Map: Look Up a Value by Its Key

A config maps an environment name to a URL. A run maps each test name to its result. A Map looks up a value by a unique key -- and a missing key quietly returns null instead of raising an error.

Automation Foundations · ashishautomationlabs.store

Lists have carried your test data since Chapter 5, and they have two limits you have been living with. First, a list cannot look something up by name -- to find the URL for the UAT environment in a list, you must walk every item and check each one. Second, a list cannot refuse duplicates -- add the same defect twice and you have it twice. Both are everyday test problems: a config maps an environment name to a URL, a run maps each test name to its result, a defect report needs each unique failure listed once, not forty times.

Java has a container for each. This article covers the first: the **Map**, for lookups by key.

A Map stores pairs. Each pair has a key you look things up by, and a value you get back. Keys are unique; values need not be.

```
public class MP1 {
    public static void main(String[] args) {
        Map<String, String> urls = new HashMap<>();
        urls.put("dev", "https://dev.shopcart.internal");
        urls.put("uat", "https://uat.shopcart.internal");
        urls.put("prod", "https://www.shopcart.com");
        System.out.println("UAT URL : " + urls.get("uat"));
        System.out.println("Has qa? : " + urls.containsKey("qa"));
        System.out.println("Missing : " + urls.get("qa"));
        System.out.println("How many: " + urls.size());
    }
}
```

```
UAT URL : https://uat.shopcart.internal
Has qa? : false
Missing : null
How many: 3
```

Read the declaration: `Map<String, String>` has two types in the angle brackets -- the key type first, then the value type. The key does not have to match the value type: `Map<String, Integer>` maps a name to a number, exactly what you want for durations or counts.

Four operations cover most work. `put(key, value)` stores a pair. `get(key)` returns the value, or `null` when the key is absent. `containsKey(key)` asks whether a key exists. `size()` counts the pairs.

That third line deserves attention, because it is the most common Map bug: a missing key returns `null`, not an error. Your program does not stop; it quietly hands you nothing. If you then use that value, you get the `NullPointerException` from Chapter 6 at some distant line that looks unrelated. Two habits protect you -- check with `containsKey` first, or supply a fallback.

That fallback, and what happens when you try to store two things under the same key, is where this gets interesting.