Walking a Map, and the Order You Can't Rely On

A HashMap gives no order guarantee at all, so an assertion that depends on it is testing an internal detail of Java, not your application -- and it will fail one day for reasons you can't reproduce.

Automation Foundations · ashishautomationlabs.store

You often need every pair, not one lookup. A Map gives you three views: its keys, its values, and its entries (the pairs). The entry loop is the one to learn:

```
Map<String, String> results = new HashMap<>();
results.put("LoginTest", "PASS");
results.put("SearchTest", "FAIL");
results.put("CartTest", "PASS");
for (Map.Entry<String, String> entry : results.entrySet()) {
    System.out.println(entry.getKey() + " -> " + entry.getValue());
}
int failures = 0;
for (String status : results.values()) {
    if (status.equals("FAIL")) {
        failures++;
    }
}
System.out.println("Failures: " + failures);
```

```
SearchTest -> FAIL
LoginTest -> PASS
CartTest -> PASS
Failures: 1
```

Look hard at that output order. The tests were added Login, Search, Cart -- and printed Search, Login, Cart. A `HashMap` does not keep insertion order, and it does not sort. The order it gives is an internal detail that you must never rely on.

This matters for testers more than for most programmers, because an assertion that depends on Map order is a test that passes today and fails next month for no visible reason. A suite builds a report from `Map<String, String> results`, then asserts that the first entry is `LoginTest`. It passes on one machine and in the pipeline for weeks -- stable by accident. A `HashMap` gives no order guarantee at all, so that assertion is not testing your application; it is testing an internal detail of Java. If order matters to the report, say so in the type: use a `LinkedHashMap` for insertion order or a `TreeMap` for sorted keys. Then the guarantee is written into the code rather than hoped for. If order does not matter, assert on the contents instead of the first entry. Never assert on `HashMap` or `HashSet` ordering -- choose a container whose guarantee matches your assertion, or write the

assertion so order is irrelevant.

The counting pattern in the second loop is worth keeping. Combined with `getOrDefault`, it becomes the standard way to tally results:

```
String[] statuses = {"PASS", "FAIL", "PASS", "SKIP", "PASS"};
Map<String, Integer> counts = new HashMap<>();
for (String s : statuses) {
    counts.put(s, counts.getOrDefault(s, 0) + 1);
}

PASS: 3
FAIL: 1
SKIP: 1
```

Read the one line that does the work: take the current count for this status, or zero if this is the first time seeing it, add one, and put it back. That single line is a test run summary, and you will write some version of it in every framework you build.

Looking things up and walking every pair covers a Map's job. The other half of this chapter's promise is refusing duplicates outright -- which is a different container entirely.