

# Arrays and Strings

DSA

Presented by  
**Nikhil Nair**

Website  
**[www.guvi.com](http://www.guvi.com)**

# Objectives

- What you will learn today
  - Revisit 1D & 2D Arrays
  - Discuss Common Array Problems
  - String Manipulation Techniques
  - Revisit StringBuilder & StringBuffer
  - Techniques: Sliding Window & Two Pointer

**common goal**





## 1D Arrays Recap

# Arrays in Everyday Data

- Arrays store data items that share one type and purpose
- Think of a shelf with numbered boxes, each box holding one value
- Index starts from zero; position decides access speed, not content meaning
- Simple structure, yet powerful when you understand boundaries

```
// Representing a simple café order list
int[] prices = {120, 90, 100, 140, 80};
System.out.println("Index 0: " + prices[0]);
System.out.println("Index 4: " + prices[4]);
// Access by index gives O(1) constant time lookup
```

# Reading and Printing Safely

- Always confirm array length before looping
- The for-each loop hides index math but can't modify the array in place
- Classic for-loop gives full control for updates and debugging
- Reading beyond valid indices triggers `ArrayIndexOutOfBoundsException`

```
int[] orders = {120, 90, 100, 140, 80};  
System.out.println("Total orders: " + orders.length);  
for (int i = 0; i < orders.length; i++) {  
    System.out.println("Order " + i + ": " + orders[i]);  
}
```

## Activity: Sum and Average

- Problem: Given an array of integers representing daily order totals, compute and print the sum and average
- Ensure no invalid index access occurs.
- Test with small and empty arrays.

```
public class OrderStats {  
    public static void main(String[] args) {  
        // TODO  
    }  
  
    public static void printSumAndAverage(int[] arr) {  
        // TODO  
    }  
}
```



## **2D Arrays: Thinking in Grids**

# Revisiting 2D Arrays

- A 2D array is an array where each element itself is an array
- Two indices identify one element: the first for row, the second for column
- Used for tabular or grid-like data such as seats, maps, or matrices
- Traversing every element needs nested loops, so the time complexity is  $O(n * m)$

```
int[][] seats = {  
    {1, 0, 1, 1},  
    {0, 1, 0, 0},  
    {1, 1, 0, 1}  
};  
System.out.println("Value at row 1, col 2: " + seats[1][2]);  
System.out.println("Rows: " + seats.length  
    + ", Cols in row 0: " + seats[0].length);
```

# Reading and Traversing a 2D Array

- Outer loop goes through each row, inner loop through each column
- Each cell is visited exactly once, total steps =  $O(n * m)$
- Always use the length of each row to avoid index errors
- Printing results as a table helps verify correctness

```
int[][] seats = {  
    {1, 0, 1},  
    {0, 1, 0},  
    {1, 1, 0}  
};  
  
for (int i = 0; i < seats.length; i++) {  
    for (int j = 0; j < seats[i].length; j++) {  
        System.out.print(seats[i][j] + " ");  
    }  
    System.out.println();  
}
```

# Activity: Memory simulation

- Problem Statement

- You are to counts all occupied seats in a library.
- Occupied seat will have a value of 1. Unoccupied will have a value of 0.
- Handle irregular row lengths without errors
- Display total occupied count and discuss its time complexity

- Tasks

- Create a method countOccupied(int[][] seats) that loops through each element and returns the total count of seats marked with 1

```
public class SeatCounter {  
    public static void main(String[] args) {  
        // TODO  
    }  
  
    public static int countOccupied(int[][] seats)  
    {  
        // TODO  
    }  
}
```



## **Common Array Problems**

# Recognizing Reusable Patterns

- What do most array problems have in common?
  - They usually involve a loop that visits each element once or more
- Most tasks follow a simple structure: traverse, check, and update
  - Once you spot the pattern, new problems stop feeling new
- Think in terms of how many full *passes* you make over data
  - One full pass is  $O(n)$ ; nested passes can reach  $O(n^2)$  or higher
- Fewer full passes mean faster and cleaner solutions

# Finding Min, Max, and Frequency

- Problem: Given an array of integers, find the smallest, largest, and frequency of each number
- Can we solve all three in one loop?
- Each element can be checked once, updating multiple results together
  - Each comparison and update takes constant time,  $O(1)$
- Result: Total time complexity =  $O(n)$  for  $n$  elements
  - Space complexity depends on how we store frequency ( $O(k)$  where  $k$  = value range)

```
int[] temps = {31, 28, 35, 29, 35, 32};

int min = temps[0];
int max = temps[0];
int[] freq = new int[51]; // temperature range 0-50

for (int t : temps) {
    if (t < min) min = t;
    if (t > max) max = t;
    freq[t]++;
    // Each check and update = O(1)
}

System.out.println("Min: " + min + ", Max: " + max);
System.out.println("Frequency of 35: " + freq[35]);
```

