Java DSA Starter Guide

A bridge from Java basics to data structures and algorithms (DSA)

Audience: Students who already know variables, types, operators, conditions, loops, methods/parameters/return values, and Big-O.

Goal: Make you ready to solve DSA problems in Java without getting stuck on language or library details.

Version: 2025-12-25

How to use this guide:

- Skim the 'Bridge checklist' first. If any item feels unfamiliar, pause and practice it before moving on.
- Copy the Java templates into your editor (I/O, frequency map, two pointers, recursion).
- For each topic, do the 'Minimum exercises' to prove you can implement it from memory.

Contents

1. Bridge checklist

If you can do everything in this section, you are ready to start DSA proper.

Minimum required skills

- **Arrays:** create, read, update; loop safely; avoid out-of-bounds; in-place updates.
- **Strings:** immutability, **equals()** vs **==**, basic operations, efficient building with **StringBuilder**.
- **Collections:** comfortable use of **ArrayList**, **HashMap**, **HashSet**, **ArrayDeque**.
- **Recursion:** base case + shrinking step; trace calls; understand call stack growth.
- **Constraints mindset:** translate input size to a feasible complexity (for example, n = 10^5 implies O(n) or O(n log n), not O(n^2)).

Readiness test (do these without hints)

- Reverse an int array in-place.
- Check if a string is a palindrome (two pointers).
- Count character frequency using a HashMap.
- Validate parentheses using ArrayDeque as a stack.
- Explain why repeated string concatenation in a loop is slow, and fix it with StringBuilder.

If you fail 2 or more items: do a 2-5 session bridge (Arrays -> Strings -> Hashing -> Stack/Queue -> Recursion).

2. Java tools for DSA

DSA problems often fail due to input speed, overflow, or incorrect comparisons. This section gives reliable defaults.

2.1 Fast input/output templates

Use BufferedInputStream/BufferedReader for speed. Scanner is simple but often too slow for large inputs.

```
// Fast input (int/long) using BufferedInputStream
import java.io.*;
import java.util.*;

class FastScanner {
    private final InputStream in = System.in;
    private final byte[] buffer = new byte[1 << 16];
    private int ptr = 0, len = 0;

    private int readByte() throws IOException {
        if (ptr >= len) {
            len = in.read(buffer);
            ptr = 0;
            if (len <= 0) return -1;
        }
        return buffer[ptr++];
    }

    long nextLong() throws IOException {
        int c;
        do { c = readByte(); } while (c <= ' ' && c != -1);
        long sign = 1;
        if (c == '-') { sign = -1; c = readByte(); }
        long val = 0;
        while (c > ' ') {
            val = val * 10 + (c - '0');
            c = readByte();
        }
        return val * sign;
    }

    int nextInt() throws IOException { return (int) nextLong(); }

    String next() throws IOException {
        int c;
        do { c = readByte(); } while (c <= ' ' && c != -1);
        StringBuilder sb = new StringBuilder();
        while (c > ' ') {
            sb.append((char) c);
            c = readByte();
        }
        return sb.toString();
    }
}
```

Output tip: build output with StringBuilder and print once.

```
StringBuilder out = new StringBuilder();
out.append(answer).append('\n');
System.out.print(out.toString());
```

2.2 Overflow rules (int vs long)

- Use **long** when values can exceed about 2.1 billion (2^31 - 1).
- If you multiply two ints, the multiplication happens as int unless you cast: **(long)a * b**.
- When using modulo, keep intermediate values in long to avoid overflow before the modulo.

```
long prod = (long) a * b;   // safe
long modProd = ((long) a % MOD) * (b % MOD) % MOD;
```

2.3 Comparisons and equality

- For objects (String, Integer, custom classes), use **equals()** for value comparison.
- Use **==** only for primitives or when you explicitly want reference identity.
- For sorting custom objects, use Comparator.

```
if (s1.equals(s2)) { /* same content */ }

Arrays.sort(arr); // primitives
Arrays.sort(items, (a, b) -> Integer.compare(a.key, b.key));
```

2.4 Debugging: trace small cases first

- Write one tiny example by hand. Step through each line and track pointers/indexes.
- Print key variables (left/right pointers, current sum, map size). Remove prints after solving.
- When stuck, reduce the input to the smallest failing test.

