

Rohit Gawande's blog

 Dashboard

Chapter 12: Strings – Continuing the DSA Journey



Rohit Gawande

Sep 28, 2024 •  36 min read

Table of contents

Table of Contents

1.What are Strings in Java?

- > How Strings Differ from Arrays
- > String Example:
- > How Strings Differ from Arrays
- > String Example:

Creating Strings in Java

The Most Imp



2. Input/Output

Handling Input and Output with Strings in Java

- › Example: Basic Input and Output with Strings
- › Example: Basic Input and Output with Strings

Understanding next() vs nextLine()

Key Points to Remember

3. String Length

Calculating the Length of a String in Java

- › Example: Calculating the Length of a String
- › Key Differences Between Array Length and String Length
- › Important Notes:
- › Comparison: Array vs. String Length
- › Example: Calculating the Length of a String
- › Key Differences Between Array Length and String Length
- › Important Notes:
- › Comparison: Array vs. String Length

Example to Illustrate:

Conclusion

4. String Concatenation

String Concatenation in Java

- › Example: String Concatenation Using the + Operator
- › Alternative: StringBuilder and StringBuffer
- › Example: String Concatenation Using the + Operator

- › Alternative: Using the concat() Method

Key Points:

5. The charAt() Method

Accessing Characters in a String Using charAt() in Java

- › Example: Using charAt() Method
- › Example: Using charAt() Method

Printing Each Character of a String Using a Loop

- › Example: Iterating Through the String
- › Example: Iterating Through the String

Explanation:

Key Points to Remember:

Explanation of the Code:

Expected Output:

Key Takeaways:

6. Check if a String is a Palindrome

Q1: Check if a String is Palindrome or not

- › What is a Palindrome?
- › Approach to Check if a String is a Palindrome:
- › What is a Palindrome?
- › Approach to Check if a String is a Palindrome:

Explanation:

Time Complexity



Output of the Program:

7. Shortest Path Problem

Q2: Finding the Shortest Path Based on Directions (E, W, N, S)

Understanding the Concept: Cartesian Coordinate System

Visualization of Movements

Example Route: "WNEENESENN"

Finding the Shortest Path (Displacement)

Java Implementation for the Shortest Path Calculation

Explanation of the Code:

Visualization of the Displacement Calculation

Time and Space Complexity Analysis

Conclusion

8. String Function: Compare Strings

String Comparison in Java

Code Explanation

Output Explanation

Why Does This Happen?

- > 1. Understanding == Operator:
- > 2. String Interning:
- > 3. new String("Rohit"):
- > 4. Using .equals():
- > 1. Understanding == Operator:
- > 2. String In
- > 3. new String("Rohit"):

> 4. Using .equals():

Why Should We Use .equals() for String Comparison?

Summary

9. String Function: Substring

Understanding Substrings in Java

What is a Substring?

Example Substrings

Custom Substring Function

Explanation of the Code:

Benefits of Using the Built-in Method

Summary

10. Find the Largest String

Finding the Largest String Lexicographically

Examples:

Using compareTo() Method:

Code Explanation:

Explanation of the Code:

Time Complexity:

11. Why Strings are Immutable

String Storage and StringBuilder in Java

> 1. Understanding String Storage

> String Interning in Java

- › Visualization of Memory Structure
- › Explanation of the Visualization:
- › Why == Doesn't Work for Comparing Strings
- › 2. Why Strings Are Immutable and Inefficient for Modifications
- › Immutability of Strings
- › Example: Appending Characters Using a String
- › Time Complexity of Using Strings for Modification:

12. String Builder

- › 3. Efficient String Modification Using StringBuilder
- › Why StringBuilder is Efficient
- › Example: Appending Characters Using StringBuilder
- › Time Complexity of Using StringBuilder:
- › Summary

13. Convert Letters to Uppercase

- › Understanding Strings and String Manipulation
- › Problem: Converting the First Letter of Each Word to Uppercase
- › Visualization: Memory Representation Using StringBuilder
- › Code: Converting the First Letter of Each Word to Uppercase
- › Sample Output
- › Explanation of the Code
- › Time Complexity Analysis
- › 14. String Compression
- › Rules for
- › Problem Clarification:

- > Why Use StringBuilder?
- > Visualization: Memory Representation Using StringBuilder
- > Code: String Compression Using StringBuilder
- > Sample Output
- > Explanation of the Code:
- > Time Complexity Analysis
- > Points to Note:
- > 15. Practice Questions
- > Conclusion
- > Related Posts
 - > 1. Chapter 1: Variables and Data Types
 - > 2. Chapter 21: LinkedList (Part 1)
- > Other Series
- > Connect With Me

Show less ^

Welcome back to the **DSA Journey!** In this chapter, we'll dive deep into **Strings in Java** and their various methods. Strings are one of the most essential data structures when solving problems in programming. By the end of this chapter, you'll have a solid grasp of how to work with strings, optimize your code, and solve related questions.

