Item 11: Always override hashCode when you override equals

You must override hashCode in every class that overrides equals. If you fail to do so, your class will violate the general contract for hashCode, which will prevent it from functioning properly in collections such as HashMap and HashSet. Here is the contract, adapted from the Object specification :

- When the hashCode method is invoked on an object repeatedly during an execution of an application, it must consistently return the same value, provided no information used in equals comparisons is modified. This value need not remain consistent from one execution of an application to another.
- If two objects are equal according to the equals(Object) method, then calling hashCode on the two objects must produce the same integer result.
- If two objects are unequal according to the equals(Object) method, it is *not* required that calling hashCode on each of the objects must produce distinct results. However, the programmer should be aware that producing distinct 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, suppose you attempt to use instances of the PhoneNumber class from Item 10 as keys in a HashMap:

```
Map<PhoneNumber, String> m = new HashMap<>();
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 instead, 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, because 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 writing a proper hashCode method for PhoneNumber. So what should a hashCode method look like? It's trivial to write a bad one. This one, for example, is always legal but should never be used:

```
// The worst possible legal hashCode implementation - 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 instances. This is exactly what is meant by the third part of the hashCode contract. Ideally, a hash function should distribute any reasonable collection of unequal instances uniformly across all int values. Achieving this ideal can be difficult. Luckily it's not too hard to achieve a fair approximation. Here is a simple recipe:

1. Declare an int variable named result, and initialize it to the hash code c for the first significant field in your object, as computed in step 2.a. (Recall from Item 10 that a significant field is a field that affects equals comparisons.)
2. For every remaining significant field f in your object, do the following:
   a. Compute an int hash code c for the field:
      i. If the field is of a primitive type, compute *Type*.hashCode(f), where *Type* is the boxed primitive class corresponding to f's type.
      ii. 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, use 0 (or some other constant, but 0 is traditional).
      iii. If the field is an array, treat it as if each significant element were a separate field. That is, compute a hash code for each significant element by applying these rules recursively, and combine the values per step 2.b. If the array has no significant elements, use a constant, preferably not 0. If all elements are significant, use Arrays.hashCode.

   b. Combine the hash code c computed in step 2.a into result as follows:
   ```
   result = 31 * result + c;
   ```
3. Return result.

When you are finished writing the `hashCode` method, ask yourself whether equal instances have equal hash codes. Write unit tests to verify your intuition (unless you used AutoValue to generate your `equals` and `hashCode` methods, in which case you can safely omit these tests). If equal instances have unequal hash codes, figure out why and fix the problem.

You may exclude *derived 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.

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, because 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 on some architectures: `31 * i == (i << 5) - i`. Modern VMs do this sort of optimization automatically.

Let's apply the previous recipe to the `PhoneNumber` class:

```
// Typical hashCode method
@Override public int hashCode() {
    int result = Short.hashCode(areaCode);
    result = 31 * result + Short.hashCode(prefix);
    result = 31 * result + Short.hashCode(lineNum);
    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 par with those in the Java platform libraries. It is simple, is reasonably fast, and does a reasonable job of dispersing unequal phone numbers into different hash buckets.

While the recipe in this item yields reasonably good hash functions, they are not state-of-the-art. They are comparable in quality to the hash functions found in the Java platform libraries' value types and are adequate for most uses. If you have a bona fide need for hash functions less likely to produce collisions, see Guava's `com.google.common.hash.Hashing` [Guava].

The Objects class has a static method that takes an arbitrary number of objects and returns a hash code for them. This method, named hash, lets you write one-line hashCode methods whose quality is comparable to those written according to the recipe in this item. Unfortunately, they run more slowly because they entail array creation to pass a variable number of arguments, as well as boxing and unboxing if any of the arguments are of primitive type. This style of hash function is recommended for use only in situations where performance is not critical. Here is a hash function for PhoneNumber written using this technique:

```
// One-line hashCode method - mediocre performance
@Override public int hashCode() {
   return Objects.hash(lineNum, prefix, areaCode);
}
```

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* the hash code the first time hashCode is invoked. Some care is required to ensure that the class remains thread-safe in the presence of a lazily initialized field (Item 83). Our PhoneNumber class does not merit this treatment, but just to show you how it's done, here it is. Note that the initial value for the hashCode field (in this case, 0) should not be the hash code of a commonly created instance:

```
// hashCode method with lazily initialized cached hash code
private int hashCode; // Automatically initialized to 0

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

Do not be tempted to exclude significant fields 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 unusable. In particular, the hash function may be confronted with a large collection of instances that differ mainly in regions you've chosen to ignore. If this happens, the hash function will map all these instances to a few hash codes, and programs that should run in linear time will instead run in quadratic time.

This is not just a theoretical problem. Prior to Java 2, the `String` hash function used at most sixteen characters evenly spaced throughout the string, starting with the first character. For large collections of hierarchical names, such as URLs, this function displayed exactly the pathological behavior described earlier.

Don't provide a detailed specification for the value returned by `hashCode`, so clients can't reasonably depend on it; this gives you the flexibility to change it. Many classes in the Java libraries, such as `String` and `Integer`, specify the exact value returned by their `hashCode` method as a function of the instance value. This is *not* a good idea but a mistake that we're forced to live with: It impedes the ability to improve the hash function in future releases. If you leave the details unspecified and a flaw is found in the hash function or a better hash function is discovered, you can change it in a subsequent release.

In summary, you *must* override `hashCode` every time you override `equals`, or your program will not run correctly. Your `hashCode` method must obey the general contract specified in `Object` and must do a reasonable job assigning unequal hash codes to unequal instances. This is easy to achieve, if slightly tedious, using the recipe on page 51. As mentioned in Item 10, the AutoValue framework provides a fine alternative to writing `equals` and `hashCode` methods manually, and IDEs also provide some of this functionality.