The Overwritten Entry and the Case-Sensitive Key

This program prints three lines, and every one is wrong -- and not one of the three mistakes is new. They're Chapter 5's uniqueness, Chapter 6's equality, and Chapter 6's identity, resurfacing in a new container.

Automation Foundations · ashishautomationlabs.store

This program compiles, runs, and prints three lines. Every one is wrong. Read first, then run.

```
public class BugHunt10 {
    public static void main(String[] args) {
        Map<String, Integer> durations = new HashMap<>();
        durations.put("LoginTest", 1200);
        durations.put("CartTest", 1200);
        durations.put("LoginTest", 1200);
        System.out.println("Tests tracked : " + durations.size());
        Map<String, String> owners = new HashMap<>();
        owners.put("LoginTest", "Priya");
        System.out.println("Owner         : " + owners.get("logintest"));
        Integer a = durations.get("LoginTest");
        Integer b = durations.get("CartTest");
        System.out.println("Same duration?: " + (a == b));
    }
}
```

```
Tests tracked : 2
Owner         : null
Same duration?: false
```

Three defects, each from a different idea in this book.

`Tests tracked : 2` after three `put` calls. The third `put` used a key already present, so it replaced the entry instead of adding one. Keys are unique. If the intent was to record a second run of `LoginTest`, the value needed to be a list of durations, or the key needed to include the run number.

`Owner : null` because the lookup used `"logintest"` and the key stored was `"LoginTest"`. Map keys are matched with `equals`, which is case-exact -- the same trap from Chapter 5's list search and Chapter 6's `.equals` rule, now hiding in a key lookup. The value is not missing; you asked for a different key. Normalizing keys to one case when you store and when you look up is the usual fix.

`Same duration?: false` when both durations are `1200`. The values came out of the Map as `Integer` objects, not `int`, and `==` on objects compares identity, exactly as Chapter 6 taught. Java

caches small `Integer` values, so this would have printed `true` for `120` and prints `false` for `1200`. That is worse than being consistently wrong, because it works in your small test and fails on real data. Compare with `.equals`, or unbox into `int` variables first.

Three lines, three wrong answers, no errors. Notice that none of these are new ideas -- they are Chapter 5 uniqueness, Chapter 6 equality, and Chapter 6 identity, all resurfacing in a new container. That is what the last chapter of a core toolkit looks like.

Checkpoint

- What are the two type parameters in `Map<String, Integer>`, in order?
- What does `get` return for a key that is not present, and what are the two

  safer alternatives?

- What happens when you `put` a key that already exists?
- Why must you never assert on the iteration order of a `HashMap`, and what are

  the two alternatives when order matters?

- Explain, using buckets, why an object with `equals` but no `hashCode` can be

  stored twice in a `HashSet`.

- When do you choose a Set over a List?

  Heavily asked, especially for automation roles: the difference between a `List`, a `Set`, and a `Map` (ordered with duplicates; unique with no order; key-value lookups); how a `HashMap` finds a value (uses `hashCode` to pick a bucket, then `equals` within it); what happens if you override `equals` but not `hashCode` (equal objects can land in different buckets, so a Set stores duplicates and a Map lookup fails -- demonstrate with the two outputs from the previous article and you will stand out); whether `HashMap` keeps order (no -- use `LinkedHashMap` for insertion order, `TreeMap` for sorted); and what `get` returns for a missing key (`null`, not an error -- a common source of `NullPointerException`).

  You can look things up by name and you can guarantee uniqueness. You can build a config Map, count results with `getOrDefault`, walk every pair with `entrySet`, and collect distinct values in a Set. You know that a missing key gives `null` rather than an error, that `put` on an existing key overwrites, and that `HashMap` order is not a guarantee you may lean on. Most importantly, you have seen with your own eyes what happens when `hashCode` is missing -- one line of output changing from 1 to 2, and a duplicate defect in your report.

  Look back at what you have. Variables and types. Decisions. Loops. Arrays and lists. Objects you designed yourself, with equality you defined. Methods, including the one the JVM calls.

Failure handling. And now lookups and uniqueness. Every Promise Note this book made is paid, and there is no part of a working core Java program you cannot now read, write, and explain in an interview.

What comes next is not more language. It is putting the language to work: organizing code into a structure that survives a growing suite, and the pattern behind the first line of nearly every test class you'll ever open -- `class LoginTest extends BaseTest`. That's Chapter 11.