Arrays Are Fixed Boxes

For four chapters you've trusted String[] and int[] on faith. Here's the whole definition -- fixed-size, numbered, one type -- and why that rigidity is the point.

Automation Foundations · ashishautomationlabs.store

For four chapters you have been trusting me. You stored passwords in a `String[]` and response times in an `int[]`. You looked them up with `[i]` and counted them with `.length`, all on a promise that the full story comes later. This is later.

An array is a **fixed-size, numbered row of boxes, where every box holds the same type of value.** Every word in that definition matters:

- **Fixed-size** -- you choose how many boxes at creation, and that number never

changes.

- **Numbered** -- each box has a position, counted from zero.
- **Same type** -- an `int[]` holds only whole numbers, a `String[]` holds only text.

You cannot mix.

Two ways to create one

When you already know the values, write them directly:

```
String[] browsers = {"chrome", "firefox", "edge"};
```

When you know only the size, not the values yet, create an empty array with `new`:

```
int[] scores = new int[4];
```

That second form raises a fair question: if the boxes are empty, what is inside them? Not nothing -- Java fills them with a default. For number types the default is `0`; for `boolean` it is `false`; for text and other object types it is `null`, the empty arrow from Chapter 3.

```
public class Arr1 {
    public static void main(String[] args) {
        int[] scores = new int[4];
        System.out.println("Fresh array : " + scores[0] + ", " + scores[1] + ", " +
scores[2] + ", " + scores[3]);
        scores[0] = 80;
        scores[1] = 92;
        scores[2] = 75;
        scores[3] = 88;
        System.out.println("After filling: " + scores[0] + ", " + scores[1] + ", " +
scores[2] + ", " + scores[3]);
```

```
        System.out.println("How many?    : " + scores.length);
    }
}
```

```
Fresh array : 0, 0, 0, 0
After filling: 80, 92, 75, 88
How many?    : 4
```

Three things to lock in

You read a box with `scores[i]` and write to it with `scores[i] = value`. Positions run from `0` to `length - 1`, so a four-box array uses positions 0, 1, 2, and 3 -- never 4. And `scores.length` gives the count, which is the safe way to write any loop over the array, exactly as you did in Chapter 4.

> *Predict, given* `int[] a = {10, 20, 30};`*: what is* `a[1]`*? What is* `a.length`*? What happens when you read* `a[3]`*?*

>

> *Answer --* `a[1]` *is* `20`*, because counting starts at zero, so position 1 is the second box.* `a.length` *is* `3`*, the number of boxes. Reading* `a[3]` *fails with* `ArrayIndexOutOfBoundsException`*, because the only positions are 0, 1, and 2.*

The next chapter's problem, already visible

An array's rigidity is exactly right when the count is known and stable -- the twelve months of a year, a fixed lookup table. It becomes wrong the moment your test data doesn't hold still: failures accumulate during a run, rows arrive from a file you didn't write, a search returns three results today and thirty tomorrow. That's where this chapter goes next -- the container built to grow.