# Activity: Compare Rows and Columns

- Problem: Given a 2D grid of integers, find the largest row sum and the largest column sum
  - Rows and columns may differ in length; handle safely
- Task:
  - Write a method findMaxSums(int[][] grid) that prints both results.
  - Use separate passes for rows and columns, and discuss complexity.
  - Predict the total time complexity before running the code

```
public class GridSums {  
    public static void main(String[] args) {  
        int[][] grid = {  
            {5, 2, 3},  
            {1, 4, 6},  
            {7, 8, 2}  
        };  
        findMaxSums(grid);  
    }  
  
    public static void findMaxSums(int[][] grid) {  
  
    }  
}
```



# **String Manipulation Techniques**

# Why Strings Behave Differently

- Why does changing a string in Java seem to create a new one?
  - Because strings are immutable; once created, their content cannot change
- Every modification such as *concat*, *replace*, or *substring* creates a new object in memory
- Equality in Java means two different things
  - `==` compares references, `.equals()` compares contents
- Most string modifications are  $O(n)$  because they must copy all characters into new memory

```
String a = "Hello";
String b = a.concat(" World");
System.out.println("a: " + a);
System.out.println("b: " + b);
System.out.println(a == b);    // false
System.out.println(a.equals(b)); // false
```

# Useful API Patterns

- Problem: Given the system message "TXN:9876#Success", create a friendly summary "Transaction 9876 completed"
- Useful String methods:
  - [indexOf\(\)](#) and [substring\(\)](#) to isolate data between markers.
  - [concat\(\)](#) to rebuild new text strings.
  - [charAt\(\)](#) to inspect or validate characters directly ( $O(1)$ ).
  - [intern\(\)](#) to store frequently repeated results in the String pool
- Each operation that scans the entire string is  $O(n)$ ; doing them sequentially keeps overall cost  $O(n)$

```
String msg = "TXN:9876#Success";

int start = msg.indexOf(':') + 1;
int end = msg.indexOf('#');
String id = msg.substring(start, end);

String result = "Transaction ".concat(id).concat("
completed");
System.out.println(result.intern());
```

# Activity: Decode an Encoded Tag

- Problem: You receive a compact tag like "A3C2Z5"
  - Each letter is followed by a digit telling how many times to repeat it.
  - Expected output: "AAACCZZZZZ".
- Task
  - Write a method decodeTag(String input) to expand the tag.
  - Use charAt() to read letters and digits one by one.
  - Use substring() or concat() to build the new string progressively.
  - Call .intern() on the final string to store it in the String pool.

```
public class TagDecoder {
    public static void main(String[] args) {
        decodeTag("A3C2Z5");
        decodeTag("B1D4");
        decodeTag("AB12"); // invalid example
    }

    public static void decodeTag(String input) {
        // TODO
    }
}
```



# **StringBuilder and StringBuffer**

# From Slow Concatenation to StringBuilder

- Each concat creates a new string and copies the old content again
  - This repeated copying makes total work grow as  $O(n^2)$  when done in loops
- StringBuilder stores text in a resizable buffer and edits it directly
  - Appending text adds new characters without creating extra objects
- Why does this difference get bigger as the loop count increases?
  - Because concat keeps copying the entire string each time, while StringBuilder just grows one piece at a time
- Common use cases
  - Building long log messages, CSV exports inside loops, etc

```

long start1 = System.currentTimeMillis();
String s = "";
for (int i = 0; i < 5000; i++) {
    s += "X";
}
System.out.println("Concat time: " +
    (System.currentTimeMillis() - start1) + "ms");

long start2 = System.currentTimeMillis();
StringBuilder sb = new StringBuilder();
for (int i = 0; i < 5000; i++) {
    sb.append("X");
}
System.out.println("Builder time: " +
    (System.currentTimeMillis() - start2) + "ms");
  
```

# Thread Safety in Action: Builder vs Buffer

- StringBuilder and StringBuffer both allow in-place edits
  - The difference is that StringBuffer adds synchronization for thread safety
- StringBuilder is faster in single-threaded code but unsafe if shared across threads. Parallel writes can overlap and cause data loss.
- When should we use StringBuffer instead of StringBuilder?
  - When multiple threads write to the same shared text, such as loggers or shared caches
- Common use cases
  - StringBuilder: dynamic query builders, file generation, data formatting.
  - StringBuffer: centralized logging, shared status strings, message aggregation

```
public class BuilderVsBuffer {
    static StringBuilder sb = new StringBuilder();
    static StringBuffer sbuf = new StringBuffer();

    public static void main(String[] args) throws InterruptedException {
        Runnable task = () -> {
            for (int i = 0; i < 1000; i++) {
                sb.append("A");
                sbuf.append("B");
            }
        };

        Thread t1 = new Thread(task);
        Thread t2 = new Thread(task);
        t1.start();
        t2.start();
        t1.join();
        t2.join();

        System.out.println("Builder length: " + sb.length());
        System.out.println("Buffer length: " + sbuf.length());
    }
}
```