Table of Contents

1. What are Strings?
 2. Input/Output
 3. String Length
 4. String Concatenation
 5. The `charAt()` Method
 6. Check if a String is a Palindrome
 7. Shortest Path Problem
 8. String Function: Compare Strings
 9. String Function: Substring
 10. Find the Largest String
 11. Why Strings are Immutable
 12. String Builder
 13. Convert Letters to Uppercase
 14. String Compression
 15. Practice Questions
-

1.What are Strings in Java?

In Java, strings are a fundamental part of working with text and sequences of characters. They allow you to store not only letters but also numbers and symbols. Think of a string as a sequence of **characters** or text. This section covers the basic functionality.

How Strings Differ from Arrays

Before we learned about arrays that store individual characters, such as:

[COPY](#)

```
char[] arr = {'a', 'b', 'c'};
```

Here, each character is stored separately, and you can treat them as individual elements. However, managing and manipulating arrays of characters for larger sequences can become difficult. That's where **Strings** come in.

String Example:

[COPY](#)

```
String str = "abcd";
```

At first glance, this might seem similar to a character array, but strings in Java come with a rich set of methods and properties that make them much more versatile. For example, they help solve many problems by providing operations like comparison, concatenation, and transformation.

Creating Strings in Java

Strings in Java can be created in two main ways.

1. **String Literal:** This is the simplest and most common way to create strings. Here, the string is directly assigned to a variable.

COPY

```
String str = "xyz";
```

2. **Using the `new` Keyword:** Alternatively, you can create strings using the `new` keyword. Even though it looks similar to the literal approach, there's a subtle difference in how memory is handled.

COPY

```
String str1 = new String("Rohit");
```

Both methods may seem the same at first glance, but they differ at the memory level. When you use string literals, they are stored in a special area called the **String Pool**, which optimizes memory usage. On the other hand, strings created with the `new` keyword are stored in the **heap** memory, creating a new object each time. We'll learn more about this in detail later.

The Most Important Point: Strings are Immutable

The **immutability** of strings is one of the most crucial properties to understand. This means that once a string is created, it **cannot be changed**. Every (characters),

Java creates a new string in memory rather than altering the original one.

2. Input/Output

Handling Input and Output with Strings in Java

Java provides the **Scanner** class to handle user input efficiently. When working with strings, there are a couple of methods provided by `Scanner` to capture user input in different ways, depending on what you need: a single word or a full line of text.

Example: Basic Input and Output with Strings

[COPY](#)

```
import java.util.Scanner;

public class Basic {
    public static void main(String[] args) {
        // Creating a Scanner object for input
        Scanner sc = new Scanner(System.in);

        // Input: Capturing a single word using next()
        String str = sc.next();

        // Output: Printing the single word
        System.out.print(str);

        // C
        String str2 = sc.nextLine();
```

```
// Output: Uncommenting the line below will print the entire string
// System.out.println(str5);

// Predefined string for example purposes
String fullName = "Rohit Gawande";

}

}
```

Understanding `next()` vs `nextLine()`

- `next()` Method:
 - The `next()` method reads **one word** of input. It stops reading when it encounters a space or a newline character.

Example:

If the input is: Rohit Gawande

```
Scanner sc = new Scanner(System.in);
String str = sc.next();
System.out.println(str); // Output: Rohit
```

COPY

Here, only the first word `Rohit` is captured and printed, ignoring the rest of the input.

- `nextLine()` Method:
 - The `nextLine()` method reads an **entire line of text** including spaces until it reaches a newline character.

Example:

If the input is: Rohit Gawande

COPY

```
String str5 = sc.nextLine();  
System.out.println(str5); // Output: Rohit Gawande
```

Here, the full line `Rohit Gawande` is captured and printed.

Key Points to Remember

1. `next()` reads **one word** (up to the first space or newline).
2. `nextLine()` reads the **entire line** of input, including spaces.
3. If you use `next()` first, and then use `nextLine()`, you might not get the expected result. This is because `next()` only reads up to the next space, but `nextLine()` reads the entire remainder of the line, which can lead to skipping behavior. To avoid this, always clear the buffer by using `sc.nextLine()` after `next()` if you plan to use both.

