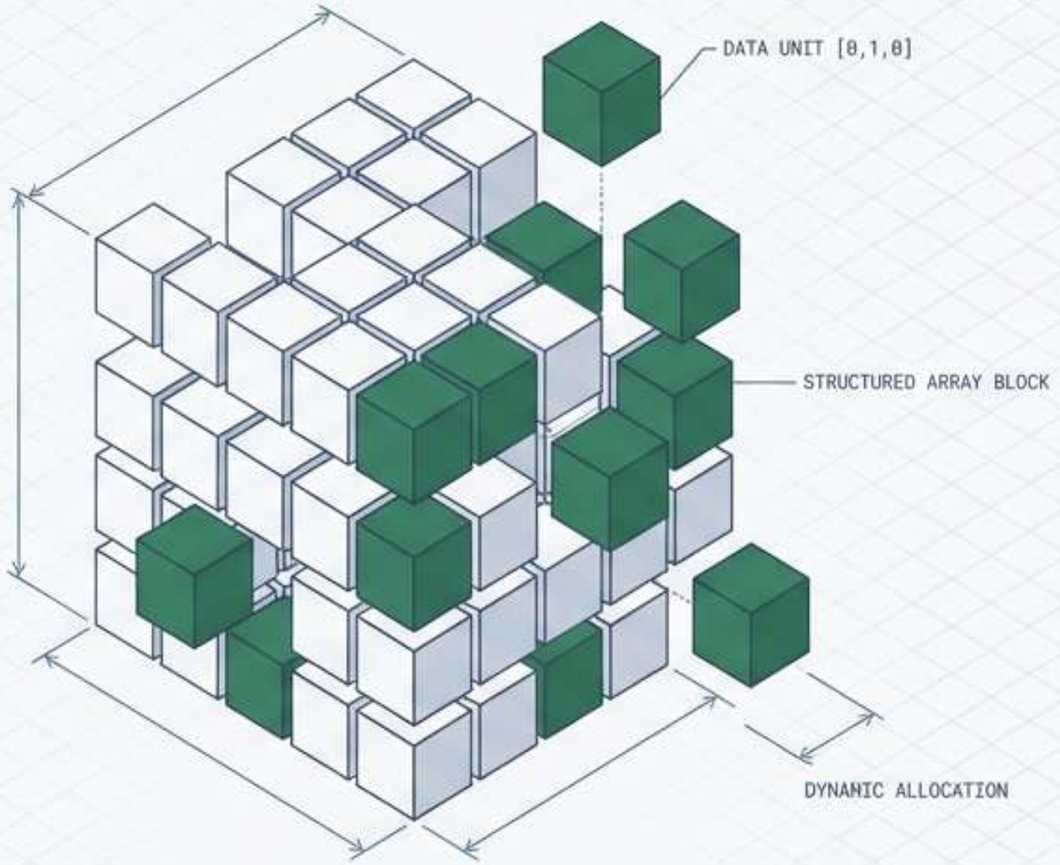
Java Arrays: The Architecture of Efficient Data

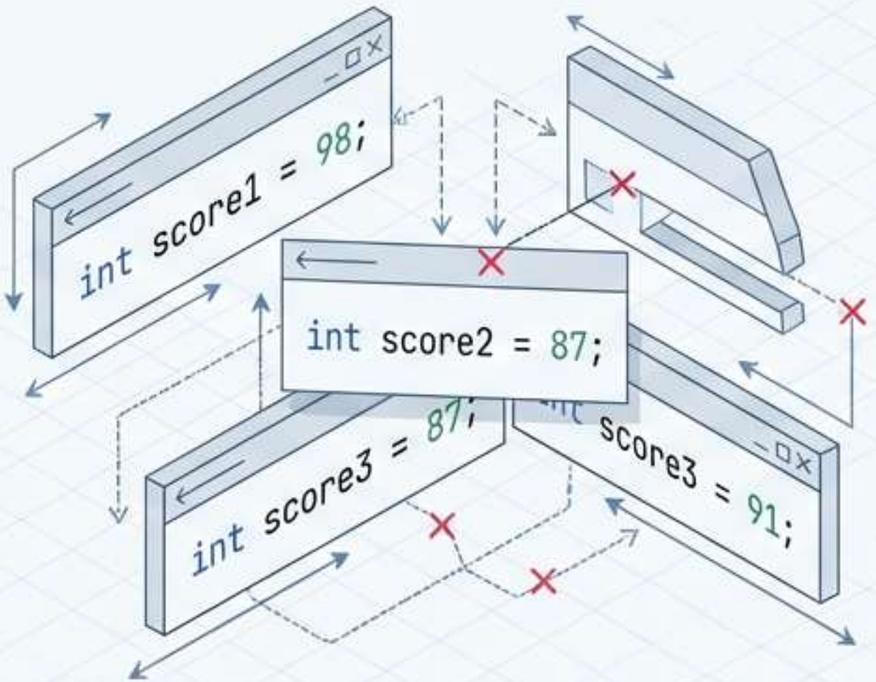
Moving from variable chaos to structured order.