# **Sliding Window Technique**

# Motivation

- Many array and string problems deal with consecutive elements (subarrays or substrings).
- A brute-force approach recomputes results for every group, even when most data overlaps.
- This is inefficient; each new step repeats part of the previous work.
- Sliding Window avoids that repetition by reusing earlier calculations.

```
// Problem: Print the sum of every 3 consecutive numbers
int[] nums = {1, 3, 2, 6, 4};
int k = 3;

// Brute-force approach: recomputes each window fully (O(n*k))
for (int i = 0; i <= nums.length - k; i++) {
    int sum = 0;
    for (int j = i; j < i + k; j++) {
        sum += nums[j]; // recomputes overlapping work
    }
    System.out.println("Sum: " + sum);
}

// Output: 6 11 12 (on new lines)
```

# What is a Sliding Window

- A window is a small, movable section of an array or string.
- Instead of recomputing from scratch, we update results as the window moves.
- Time drops from  $O(n*k)$  to  $O(n)$  as a result of one pass across the data.
- Idea: subtract what left, add what entered, reuse the rest.

```
// Problem: Print the sum of every 3 consecutive numbers efficiently
int[] nums = {1, 3, 2, 6, 4};
int k = 3;

int windowSum = 0;

// Calculate sum of first window
for (int i = 0; i < k; i++) {
    windowSum += nums[i];
}
System.out.println("Sum: " + windowSum);

// Slide the window forward, reusing previous work
for (int i = k; i < nums.length; i++) {
    windowSum += nums[i] - nums[i - k]; // remove leftmost, add new
    System.out.println("Sum: " + windowSum);
}

// Output: 6 11 12 (on new lines)
```

# Fixed-Size Sliding Window

- Fixed-size windows have a constant length (like 3 elements).
- Used for rolling averages, maximum sum subarrays, or moving metrics.
- Example: Find the maximum sum of any 3 consecutive numbers.

// Problem: Find the maximum sum of any 3 consecutive elements

```
int[] nums = {2, 1, 5, 1, 3, 2};  
int k = 3;
```

```
int sum = 0;  
for (int i = 0; i < k; i++) sum += nums[i];  
int maxSum = sum;
```

// Slide the window across the array

```
for (int i = k; i < nums.length; i++) {  
    sum += nums[i] - nums[i - k];  
    maxSum = Math.max(maxSum, sum);  
}
```

```
System.out.println("Max sum: " + maxSum);
```

// Max sum: 9

# Variable-Size Sliding Window

- Window size changes dynamically based on the problem's condition.
- Common in string problems or subarray length optimizations.
- Example: Find the length of the longest substring without repeating characters.

```
// Problem: Find length of longest substring without repeating characters
String s = "abcabcbb";
int left = 0, maxlen = 0;
Set<Character> seen = new HashSet<>();

for (int right = 0; right < s.length(); right++) {
    // Shrink window when duplicate found
    while (seen.contains(s.charAt(right))) {
        seen.remove(s.charAt(left));
        left++;
    }
    // Expand window
    seen.add(s.charAt(right));
    maxlen = Math.max(maxlen, right - left + 1);
}

System.out.println("Longest substring length: " + maxlen);
// Longest substring length: 3
```



## Two Pointer Technique

# Motivation: Why Two Pointers?

- Many problems require comparing or combining elements from both ends of an array or string
- Using one pointer per direction lets us process data efficiently without extra memory
- Instead of nested loops ( $O(n^2)$ ), two pointers often bring it down to  $O(n)$
- Core idea: move pointers toward each other or along the data until a condition is met.

```
// Problem: Check if array has a pair that sums to a target
int[] nums = {1, 3, 4, 7, 10};
int target = 11;

// Brute-force: check every possible pair ( $O(n^2)$ )
for (int i = 0; i < nums.length; i++) {
    for (int j = i + 1; j < nums.length; j++) {
        if (nums[i] + nums[j] == target) {
            System.out.println("Found pair: " + nums[i] + ", " + nums[j]);
        }
    }
}

// Found pair: 1, 10
// Found pair: 4, 7
```

# How Two Pointers Work

- Keep one pointer at the start (left), another at the end (right).
- Adjust pointers based on comparison or condition.
- This reduces unnecessary work by skipping parts that can't match.
- Works best on sorted arrays or ordered data.

```
// Problem: Find all pairs that sum to target using two pointers
int[] nums = {1, 3, 4, 7, 10};
int target = 11;

int left = 0, right = nums.length - 1;

while (left < right) {
    int sum = nums[left] + nums[right];
    if (sum == target) {
        System.out.println("Found pair: " + nums[left] + ", " + nums[right]);
        left++;
        right--;
    } else if (sum < target) {
        left++; // need a larger sum
    } else {
        right--; // need a smaller sum
    }
}

// Found pair: 1, 10
// Found pair: 4, 7
```

## Example: Reverse a String In-Place

- Two pointers can move toward each other to swap values directly.
- Common pattern for reversing arrays or strings.
- Saves space by avoiding creation of a new array.

```
// Problem: Reverse a string in place using two pointers
char[] chars = "hello".toCharArray();

int left = 0, right = chars.length - 1;
while (left < right) {
    char temp = chars[left];
    chars[left] = chars[right];
    chars[right] = temp;

    left++;
    right--;
}

System.out.println(new String(chars));
```



**That's for today!**  
**Any questions?**