3. String Length

Calculating the Length of a String in Java

In Java, calculating the length of a string is simple, but it works a bit differently from `C++` (called `.length`), while **strings** use a method called `.length()`. This is an important

distinction because arrays are not objects, whereas strings are, and objects in Java have methods to perform operations.

Example: Calculating the Length of a String

COPY

```
String fullName = "Rohit Gawande";  
System.out.println(fullName.length()); // Output: 14
```

Key Differences Between Array Length and String Length

- **Array Length:**
 - Arrays use a property called `.length` to determine the size.
 - Example:

COPY

```
int[] arr = {1, 2, 3, 4};  
System.out.println(arr.length); // Output: 4
```

- **String Length:**
 - Strings use the `.length()` method, which is a function that calculates and returns the number of characters in the string, including spaces.
 - Example:

COPY



```
System.out.println(name.length()); // Output: 5
```

Important Notes:

- **Spaces are counted** as characters when using `.length()`. So, if your string contains spaces, they will be included in the total count.
- **String immutability** ensures that even though the string length is calculated dynamically, the string's content does not change.

Comparison: Array vs. String Length

Feature	Arrays	Strings
Property	<code>.length</code>	<code>.length()</code>
Type	Field (property)	Method (function)
Counts	Elements in array	Characters (including spaces) in string

Example to Illustrate:

COPY

```
String fullName = "Rohit Gawande";  
System.out.println("Length of fullName: " + fullName.length()); //
```



In this case, the string "Rohit Gawande" has 14 characters. Each letter is counted, and so is the space between "Rohit" and "Gawande".

Conclusion

The `.length()` method in Java strings calculates the total number of characters, including spaces. It's a function, unlike arrays, which use the `.length` property. Understanding this difference is essential when dealing with string manipulations and comparisons.

4. String Concatenation

String Concatenation in Java

Concatenation is the process of combining two or more strings into one. In Java, this can be done using the `+` operator or the `concat()` method. It's a common and straightforward way to join strings, whether you're combining names, building messages, or constructing dynamic text.

Example: String Concatenation Using the `+` Operator

[COPY](#)

```
String firstName = "Rohit";  
String lastName = "Gawande";  
String fullName = firstName + " " + lastName; // Using the + operator  
System.out.println(fullName); // Output: Rohit Gawande
```


In the example above:

- The `+` operator is used to concatenate `firstName` and `lastName` with a space `" "` in between.
- The resulting string `"Rohit Gawande"` is stored in `fullName` and printed.

Alternative: Using the `concat()` Method

You can also concatenate strings using the `concat()` method, though it's less commonly used compared to the `+` operator.

COPY

```
String firstName = "Rohit";  
String lastName = "Gawande";  
String fullName = firstName.concat(" ").concat(lastName);  
System.out.println(fullName); // Output: Rohit Gawande
```

Both approaches give the same result, but the `+` operator is preferred due to its simplicity and readability.

Key Points:

1. Using the `+` Operator:

- The `+` operator is a simple way to combine strings and is widely used.



- It works by creating a new string in memory and combining the two strings.

COPY

```
String fullName = firstName + " " + lastName;
```

2. Using `concat()` Method:

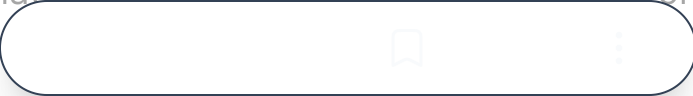
- The `concat()` method is available in the `String` class to join two strings.
- It's functionally equivalent to the `+` operator but requires more typing.

COPY

```
String fullName = firstName.concat(" ").concat(lastName);
```

3. Performance Considerations:

- For smaller concatenations (like combining two or three strings), the `+` operator is perfectly fine.
- However, when performing a large number of string concatenations (like in loops), the `+` operator can create many temporary string objects, leading to performance issues. In such cases, using **`StringBuilder`** (which we'll discuss later) is more efficient.

String concatenation is an essential operation when working with strings in Java.  both provide convenient ways to join strings, with the `+` operator being more

commonly used due to its simplicity. Next, we'll explore how to access individual characters in a string using the `charAt()` method!

5. The `charAt()` Method

Accessing Characters in a String Using `charAt()` in Java

In Java, just like you can access elements in an array using an index, you can access individual characters in a string using the `charAt()` method. This method allows you to retrieve the character at a specific index, where indexing starts from `0`.