Why do we need Arrays?

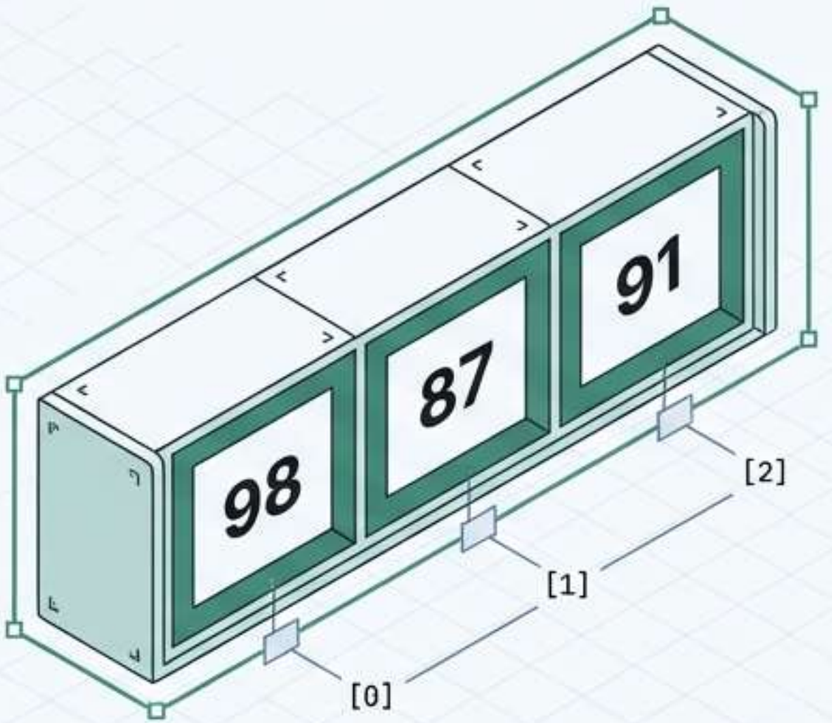
Testleaf
Always Ahead

THE PROBLEM



Multiple variables for related data = Hard to Manage.

