Enums: A Fixed Set the Compiler Can Check

A String accepts any text, so a typo compiles and falls silently to a default. An enum has a fixed set the compiler checks, and a bad value is rejected loudly instead of guessed at.

Automation Foundations · ashishautomationlabs.store

Constants fix repetition. They do not fix a deeper problem: a `String` can hold anything. Write `String env = "uat"` and nothing stops a colleague from passing `"UAT"`, `"uat "`, or `"uta"`. Every one of those compiles. You find out at run time, or worse, you do not.

An **enum** is a type with a fixed set of values, and the compiler knows all of them:

```
enum Environment {
    DEV("https://dev.shopcart.internal"),
    UAT("https://uat.shopcart.internal"),
    PROD("https://www.shopcart.com");
    private final String url;
    Environment(String url) {
        this.url = url;
    }
    public String getUrl() {
        return url;
    }
}
```

```
Name    : UAT
URL     : https://uat.shopcart.internal
Is prod?: false
All     :
  DEV -> https://dev.shopcart.internal
  UAT -> https://uat.shopcart.internal
  PROD -> https://www.shopcart.com
```

Look at what that enum is. It has a field, a constructor, and a method -- an enum is a class, with a fixed set of instances created for you. Each constant carries its own data, so the environment and its URL travel together instead of living in a Map somewhere else. Note the useful pieces: `values()` gives every constant, which makes "run against all environments" a one-line loop. `toString` prints the constant's name for free. And comparing enums with `==` is correct and safe -- unlike Strings, where Chapter 6 taught you otherwise -- because each constant is a single shared instance.

A second example shows the rest of the toolkit:

```
enum Severity { P1, P2, P3, P4 }
```

```
P1 -> SLA 4h
P2 -> SLA 24h
P3 -> SLA 72h
P4 -> SLA 168h
Parsed  : P2
Position: 1
```

Enums work in a `switch` from Chapter 3, and inside a `switch` you write the bare constant name. `valueOf("P2")` converts text into the constant, which is how you read a severity from a config file. And `ordinal()` gives its position, counting from zero.

Here is the payoff over a String constant. A typo in an enum name is a compile error -- the code will not build. And when the value arrives as text at run time, `valueOf` refuses anything invalid, loudly:

```
Rejected at run time: No enum constant Severity.p1
```

Lowercase `"p1"` was rejected with a message naming the exact problem. Compare that with the String version of the same mistake, which quietly falls to an `else` branch and reports the wrong SLA.

This is exactly the trap a framework passing the environment around as a plain `String` falls into: a helper does `if (env.equals("prod"))` to decide whether to skip destructive tests. Strings are simple, and everyone knows what `"prod"` means -- until someone passes `"PROD"` from a CI variable. That check silently returns `false`, the destructive tests run, and they run against production. You could lowercase it before comparing -- in every place that compares it, forever, with still no list of valid environments anywhere. With an enum, `Environment.PROD` cannot be misspelled, the compiler checks every use, and `values()` documents the whole set in one line. When a value is one of a known set, make it an enum. The cost is six lines once; the cost of a String is a class of silent bug that only shows up in the environment you least want it in.

So: when is a plain constant enough, and when do you need an enum? Use a constant for a single value with no siblings, such as a file path. Use an enum when the value is one of a fixed, known set -- environments, severities, browsers, statuses. If you find yourself writing `if` chains comparing a String against a handful of known values, you wanted an enum.

Packages, access levels, constants, enums -- four ways of making a suite navigable instead of merely functional. The closing bug hunt shows what happens when a suite skips all four.