equals and hashCode for Your Own Types

Two Defect objects with identical data still count as unequal until you say otherwise -- the promise opened in Chapter 6, and the reason a defect list can catch duplicates.

Automation Foundations · ashishautomationlabs.store

Chapter 6 taught that `==` compares identity and `.equals` compares contents, and it left a promise: for your own classes, you must define what "equal" means. Here's why. Make two `Defect` objects with identical data and compare them, with no `equals` written:

```
class Defect {
    String id;
    String severity;

    Defect(String id, String severity) {
        this.id = id;
        this.severity = severity;
    }
}

Defect a = new Defect("DEF-101", "HIGH");
Defect b = new Defect("DEF-101", "HIGH");
System.out.println("a == b      : " + (a == b));
System.out.println("a.equals(b) : " + a.equals(b));
```

```
a == b      : false
a.equals(b) : false
```

Both `false`. `==` is `false` because they're two different objects -- expected. But `.equals` is `false` too, and that surprises people. The reason: until you say otherwise, your class's built-in `.equals` does the exact same thing as `==` -- it compares identity, not contents. Two defects with the same id and severity are still, to Java, two unrelated objects.

Writing your own equals

You fix this by writing your own `equals`, which spells out what makes two defects equal -- and its required partner, `hashCode`:

```
@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    Defect other = (Defect) o;
    return id.equals(other.id) && severity.equals(other.severity);
}
```

```
@Override
public int hashCode() {
    return Objects.hash(id, severity);
}
```

With that added to the `Defect` class, the same comparison changes its answer:

```
a == b      : false
a.equals(b) : true
```

Read the `equals` method in plain English. `@Override` announces that you're replacing the built-in method. It accepts any object `o`. If it's literally the same object, it's equal. If `o` is `null`, or isn't even a `Defect`, it's not equal. Otherwise, cast `o` to a `Defect` so you can read its fields, and declare them equal when both the id and the severity match -- comparing text with `.equals`, of course. `hashCode` is the partner: any two objects your `equals` calls equal must return the same number here, and `Objects.hash(...)` builds one from the same fields. The rule to memorize is short: **whenever you write `equals`, write `hashCode` too, from the same fields** -- write them together or not at all. If you write `equals` without `hashCode`, your objects behave correctly in a list but break in the `Set` and `Map` containers coming later, which use `hashCode` to find things.

Why it matters: duplicate detection

Recall from Chapter 5 that a list's `contains` searches with `.equals`. So a defect list can now recognize a duplicate by its contents:

```
List<Defect> known = new ArrayList<>();
known.add(new Defect("DEF-101", "HIGH"));
known.add(new Defect("DEF-102", "LOW"));
Defect lookFor = new Defect("DEF-101", "HIGH");
System.out.println("Already known? " + known.contains(lookFor));
```

```
Already known? true
```

A brand-new object, never added to the list, is correctly found as already known -- because you taught `Defect` what equal means. Without your `equals`, that line prints `false`, and your suite files the same defect twice.

> *A `Defect` class overrides `equals` to compare by id and severity, with a matching `hashCode`. A list already contains `new Defect("DEF-9", "LOW")`. What does `list.contains(new Defect("DEF-9", "LOW"))` return, and why?*

>

> *Answer --* `true`*.* `contains` *searches with* `.equals`*, and your* `equals` *compares id and severity, which both match. Without the override it would return* `false`*, because the default* `equals` *compares identity and the two objects are different.*

This is the note opened in Chapter 6, now paid: you teach your own types what equality means by writing `equals` and its partner `hashCode` from the fields that matter.