**Arrays:**

**JAVA:**

1. **Declaration**

int [] a; // preferred

int a2[]; // also valid

1. **Allocation**

a = new int[5]; // default 0s

1. **Declaration + init**

int[] b = {1, 2, 3};

1. **Access**

a[0] = 10;

int x = b[2];

1. **Length**

int n = b.length;

1. **Iterate**

for (int i = 0; i < b.length; i++) System.out.println(b[i]);

for (int v : b) System.out.println(v);

1. **Multidimensional**

int[][] m = new int[2][3]; // 2x3 matrix

int[][] ragged = new int[2][]; // jagged

ragged[0] = new int[3];

ragged[1] = new int[1];

1. **Arrays utils**

import java.util.Arrays;

Arrays.sort(b);

int idx = Arrays.binarySearch(b, 2);

System.out.println(Arrays.toString(b));

1. **Dynamic alternative (resizable)**

import java.util.ArrayList;

ArrayList<Integer> list = new ArrayList<>();

list.add(10);

int v = list.get(0);

int size = list.size();

list.remove(0);

1. **Imports**

import java.util.Arrays;

import java.util.ArrayList;

**C:**

1. **Declaration**

int a[5]; // uninitialized

int b[5] = {1,2,3}; // rest zero-initialized

int c[] = {1,2,3,4}; // size inferred

1. **Access**

a[0] = 10;

int x = b[2];

1. **Length (compile-time when size known)**

#define LEN(arr) (sizeof(arr)/sizeof((arr)[0]))

1. **Iterate**

for (size\_t i = 0; i < LEN(b); i++) printf("%d ", b[i]);

1. **Multidimensional**

int m[2][3] = { {1,2,3},{4,5,6} };

1. **Dynamic allocation**

int n = 5;

int \*p = malloc(n \* sizeof \*p);

p[0] = 42;

1. **Resize**

int \*q = realloc(p, 10 \* sizeof \*p);

1. **Free**

free(q);

Notes: No bounds checking; remember to free. Use memset, memcpy for bulk ops.

**C++:**

1. **Fixed-size & std::array**

int a[5] = {1,2,3,0,0}; // C-style array

#include <array>

std::array<int,5> arr = {1,2,3,4,5};

arr[0] = 10;

size\_t n = arr.size();

for (int v : arr) std::cout << v << " ";

1. **Dynamic / Resizable (vector)**

#include <vector>

std::vector<int> v; // empty

v.push\_back(10);

v.emplace\_back(20);

int x = v[0]; // unchecked

int y = v.at(0); // bounds-checked

size\_t sz = v.size();

v.resize(10); // may fill with 0s for int

v.reserve(100); // capacity hint

v.erase(v.begin()+1);

v.insert(v.begin(), 5);

1. **Multidimensional**
2. // Static matrix
3. int m[2][3] = {{1,2,3},{4,5,6}};
4. // Vector of vectors (resizable)
5. std::vector<std::vector<int>> g(2, std::vector<int>(3, 0));
6. g[0][1] = 7;
7. **Node.js (JavaScript)**
8. **Array basics**
9. // Declaration / init
10. const a = []; // empty
11. const b = [1, 2, 3];
12. // Access / length
13. a[0] = 10;
14. const x = b[2];
15. const n = b.length;
16. // Iterate
17. for (let i = 0; i < b.length; i++) console.log(b[i]);
18. for (const v of b) console.log(v);
19. b.forEach((v, i) => console.log(i, v));
20. // Common methods (return new array unless noted)
21. b.push(4); // mutate: add end
22. b.pop(); // mutate: remove end
23. b.unshift(0); // mutate: add start
24. b.shift(); // mutate: remove start
25. const c = b.slice(1, 3); // non-mutating subarray
26. b.splice(1, 2, 9, 9); // mutate: remove/insert
27. const d = b.map(x => x\*2);
28. const e = b.filter(x => x%2===0);
29. const s = b.reduce((acc, v) => acc+v, 0);
30. b.sort((x,y)=>x-y); // numeric sort (mutates)
31. const has = b.includes(2);
32. const idx = b.indexOf(2);
33. // Multidimensional
34. const m = Array.from({length: 2}, \_ => Array(3).fill(0));
35. m[0][1] = 7;
36. Notes: Arrays are dynamic; can be sparse; methods like map/filter are O(n).
37. **Python**
38. **Lists (dynamic array)**
39. # Declaration / init
40. a = [] # empty
41. b = [1, 2, 3]
42. # Access / length
43. a.append(10)
44. x = b[2]
45. n = len(b)
46. # Iterate
47. for i in range(len(b)):
48. print(b[i])
49. for v in b:
50. print(v)
51. # Common ops
52. b.append(4) # add end
53. b.pop() # remove end -> returns value
54. b.insert(1, 99) # insert at index
55. b.remove(99) # remove first matching value
56. b.extend([7,8]) # concatenate in place
57. c = b[1:3] # slicing (non-mutating)
58. d = b[::-1] # reversed copy
59. b.sort() # in-place
60. e = sorted(b) # new sorted list
61. s = sum(b)
62. exists = 2 in b
63. idx = b.index(2) # raises ValueError if not found
64. # Multidimensional
65. m = [[0]\*3 for \_ in range(2)] # safe (no aliasing)
66. m[0][1] = 7
67. **Array (typed) & NumPy (bonus)**
68. import array as arr
69. a = arr.array('i', [1,2,3]) # typed ints; fewer ops than list
70. # For heavy numeric work prefer NumPy (ndarray).

**Quick Reference (Complexities of common ops)**

| **Language / Structure** | **Index access** | **Append** | **Insert middle** | **Delete middle** | **Search (unsorted)** |
| --- | --- | --- | --- | --- | --- |
| Java array | O(1) | — | — | — | O(n) |
| Java ArrayList | O(1) amort. | O(1) amort. | O(n) | O(n) | O(n) |
| C array | O(1) | — | — | — | O(n) |
| C++ std::vector | O(1) amort. | O(1) amort. | O(n) | O(n) | O(n) |
| JS Array | O(1) amort. | O(1) amort. | O(n) | O(n) | O(n) |
| Python list | O(1) amort. | O(1) amort. | O(n) | O(n) | O(n) |

“—” means fixed size (no append/insert without reallocation).

**Common Pitfalls & Tips**

* **Java**: arr.length (field) vs list.size() (method).
* **C**: Arrays don’t carry length; pass size separately; beware out-of-bounds.
* **C++**: Prefer std::vector/std::array; use at() for bounds checks.
* **JS**: sort() mutates and lexicographically sorts by default—always pass a comparator for numbers.
* **Python**: Use list comprehensions and sum/sorted idioms; avoid [[0]\*m]\*n for matrices (aliasing).