List Operations, and the Choice That Matters

size, get, contains, remove, isEmpty -- the five list operations you'll use most -- plus the one-line rule for choosing an array over a list, or the reverse.

Automation Foundations · ashishautomationlabs.store

A list does far more than grow. Here are the operations you'll use most, in one place:

```
import java.util.ArrayList;
import java.util.List;

public class ListOps {
    public static void main(String[] args) {
        List<String> suite = new ArrayList<>();
        suite.add("LoginTest");
        suite.add("SearchTest");
        suite.add("CheckoutTest");
        System.out.println("Suite size       : " + suite.size());
        System.out.println("First test       : " + suite.get(0));
        System.out.println("Has CheckoutTest?: " + suite.contains("CheckoutTest"));
        suite.remove("SearchTest");
        System.out.println("After removal    : " + suite);
        System.out.println("Is empty?        : " + suite.isEmpty());
    }
}
```

```
Suite size       : 3
First test       : LoginTest
Has CheckoutTest?: true
After removal    : [LoginTest, CheckoutTest]
Is empty?        : false
```

Five operations worth naming. `size()` gives the count -- the list's version of an array's `.length`. `get(i)` reads the item at a position, the way `[i]` does for an array. `contains(...)` searches the whole list and answers `true` or `false`, which an array cannot do for you. `remove(...)` takes an item out and the list closes the gap automatically. And `isEmpty()` is the clear way to ask "is there anything here?" Printing a list shows its contents in square brackets -- `[LoginTest, CheckoutTest]` -- a small courtesy an array does not give you.

> *Predict, given `List<String> env = new ArrayList<>(); env.add("dev");*

env.add("uat"); env.add("prod"); env.remove("uat"); `-- what does printing` env`

> *show, and what does* `env.size()` *return?*

>

> *Answer --* `[dev, prod]` *and* `size = 2`*. Adding three then removing one leaves two, and the list closes the gap so the order stays* `dev, prod`*.*

Two questions worth settling now. A list can hold numbers, not just text -- you write `List<Integer>`, not `List<int>`. Lists hold object types, and `Integer` is the object form of `int`; Java converts between the two for you automatically, so `add(5)` just works. And key-value data -- an environment name mapped to a URL -- is a different container called a `Map`, which arrives once you've met object equality in Chapter 6.

Array or list? A clear rule

You now have two containers, so you need a way to choose. The question is almost always about size.

Use an **array** when the number of items is known and does not change: a fixed matrix, a lookup table you fill once. Arrays are also the only way to hold primitives such as `int` directly, and they carry slightly less overhead.

Use an **ArrayList** when the number of items is unknown or changes over time, or when you need to add, remove, or search. In real automation that is most of the time -- collected failures, filtered rows, a growing list of URLs. When you're unsure, choose the list. The cost is negligible, and the flexibility is exactly what test code tends to need.

A suite once collected the IDs of failing tests into a fixed `String[]` sized to 500. "Five hundred is plenty," the reasoning went -- no imports, no generics needed. Then came a run with 501 failures, and it crashed mid-run. On every ordinary day with four failures, it printed an array with 496 nulls trailing after them, and someone added a counter to track the real count -- which every other piece of code touching that array now also had to know about and honor. A list carries its own size for free. You add four, its size is four, and it can never be wrong. A fixed array plus a manual counter is a list you are rebuilding by hand, one bug at a time.