Example: Using `charAt()` Method

[COPY](#)

```
String fullName = "Rohit Gawande";  
System.out.println(fullName.charAt(6)); // Output: G
```

In the above example:

- The string "Rohit Gawande" is stored in the variable `fullName`.
- The `charAt(6)` method returns the character at index `6`, which is 'G'.

Indexing starts from `0`, so the character at index `6` is the 7th character in the string, wh

Printing Each Character of a String Using a Loop

You can iterate over each character of a string by using a loop and the `charAt()` method. Here's how you can print each character in the string one by one:

Example: Iterating Through the String

COPY

```
public class StringCharAtExample {  
    public static void main(String[] args) {  
        String fullName = "Rohit Gawande";  
        print(fullName); // Calling the print function to print a  
    }  
  
    // Method to print each character of a string  
    public static void print(String str) {  
        for (int i = 0; i < str.length(); i++) {  
            System.out.print(str.charAt(i)); // Printing each char  
        }  
    }  
}
```



Output:

COPY

Rohit Gawande

In this example:



- We define a method `print(String str)` that takes a string as input.
- The `for` loop iterates over each character of the string from index `0` to `str.length() - 1`.
- The `charAt(i)` method retrieves the character at each index and prints it.

Explanation:

1. `charAt()` Method:

- This method returns the character at a specified index. The index must be a valid number between `0` and `length - 1` where `length` is the total number of characters in the string.

COPY

```
fullName.charAt(6); // Returns 'G'
```

2. Using Loops to Print Each Character:

- A `for` loop iterates over the string, and at each iteration, the `charAt()` method prints the character at the current index.

COPY

```
for (int i = 0; i < fullName.length(); i++) {  
    System.out.print(fullName.charAt(i));  
}
```

Key Points



- **Indexing starts from 0**, so the first character of the string is at index 0.
- The `charAt()` method throws a `StringIndexOutOfBoundsException` if you try to access an index that is out of range (i.e., less than 0 or greater than or equal to the length of the string).

Here's the full Java program along with a detailed explanation of each part and the expected output:

COPY

```
import java.util.Scanner;

public class Basic {

    // Method to print each character of a string
    public static void print(String str) {
        for(int i = 0; i < str.length(); i++) {
            System.out.print(str.charAt(i));
        }
    }

    public static void main(String[] args) {
        // String declaration
        String str = "xyz";
        String str1 = new String("xyz");

        // Strings are IMMUTABLE

        // In
        Scanner sc = new Scanner(System.in);
```

```
System.out.print("Enter a string (next): ");
String strr = sc.next(); // next() will read input until it reaches a space
System.out.println("Output (next): " + strr);

System.out.print("Enter a full line of text (nextLine): ");
String str5 = sc.nextLine(); // nextLine() reads the entire line
System.out.println("Output (nextLine): " + str5);

// String length
String fullname = "Rohit Gawande";
System.out.println("Length of fullname: " + fullname.length());

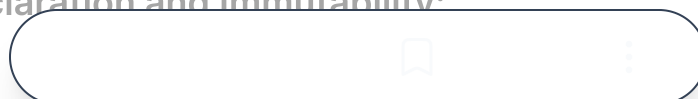
// String concatenation
String firstName = "Rohit";
String lastName = "Gawande";
String fullName = firstName + " " + lastName;
System.out.println("Concatenated Full Name: " + fullName);

// Accessing character at a specific index using charAt
System.out.println("Character at index 6: " + fullName.charAt(6));

// Printing the entire string character by character using
System.out.print("Full Name character by character: ");
print(fullName);
}
}
```

Explanation of the Code:

1. String Declaration and Immutability:



- The strings `str` and `str1` are both initialized with the same value `"xyz"`.
- `str` is a direct string declaration, whereas `str1` uses the `new` keyword. Although they seem the same, they are treated differently in memory (this is part of the immutability and memory handling of strings in Java).

2. Input and Output:

- `sc.next()` reads a single word input, stopping at the first space.
- `sc.nextLine()` reads an entire line of input, including spaces.

3. String Length:

- The `length()` method calculates the total number of characters in the string, including spaces.
- For the string `"Rohit Gawande"`, the length is `13`.

4. String Concatenation:

- The first and last names are concatenated using the `+` operator.
- The result is `"Rohit Gawande"`, with a space between the two strings.

5. Accessing Characters Using `charAt()`:

- The `charAt(6)` method retrieves the character at index `6`, which in this case is `'G'`.