THE SOLUTION

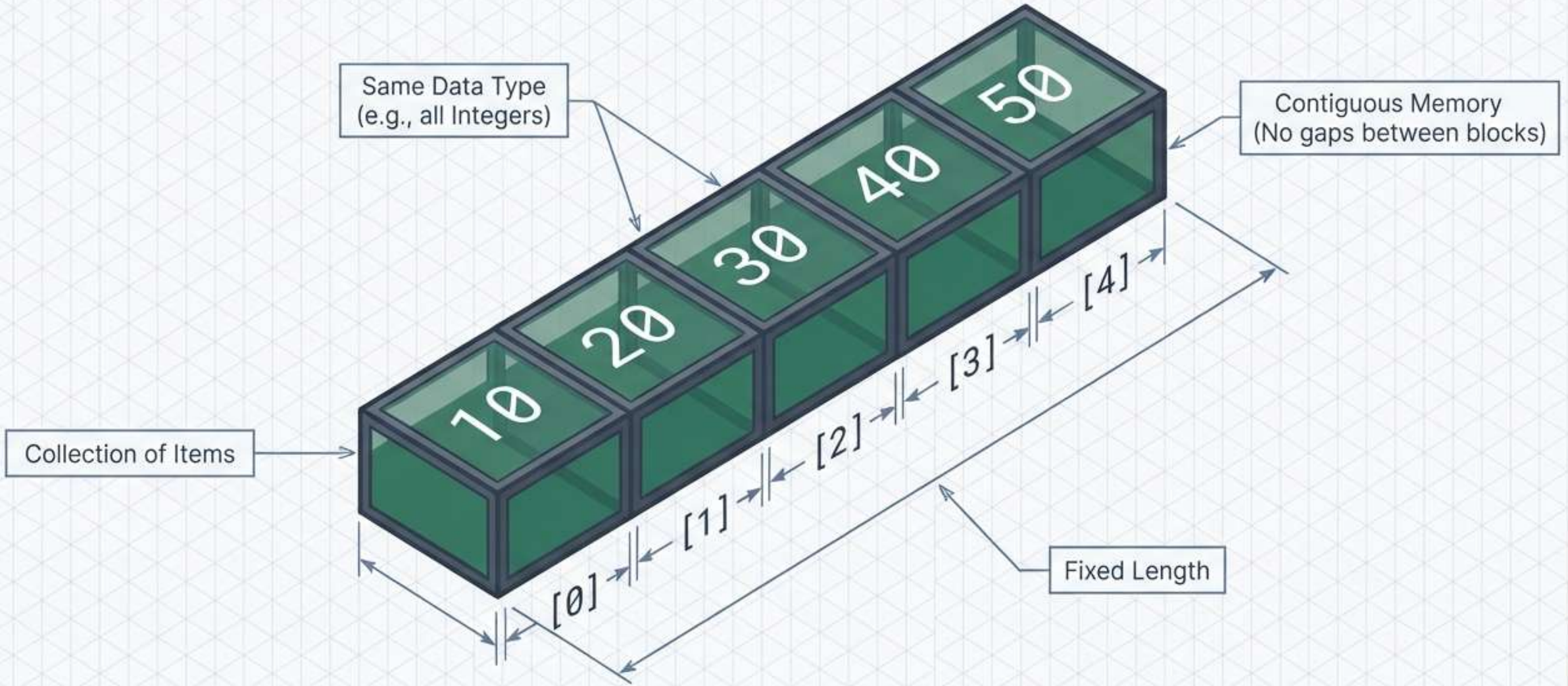


Single variable container = Organized & Efficient.

Arrays bring order to data management, allowing you to handle collections as a single unit.

What is an Array?