3. Arrays and indexing patterns

Arrays are the foundation of most DSA. Most bugs here are index mistakes.

3.1 Core facts

- Array length is fixed. Index range is 0 to n-1.
- Out-of-bounds throws ArrayIndexOutOfBoundsException.
- In-place means you modify the same array without extra arrays (O(1) extra space).

3.2 Standard loop templates

```
// forward scan
for (int i = 0; i < n; i++) { }

// reverse scan
for (int i = n - 1; i >= 0; i--) { }

// two pointers on array
int l = 0, r = n - 1;
while (l < r) {
    // use arr[l], arr[r]
    l++; r--;
}
```

3.3 Patterns you will use constantly

- **Two pointers:** shrink a search space from both ends (reverse, pair sum in sorted array, partition).
- **Sliding window:** maintain a window [l..r] while expanding r and shrinking l (subarrays, substrings).
- **Prefix sums:** precompute cumulative sums to answer range-sum fast and to solve many subarray problems.
- **Difference array:** range updates in O(1) each, then prefix to finalize.

3.4 Prefix sum template

```
// prefix[i] = sum of arr[0..i-1] (note shift)
long[] prefix = new long[n + 1];
for (int i = 0; i < n; i++) prefix[i + 1] = prefix[i] + arr[i];

// range sum [l..r] inclusive:
long sumLR = prefix[r + 1] - prefix[l];
```

3.5 Minimum exercises

- Reverse array in-place.
- Rotate array by k (try both extra-array and in-place with reverse trick).
- Compute prefix sums and answer q range-sum queries.
- Find the maximum subarray sum (Kadane).

4. Strings in Java

Most DSA string solutions depend on understanding immutability and using the right tools.

4.1 Key facts

- String is immutable: operations like +, concat, replace create new strings.
- Use StringBuilder for repeated building/modification.
- Use equals() for content comparison.

4.2 Common operations

```
int n = s.length();
char c = s.charAt(i);
String sub = s.substring(l, r); // l inclusive, r exclusive
char[] chars = s.toCharArray();
```

4.3 StringBuilder template

```
StringBuilder sb = new StringBuilder();
sb.append('a');
sb.append(123);
sb.append("xyz");
String result = sb.toString();
```

4.4 Patterns on strings

- **Two pointers:** palindrome checks, reverse vowels, compare ends.
- **Sliding window:** longest substring with constraints (k distinct, no repeats, at most k replacements).
- **Frequency counting:** anagrams, permutations in substring, character constraints.
- **Parsing:** split is convenient but can be slow; manual parsing is faster.

4.5 Palindrome template

```
boolean isPalindrome(String s) {
    int l = 0, r = s.length() - 1;
    while (l < r) {
        if (s.charAt(l) != s.charAt(r)) return false;
        l++; r--;
    }
    return true;
}
```

4.6 Minimum exercises

- Check palindrome (basic).
- Valid anagram (use int[26] or HashMap).
- Longest substring without repeating characters (sliding window).
- String compression (use StringBuilder).

5. Collections you must know

Most modern DSA solutions are 40% algorithm and 60% using the right data structure correctly.

5.1 ArrayList (dynamic array)

- Use when you need resizable array-like behavior.
- Random access is O(1). Inserting/removing in the middle is O(n).
- Prefer ArrayList over LinkedList for most DSA tasks.

```
ArrayList<Integer> list = new ArrayList<>();
list.add(10);
int x = list.get(0);
```

5.2 HashMap (key -> value)

- Use for frequency counting, last seen index, grouping, mapping values to counts.
- Average O(1) put/get/containsKey (worst-case exists, rarely relevant in normal problem settings).
- Use getOrDefault for clean counting.

```
HashMap<Character, Integer> freq = new HashMap<>();
for (char ch : s.toCharArray()) {
    freq.put(ch, freq.getOrDefault(ch, 0) + 1);
}
```

5.3 HashSet (unique elements)

- Use for membership tests and uniqueness constraints.
- Average O(1) add/contains/remove.

```
HashSet<Integer> seen = new HashSet<>();
if (!seen.add(val)) { /* duplicate */ }
```

