Why hashCode Was Not Optional

The same equals, the same two defects, and one missing hashCode method is the entire difference between a Set size of 1 and a duplicate defect filed twice.

Automation Foundations · ashishautomationlabs.store

Now the payoff for Chapter 7, and the clearest demonstration in this book of why `equals` and `hashCode` must travel together.

Here is a `Defect` class with a correct `equals` comparing by `id` -- and a matching `hashCode`. Two defects with the same id are added to a Set:

```
class Defect {
    String id;
    Defect(String id) { this.id = id; }
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        return id.equals(((Defect) o).id);
    }
    @Override
    public int hashCode() {
        return Objects.hash(id);
    }
}
Set<Defect> seen = new HashSet<>();
seen.add(new Defect("DEF-101"));
seen.add(new Defect("DEF-101"));
System.out.println("Set size: " + seen.size());

Set size: 1
```

Correct -- the duplicate was recognized and refused. Now delete the `hashCode` method and change nothing else. Same `equals`, same test:

```
Set size: 2
```

The very same objects, by the very same `equals`, are now stored twice. This is not a subtle difference; it is a duplicate defect filed in your report.

The reason is how a `HashSet` and `HashMap` actually work. They do not compare your object against every item -- that would be slow. They first call `hashCode()` to compute a number, and use it to jump straight to a small group of candidates. Only within that group do they call `equals`. So when `hashCode` is missing, Java uses the default, which is based on object identity. The two identical defects get different numbers, land in different groups, and `equals` is never even called between them.

That is the whole rule, now with a reason attached: **objects that are equal must return the same** `hashCode`. Break it and your objects work fine in a `List` and silently misbehave in every `Set` and `Map`. It is also why `String` is such a safe key -- it has a well-built `equals` and `hashCode` already. This also answers whether one of your own classes can be a Map key: yes, and that is exactly when the `equals`/`hashCode` rule becomes critical. A key type without both is a key you cannot reliably find again -- most of the time a `String` or a number is the simpler, safer key.

Promise Note #3 -- opened in Chapter 5, when lists could not do lookups or refuse duplicates. Paid here, with the Map and the Set. That was the last note outstanding: #1 (the `main` method) was paid in Chapter 8, #2 (arrays) in Chapter 5, and #4 (`equals` and `hashCode`) in Chapter 7. The book owes you nothing.

Every idea in this container has resurfaced from somewhere earlier in this book. The closing bug hunt proves it -- three wrong lines, and not one of them is a new mistake.