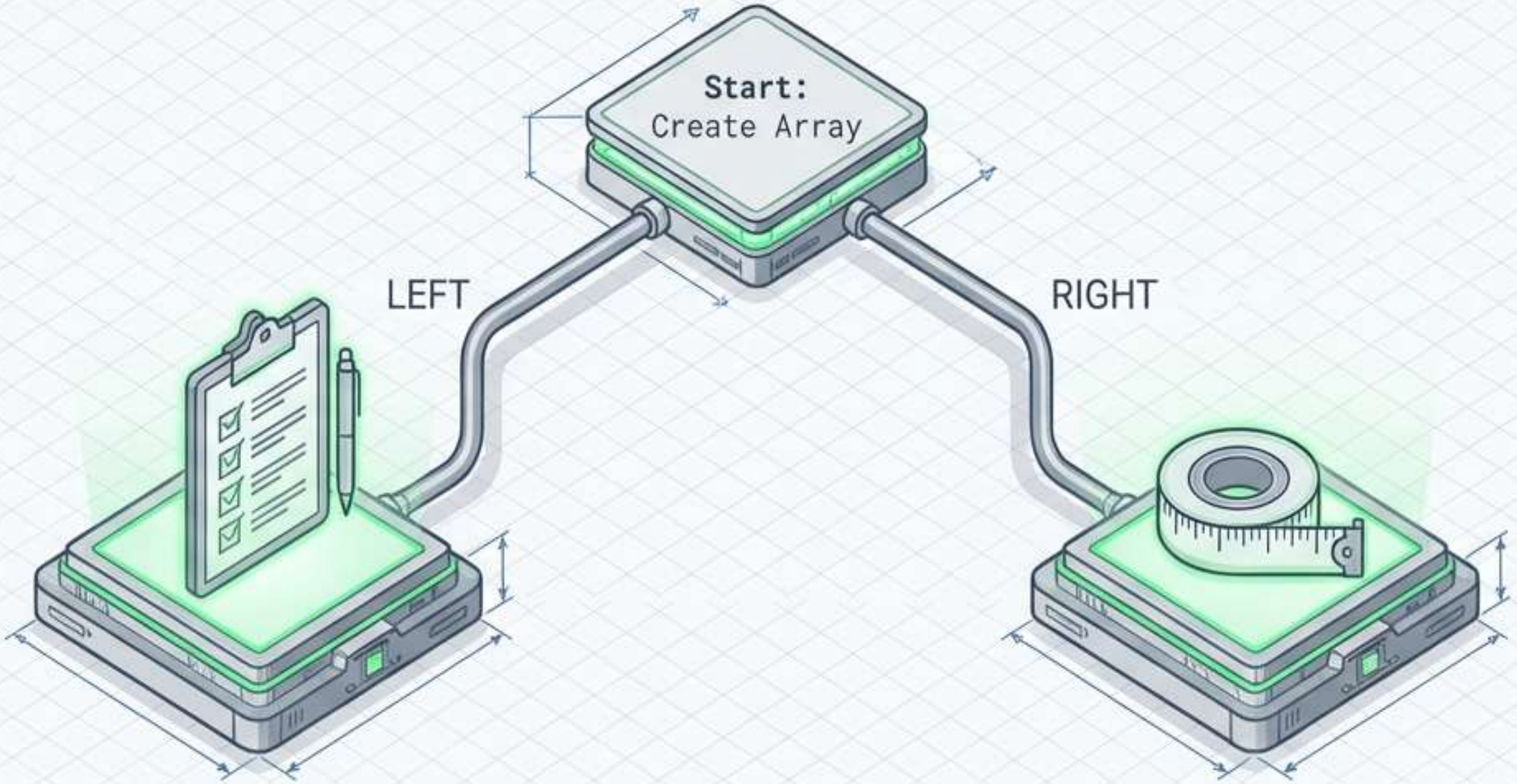
Testleaf
Always Ahead



Two Blueprints for Construction

Testleaf
Always Ahead

How you create an array depends on what you know at the start.



Path A: Array Literal

I already know the data values.

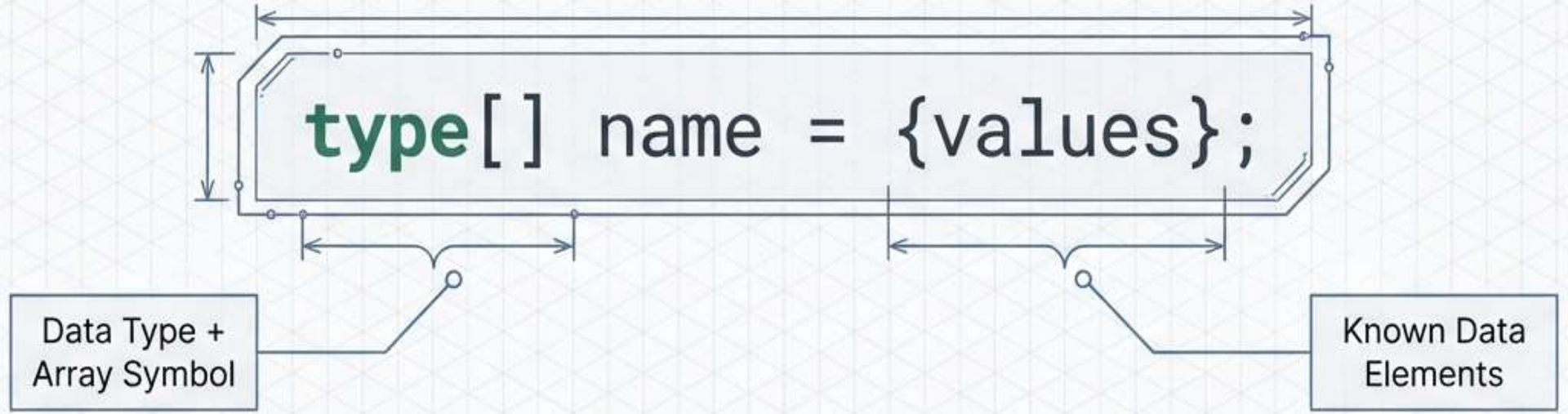
Path B: Instantiation

I know the size, but not the values yet.

