One Object Instead of Three Arrays

A test result is a name, a status, and a duration that belong together. Three parallel arrays hold them apart, drifting silently; a class keeps each result whole.

Automation Foundations · ashishautomationlabs.store

You've spent this whole book using objects that other people built. A `String` is an object: it carries text and knows how to compare, search, and change its own case. An `ArrayList` is an object: it carries items and knows how to add, remove, and count them. Each one bundles data together with the operations that belong to it.

Now you build your own. Think about the things you work with all day. A test result isn't one value -- it's a name, a status, and a duration that belong together. A defect is an id, a severity, and a status. So far you've had to carry those pieces in separate variables, or in parallel arrays that must be kept in step by hand.

The problem: pieces that drift apart

Here's how test results look without a class -- three arrays, one per piece:

```
String[] names   = {"LoginTest", "SearchTest", "CartTest"};
String[] statuses = {"PASS",      "FAIL",       "PASS"};
int[]    durations = {240,         512,          180};
```

The name at position 1 is `SearchTest`, its status is at `statuses[1]`, its duration at `durations[1]`. The three arrays are held together only by matching positions, and nothing enforces that. Sort one array and forget the others, insert a row in two of the three, and now `SearchTest` is reported as `PASS` with someone else's duration. The data has drifted, silently, and no error will tell you.

The fix is to stop storing three parallel lists and start storing one list of things, where each thing keeps its own pieces together. That thing is an object, and to make objects you first write a class.

A class is a blueprint; its fields are the data

A class is a blueprint for a kind of object. Here's the simplest possible one -- a `TestResult` that holds three pieces of data:

```
class TestResult {
    String name;
    String status;
    int durationMs;
}
```

Those three lines are its **fields** -- the data every `TestResult` will carry. A field is just a variable that belongs to the object instead of to a single method. Writing the class doesn't create any results yet; it describes what a result is, the way a form describes what a record will contain before anyone fills one in.

To actually make one, you use `new`, then fill in its fields:

```
public class CD1 {
    public static void main(String[] args) {
        TestResult r = new TestResult();
        r.name = "LoginTest";
        r.status = "PASS";
        r.durationMs = 240;
        System.out.println("Test   : " + r.name);
        System.out.println("Status : " + r.status);
        System.out.println("Time   : " + r.durationMs + " ms");
    }
}
```

```
Test   : LoginTest
Status : PASS
Time   : 240 ms
```

`new TestResult()` builds one object in memory and hands back an arrow to it, which you've understood since Chapter 2. The variable `r` holds that arrow, and `r.name` reaches through the arrow to read or set a field. The dot is the same dot you've used on Strings and lists all along -- it means "reach into this object."

Objects are instances; each keeps its own values

One class, many objects. Each object made from the class is called an **instance**, and every instance carries its own copy of the fields. Make two `TestResult` objects and they don't share data -- one can be a `PASS` at 240 ms while the other is a `FAIL` at 512 ms, and neither disturbs the other.

This is exactly what the parallel arrays couldn't guarantee. In an object, the name, status, and duration are one unit. You cannot sort a result's name away from its status, because they're no longer in separate lists -- they're one thing.

> *Using the `TestResult` class above, answer from memory: is `TestResult` a class or an object? In `TestResult r = new TestResult();`, is `r` a class or an object? How many objects does one `new` create?*

>

> *Answer -- `TestResult` is a class, the blueprint. `r` is an object (an instance); it holds an arrow to it. One `new` creates exactly one object -- ten `new` calls make ten separate objects, each with its own fields.*