5.4 Deque (stack and queue) - use ArrayDeque

- Use ArrayDeque for stack/queue operations. Avoid Stack (legacy) and LinkedList for stack unless required.
- As stack: push/pop/peek using addLast/removeLast/peekLast (or push/pop).
- As queue: addLast/removeFirst/peekFirst.

```
ArrayDeque<Integer> st = new ArrayDeque<>();
st.addLast(1);             // push
int top = st.removeLast(); // pop

ArrayDeque<Integer> q = new ArrayDeque<>();
q.addLast(1);                // enqueue
int front = q.removeFirst(); // dequeue
```

5.5 PriorityQueue (heap)

- Use when you repeatedly need min/max among changing elements.

- Java PriorityQueue is a min-heap by default.
- For max-heap, use Collections.reverseOrder() or custom comparator.

```
PriorityQueue<Integer> minHeap = new PriorityQueue<>();
PriorityQueue<Integer> maxHeap = new PriorityQueue<>(Collections.reverseOrder());
```

5.6 Minimum exercises

- Frequency map (most frequent element).
- Two sum using HashMap.
- Valid parentheses using ArrayDeque.
- K largest elements using PriorityQueue.

6. Recursion essentials

Recursion is a method calling itself with a smaller input until a base case.

6.1 The only three rules

- Base case: stop condition that returns immediately.
- Progress: every call moves closer to the base case.
- Work split: do a small amount of work before/after recursive calls.

6.2 Trace template (mental model)

To trace recursion, write a call tree and track what each call returns. Focus on one level at a time.

6.3 Example: power (fast exponentiation)

```
long pow(long a, long e) {
    if (e == 0) return 1;
    long half = pow(a, e / 2);
    long res = half * half;
    if (e % 2 == 1) res *= a;
    return res;
}
```

6.4 Memoization (top-down DP)

Memoization means: store results of subproblems so you do not recompute them.

```
long fib(int n, long[] memo) {
    if (n <= 1) return n;
    if (memo[n] != -1) return memo[n];
    memo[n] = fib(n - 1, memo) + fib(n - 2, memo);
    return memo[n];
}
```

6.5 Backtracking basics

- Choose: pick an option.
- Explore: recurse.
- Un-choose: undo the choice (restore state).
- Use for permutations, combinations, subsets, N-Queens.

```
void subsets(int idx, int[] arr, ArrayList<Integer> cur, ArrayList<ArrayList<Integer>> ans) {
    if (idx == arr.length) { ans.add(new ArrayList<>(cur)); return; }
    // not take
    subsets(idx + 1, arr, cur, ans);
    // take
    cur.add(arr[idx]);
    subsets(idx + 1, arr, cur, ans);
    cur.remove(cur.size() - 1);
}
```

6.6 Minimum exercises

- Factorial (recursive) and explain stack frames.
- Fibonacci with memoization.
- Generate subsets of an array.
- Generate permutations of a string (optional for beginners).

7. Core problem-solving patterns

These patterns cover a large fraction of interview and contest problems. Learn them as templates.

7.1 Two pointers

- Works best on sorted arrays/strings or when comparing ends.
- Typical problems: pair sum, remove duplicates, partition, palindrome.

```
// Example: remove duplicates from sorted array in-place
int write = 0;
for (int read = 0; read < n; read++) {
    if (read == 0 || arr[read] != arr[read - 1]) {
        arr[write++] = arr[read];
    }
}
// new length is write
```

7.2 Sliding window

- Use when you need best/count over all subarrays/substrings with a constraint.
- Maintain a window [l..r] and a data structure representing the window (count map, sum, max).
- Two common forms: fixed-size window, variable-size window.

```
// Variable-size sliding window skeleton
int l = 0;
for (int r = 0; r < n; r++) {
    // expand: include element at r

    while (/* window invalid */) {
        // shrink: remove element at l
        l++;
    }

    // update answer using current valid window [l..r]
}
```