Blueprint A: The Array Literal

Testleaf
Always Ahead

Usage: Data-Driven Construction



Valid Syntax Variations

```
int[] scores = {98, 87};
```

Preferred Java Style

Roboto Mono

```
int scores[] = {98, 87};
```

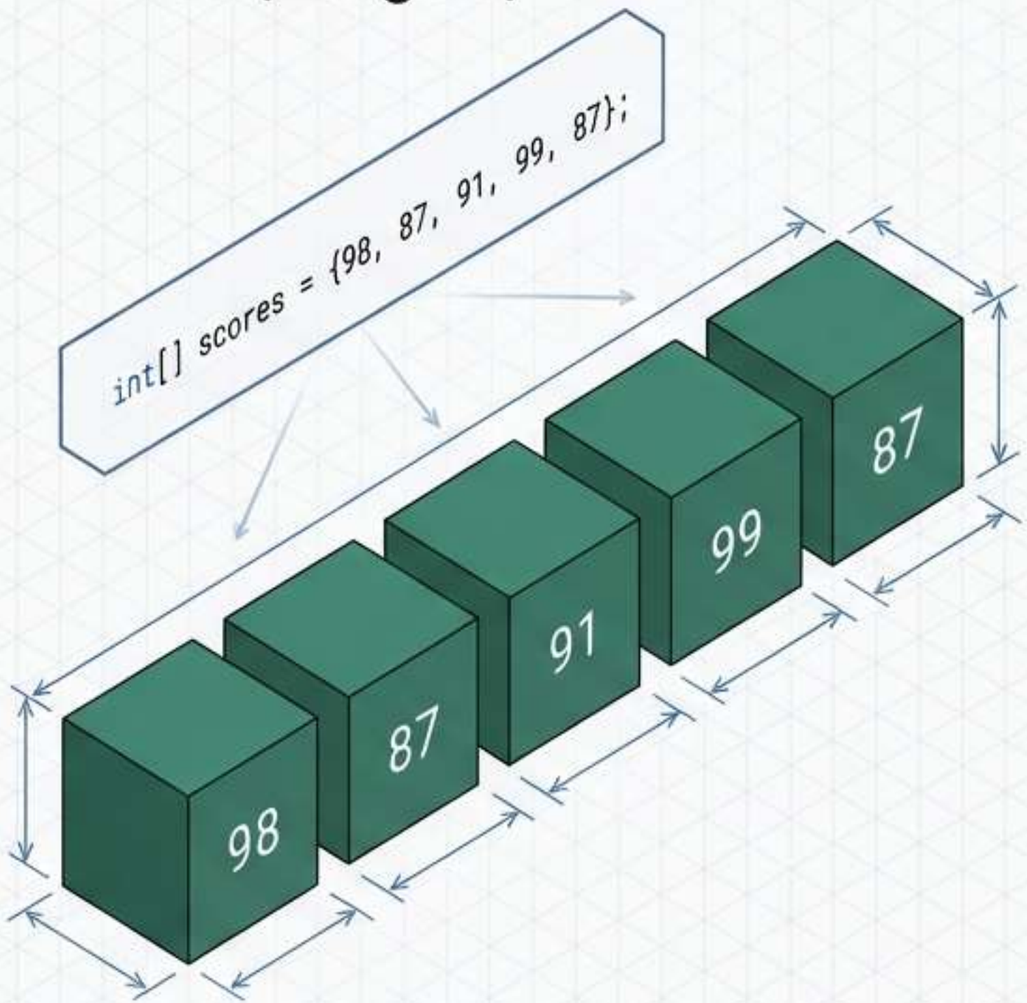
C-Style (Valid but less common)

