix;
        result = 31 * result + lineNumber;
        hashCode = result;
    }
    return result;
}
```

While the recipe in this item yields reasonably good hash functions, it does not yield state-of-the-art hash functions, nor do the Java platform libraries provide such hash functions as of release 1.6. Writing such hash functions is a research topic, best left to mathematicians and theoretical computer scientists. Perhaps a later release of the platform will provide state-of-the-art hash functions for its classes and utility methods to allow average programmers to construct such hash functions. In the meantime, the techniques described in this item should be adequate for most applications.

Do not be tempted to exclude significant parts of an object from the hash code computation to improve performance. While the resulting hash function may run faster, its poor quality may degrade hash tables' performance to the point where they become unusably slow. In particular, the hash function may, in prac-

tice, be confronted with a large collection of instances that differ largely in the regions that you've chosen to ignore. If this happens, the hash function will map all the instances to a very few hash codes, and hash-based collections will display quadratic performance. This is not just a theoretical problem. The `String` hash function implemented in all releases prior to 1.2 examined at most sixteen characters, evenly spaced throughout the string, starting with the first character. For large collections of hierarchical names, such as URLs, this hash function displayed exactly the pathological behavior noted here.

Many classes in the Java platform libraries, such as `String`, `Integer`, and `Date`, include in their specifications the exact value returned by their `hashCode` method as a function of the instance value. This is generally *not* a good idea, as it severely limits your ability to improve the hash function in future releases. If you leave the details of a hash function unspecified and a flaw is found or a better hash function discovered, you can change the hash function in a subsequent release, confident that no clients depend on the exact values returned by the hash function.