The Magic String That Matched by Coincidence

This program prints four lines, and three of them expose a defect -- one per idea in this chapter. Organization isn't decoration; every one of these is a bug that structure would have prevented.

Automation Foundations · ashishautomationlabs.store

This program compiles and runs, printing four lines. Three of them expose a defect. Read first, then run.

```
public class BugHunt13 {
    public static void main(String[] args) {
        Config.environment = "uat";
        System.out.println("URL      : " + buildUrl("uat"));
        System.out.println("Timeout  : " + Config.timeout);
        Config.timeout = -5;
        System.out.println("Timeout  : " + Config.timeout);
        System.out.println("Severity : " + slaFor("p1"));
    }
    static String buildUrl(String env) {
        return "https://" + env + ".shopcart.internal";
    }
    static int slaFor(String severity) {
        if (severity.equals("P1")) {
            return 4;
        }
        return 168;
    }
}
class Config {
    public static String environment = "dev";
    public static int timeout = 30;
}
```

```
URL      : https://uat.shopcart.internal
Timeout  : 30
Timeout  : -5
Severity : 168
```

Three defects, one per idea in this chapter.

`Config.environment` was set to `"uat"`, and then `buildUrl("uat")` was called with a literal instead of reading the config. The URL is correct by coincidence. Change `Config.environment` to `"prod"` and this line still builds a UAT URL, because it never consulted the setting at all. A magic string that happens to match a config value is worse than one that does not, because the mismatch stays hidden until the day it matters.

`Config.timeout` is `public static` and not `final`, so any code anywhere can reassign it -- and the third line does, to `-5`. A negative timeout is meaningless, and nothing objected. Constants should be `static final`, and settings that genuinely change should be read through a method that can validate them.

`slaFor("p1")` returned `168`, the default, when P1 should be `4`. The comparison is case-exact and the argument was lowercase, so the `if` was false and the method silently fell to the fallback. A P1 defect was given a week-long SLA. With `Severity` as an enum, `Severity.P1` could not be miswritten, and reading it from text with `valueOf` would have thrown the clear message from the previous article instead of returning a wrong number.

Four printed lines, three real defects, no errors anywhere. Organization is not decoration -- every one of these is a bug that structure would have prevented.

Checkpoint

- What does a package declaration change about a class's real name, and

where must it appear?

- List the four access levels from tightest to loosest, and say what

package-private allows.

- What do `static` and `final` each contribute to a constant?
- Why give a constants or utility class a `private` constructor?
- Give two things an enum can do that a String constant cannot.
- Why is `==` safe for comparing enums when it is not safe for Strings?

This chapter is the shape of every framework repository you will open. Packages separate tests, pages, utils, and models so a newcomer can navigate without a guide. Page Objects keep their locators private and expose public action methods, which is Chapter 7's encapsulation applied at framework scale. `BaseTest` shares the driver with `protected`. Timeouts, file paths, and base URLs live in one constants class rather than scattered through the tests. And enums carry environment, browser, and severity -- with `values()` making cross-environment or cross-browser runs a single loop.

Steady, practical interview questions, all from this chapter: the four access modifiers and what each allows (the four widening rings -- reciting them including package-private marks you out); what `static final` means together (belongs to the class, cannot be reassigned); why use an enum instead of String constants (fixed set, compile-time checking, can carry data and methods, safe with `==`); whether an enum can have a constructor and fields (yes -- it is a class with a fixed set of instances); what package-private is (no modifier, visible within the same package only);

and why a utility class is made `final` with a `private` constructor (it is never meant to be instantiated or extended).

You can turn a folder into a project. You can group classes into packages that tell a newcomer where to look. You can choose an access level with a reason and defend it in an interview, and replace scattered literals with named constants in a class that cannot be instantiated. You can write an enum that carries its own data and methods, and loop over `values()` to run across every environment. You can parse text into a constant with `valueOf`, getting a clear failure rather than a silent wrong answer. More importantly, you can look at an unfamiliar framework and read its structure -- a skill that arrives before you can read all of its code, and it's what makes joining a new team survivable.

Your suite is organized now. The next problem is what flows through it. Test data arrives as text -- a price scraped from a page as `"Rs 1,299.00"`, a status read from a response, a name that arrives with trailing spaces from a spreadsheet. Before you can compare any of it against an expectation, you have to clean it, cut it, and convert it. Text is the raw material of testing, and handling it well is Chapter 14.