6. Printing the

- The method `print(String str)` loops through each character in the string and prints them one by one using the `charAt()` method.

Expected Output:

[COPY](#)

```
Enter a string (next): xyz
```

```
Output (next): xyz
```

```
Enter a full line of text (nextLine): Rohit Gawande
```

```
Output (nextLine):
```

```
Length of fullname: 13
```

```
Concatenated Full Name: Rohit Gawande
```

```
Character at index 6: G
```

```
Full Name character by character: Rohit Gawande
```

Key Takeaways:

1. **Immutability of Strings:** Strings in Java cannot be changed after they are created. This immutability ensures that strings are thread-safe and efficient in terms of memory management.
2. **Input Handling:** The `Scanner` class provides different methods (`next()`, `nextLine()`) to handle input, and the choice of method affects how much text is read.
3. **String Length:** The `length()` function calculates the number of characters,

4. **Character Access:** The `charAt()` method allows you to retrieve characters at specific indices.
5. **Looping Through Strings:** By using loops and `charAt()`, you can easily process or print strings character by character.

This program showcases fundamental string operations in Java, which form the foundation for more complex string manipulations you'll encounter in data structures and algorithms.

6. Check if a String is a Palindrome

Q1: Check if a String is Palindrome or not

What is a Palindrome?

A palindrome is a word, number, or sequence of characters that reads the same forward and backward. Some common examples of palindromes are:

- "racecar"
- "noon"
- "madam"

For a string to be a palindrome, its characters from the beginning and the end must match as you move inward towards the middle.

Approach to Check if a String is a Palindrome:

The simplest way to check if a string is a palindrome is to divide it by half and compare the first character with the last, the second character with the second last, and so on. If all corresponding characters are equal, the string is a palindrome. Otherwise, it is not.

Here is the full program for checking whether a string is a palindrome:

[COPY](#)

```
// WAP to check if a string is a Palindrome or not
public class Palindrome {

    // Function to check if the string is a palindrome
    public static boolean isPalindrome(String str) {
        for (int i = 0; i < str.length() / 2; i++) {
            int n = str.length(); // length of the string
            // Explanation of (n - 1 - i):
            // - n-1 gives the index of the last character.
            // - n-1-i gives the corresponding character from the r
            if (str.charAt(i) != str.charAt(n - 1 - i)) {
                return false; // If characters don't match, it's r
            }
        }
        return true; // If all characters match, it's a palindrome
    }

    // Main method to test the function
    public static void main(String[] args) {
        String str = "racecar"; // Example 1: palindrome
        System.out.println(isPalindrome(str)); // Output: true

        String str1 = "rohit"; // Example 2: not a palindrome
        System.out.println(isPalindrome(str1)); // Output: false
    }
}
```

Explanation

- `for (int i = 0; i < str.length() / 2; i++) :`
 - The loop runs until the middle of the string (`str.length() / 2`), as we only need to compare the first half with the second half.
- `str.charAt(i) != str.charAt(n - 1 - i) :`
 - The first character at index `i` is compared with the corresponding character from the end, which is at index `n - 1 - i`.
 - `n - 1` gives the index of the last character in the string (since array indices are 0-based).
 - `n - 1 - i` is used to get the character from the end that corresponds to the character at the `i`-th position from the start.

Time Complexity:

- **Time Complexity:** $O(n/2) \approx O(n)$
 - The loop iterates over only half of the string (i.e., $n/2$ times), but in Big-O notation, constant factors are dropped, so the time complexity becomes **$O(n)$** , where n is the length of the string.
 - For each iteration, you are doing constant-time comparisons (`charAt()`), so the overall complexity remains linear.
- **Space Complexity:** $O(1)$
 - We are not using any extra space that grows with the input size. T

Output of the Program:

COPY

```
true // for "racecar" (which is a palindrome)
false // for "rohit" (which is not a palindrome)
```

This approach efficiently checks if a string is a palindrome by comparing corresponding characters from the start and the end until the middle of the string.

7. Shortest Path Problem

Q2: Finding the Shortest Path Based on Directions (E, W, N, S)

The task is to determine the shortest path (displacement) from the starting point to the destination given a set of movement directions ('E' for East, 'W' for West, 'N' for North, and 'S' for South). Let's dive deeper into the concept, how it's visualized, and how we compute the shortest path using Java.