Roboto Mono

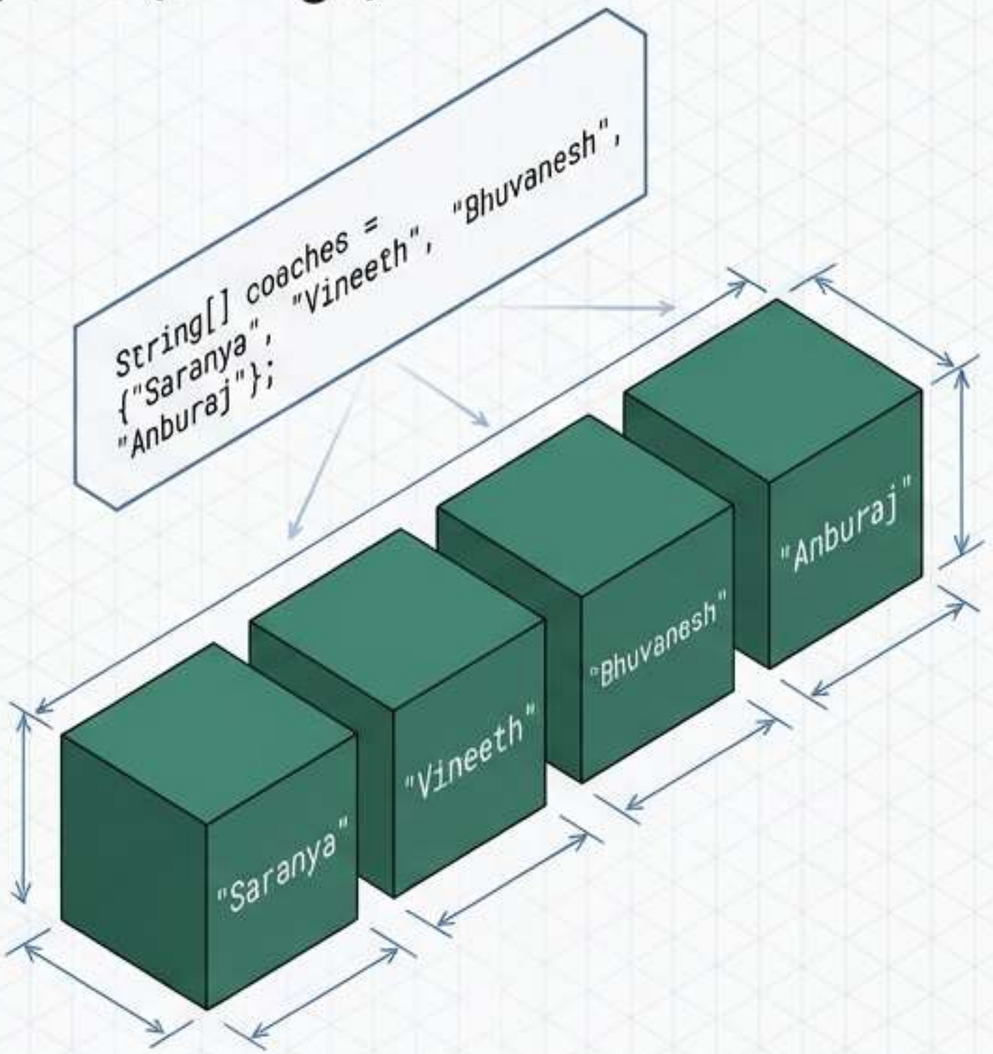
The Literal Blueprint in Action

Testleaf
Always Ahead

Primitives (Integers)



Objects (Strings)