7.3 Prefix sums + hash map

- Key idea: subarray sum from i+1..j equals prefix[j] - prefix[i].
- To count subarrays with sum = k, store counts of prefix sums seen so far.

```
long sum = 0;
HashMap<Long, Integer> count = new HashMap<>();
count.put(0L, 1);

long ans = 0;
for (int x : arr) {
    sum += x;
    ans += count.getOrDefault(sum - k, 0);
    count.put(sum, count.getOrDefault(sum, 0) + 1);
}
```

7.4 Binary search

- Use when the answer space is sorted or monotonic (true/false changes once).
- Typical: find first/last occurrence, lower/upper bound, search on answer.

```
// Lower bound: first index i where arr[i] >= target (arr sorted)
int lo = 0, hi = n; // hi is exclusive
while (lo < hi) {
    int mid = lo + (hi - lo) / 2;
    if (arr[mid] >= target) hi = mid;
    else lo = mid + 1;
}
int lower = lo;
```

8. Complexity and constraints

Use constraints to pick the correct pattern. A correct but slow solution is still wrong in timed settings.

8.1 Quick feasibility table (rule of thumb)

Rough mental guide for typical competitive programming judges:

- n <= 10^3: O(n^2) may pass; O(n^3) usually fails.
- n <= 10^5: aim for O(n) or O(n log n).
- n <= 10^6: O(n) is preferred; avoid heavy structures inside tight loops unless necessary.
- Large value range (like up to 10^9) usually affects overflow and hashing, not iteration cost.

8.2 How to choose an approach

- Contiguous subarray/substring -> sliding window or prefix sum.
- Sorted input (or sortable) -> two pointers or binary search.
- Counts/frequencies -> HashMap/HashSet or int[] for small alphabets.
- Repeated min/max queries -> heap (PriorityQueue) or deque trick.

9. Common pitfalls in Java DSA

- **String equality:** use equals(), not ==.
- **Substring bounds:** substring(l, r) excludes r.
- **Index errors:** i < n, not i <= n. For prefix sums, be consistent about inclusive/exclusive.
- **Overflow:** cast before multiply; use long for sums.
- **Mutable state in recursion:** copy lists when storing answers; undo changes (backtracking).
- **Using the wrong structure:** LinkedList for stack/queue is slower than ArrayDeque; Stack is legacy.
- **Mod arithmetic:** (a - b) % MOD can be negative; normalize: (x % MOD + MOD) % MOD.
- **Time wasted on I/O:** use fast input for large tests; avoid printing inside loops.
- **Comparator bugs:** do not subtract ints for comparison if values can overflow; use Integer.compare.

```
// Safe comparator
Arrays.sort(arr, (a, b) -> Integer.compare(a, b));
```

10. Practice plan + readiness tests

A small, focused set of problems beats random grinding. This plan assumes 60-90 minutes/day.

10.1 Order of mastery (recommended)

- Arrays basics -> Two pointers -> Sliding window -> Prefix sums -> Hashing patterns
- Sorting + Binary search
- Stacks/Queues (monotonic stack, deque patterns)
- Recursion -> Backtracking -> Trees (DFS/BFS) -> Graphs
- Dynamic Programming (start with 1D DP, then 2D)

10.2 Daily routine

- Warm-up (10 min): retype one template from memory (sliding window / binary search / prefix sums).
- Solve (40-60 min): 1-2 problems focused on a single pattern.
- Review (10-20 min): write 3 bullets - pattern used, invariant, and the biggest pitfall.

10.3 Pattern-based starter problem list

- Arrays: move zeros, rotate array, max subarray sum.
- Two pointers: two sum (sorted), remove duplicates, palindrome.
- Sliding window: longest substring without repeats, at most k distinct, min size subarray sum.
- Prefix sum + map: subarray sum equals k, longest subarray with sum k.
- Stack: valid parentheses, next greater element.
- Binary search: lower bound, first/last occurrence, search on answer (later).

10.4 Final readiness check before starting the full DSA course

- You can implement fast I/O template and solve at least one problem using it.
- You can solve a sliding window problem without copying a solution.
- You can explain why your solution is O(n) or O(n log n) in plain words.
- You can optimize a brute-force approach using a known pattern (hashing/prefix/window).

End note: Once this bridge feels easy, start the full roadmap confidently: arrays/strings -> hashing -> stack/queue -> trees/graphs -> DP.