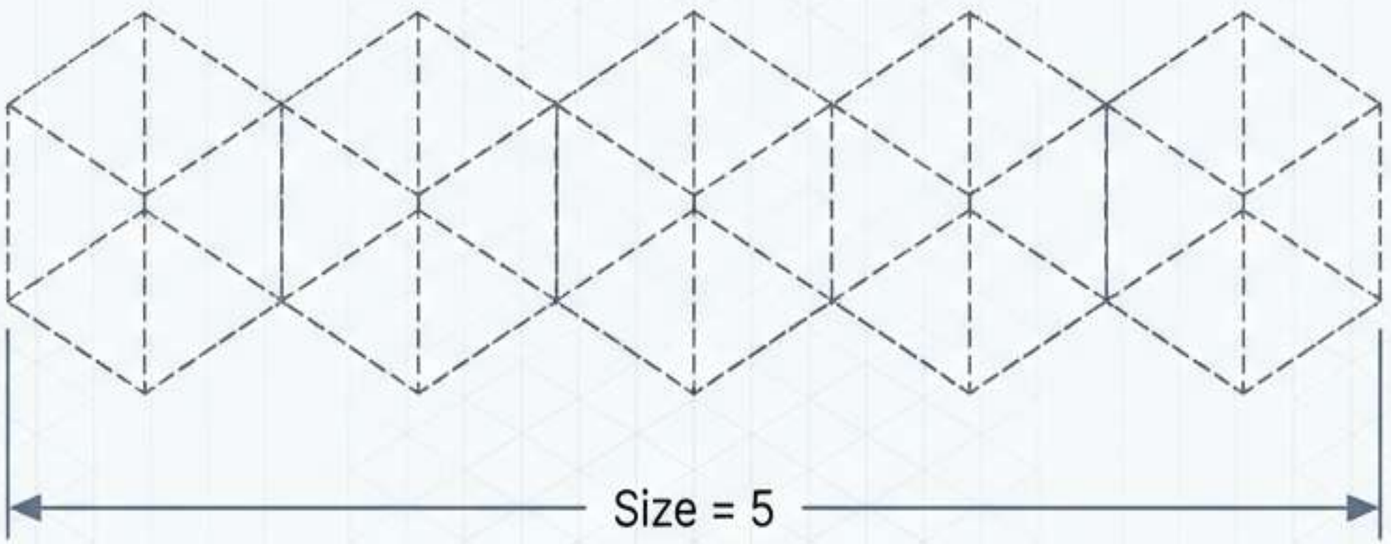
Blueprint B: Instantiation

Testleaf
Always Ahead

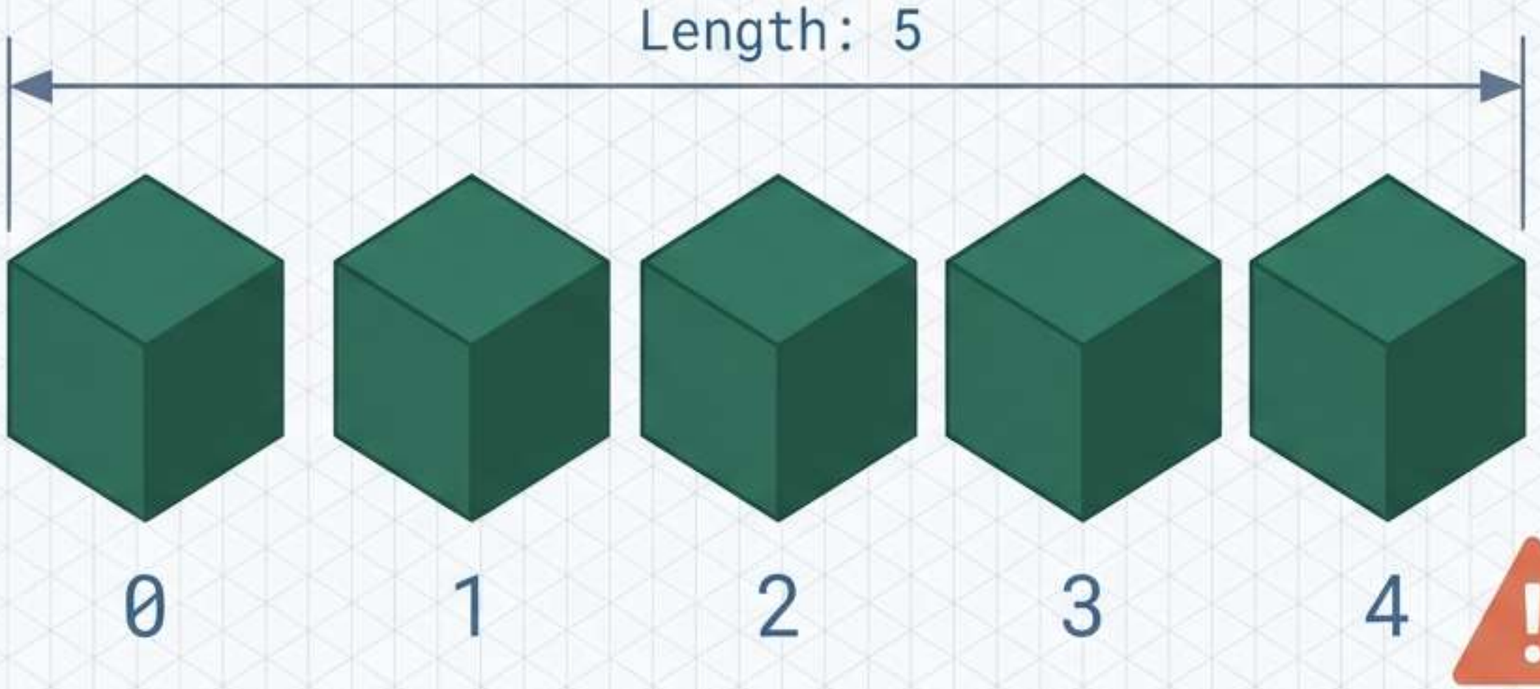
Usage: Capacity-First Construction (Size is known, Data is pending)

'new' = Allocate memory space immediately.

```
int[] scores = new int[5];
```



Mapping the Memory (Indexing)



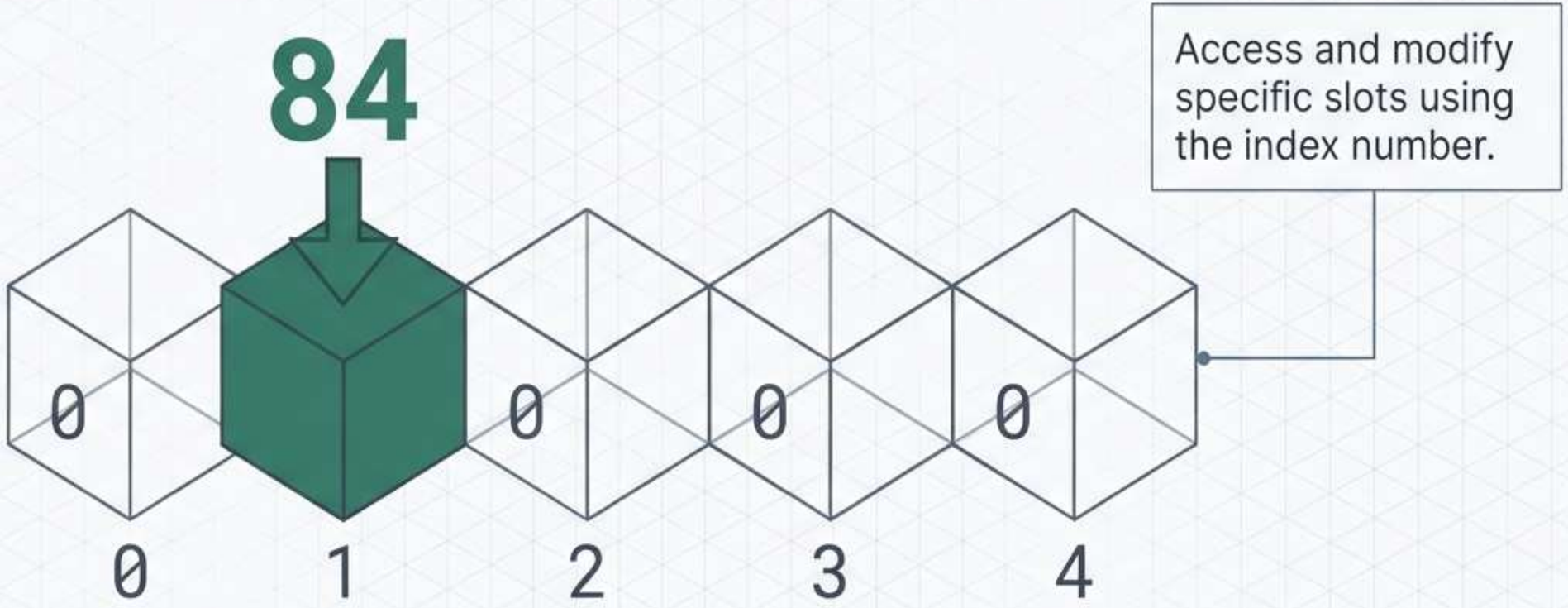
Critical Constraint:
Max Index = Length - 1

Fixed Length: Once created, this structure cannot be resized.

Writing to the Array

Testleaf
Always Ahead

```
scores[1] = 84;
```




```
State: [0, 84, 0, 0, 0]
```