Understanding the Concept: Cartesian Coordinate System

We'll use a **Cartesian coordinate system** to represent movement directions:



1. The Coordinate Axes:

- The starting point (origin) is represented as $((0, 0))$.
- The x-axis runs horizontally (East/West), while the y-axis runs vertically (North/South).

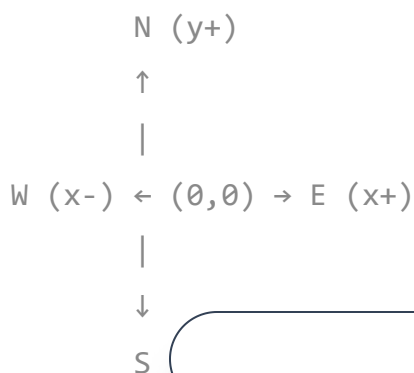
2. Movement and Directions:

- **North (N):** Moves up one unit, so it increases the y-coordinate by 1.
- **South (S):** Moves down one unit, so it decreases the y-coordinate by 1.
- **East (E):** Moves right one unit, so it increases the x-coordinate by 1.
- **West (W):** Moves left one unit, so it decreases the x-coordinate by 1.

Visualization of Movements

To visualize the movement:

- Imagine the following grid where the origin $((0, 0))$ is the starting point:



COPY

- This grid helps us understand how each direction affects the coordinates:
 - Moving **North** increases y .
 - Moving **South** decreases y .
 - Moving **East** increases x .
 - Moving **West** decreases x .

Example Route: "WNEENESENN"

Let's break down the given path: "WNEENESENN". We'll track the coordinates after each movement step-by-step.

1. **Start at** $((0, 0))$ (origin).
2. **W (West)** → Move left: $((x, y) = (-1, 0))$
3. **N (North)** → Move up: $((x, y) = (-1, 1))$
4. **E (East)** → Move right: $((x, y) = (0, 1))$
5. **E (East)** → Move right: $((x, y) = (1, 1))$
6. **N (North)** → Move up: $((x, y) = (1, 2))$
7. **E (East)** → Move right: $((x, y) = (2, 2))$
8. **S (South)** → Move down: $((x, y) = (2, 1))$
9. **E (East)** → Move right: $((x, y) = (3, 1))$
10. **N (North)** → Move up: $((x, y) = (3, 2))$
11. **N (North)** → Move up: $((x, y) = (3, 3))$

The final coordinates after all movements are $((3, 3))$.

Finding the Shortest Path (Displacement)

The shortest path from the origin to the destination ((3, 3)) is the **displacement** between these two points. Displacement is different from the total path taken; it's the straight-line distance (Euclidean distance) from the start to the end point.

1. Mathematical Calculation:

- The displacement formula based on the Pythagorean theorem:

- $$\text{Displacement} = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

- Simplified Formula:** Given the starting point (x_1, y_1) is (0, 0) and the destination (x_2, y_2) is (x, y), the formula simplifies to:

$$\text{Disp} = \sqrt{x^2 + y^2}$$

Java Implementation for the Shortest Path Calculation

COPY

```
public class Q2 {
    public static float getShortestPath(String path) {
        int x = 0, y = 0; // Initialize coordinates at the origin

        // Iterate through each character in the path
        for (int i = 0; i < path.length(); i++) {
            // current directic
```



```

        // Update x and y based on the direction
        if (dir == 'S') {
            y--; // Move South (decrease y)
        } else if (dir == 'N') {
            y++; // Move North (increase y)
        } else if (dir == 'E') {
            x++; // Move East (increase x)
        } else if (dir == 'W') {
            x--; // Move West (decrease x)
        }
    }

    // Calculate displacement using the formula: sqrt(x^2 + y^2)
    int x2 = x * x;
    int y2 = y * y;
    return (float) Math.sqrt(x2 + y2);
}

public static void main(String[] args) {
    String path = "WNEENESENNN"; // Given route
    System.out.println(getShortestPath(path)); // Output: 5
}
}

```

Explanation of the Code:

1. **Initialization:** The coordinates (x) and (y) start at (0, 0).
2. **Loop through the path:**

- For each character in the path:

- **S (South):** (y--)

- N (North): (y++)
- E (East): (x++)
- W (West): (x--)

Visualization of the Displacement Calculation

Question 2

Given a route containing 4 directions (E, W, N, S), find the **shortest** path to reach destination.

"WNEENESENNN"

$$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

$$\sqrt{(3-0)^2 + (4-0)^2}$$

$$= \sqrt{9+16}$$

$$= \sqrt{25} = 5$$

STRINGS

Time and Space Complexity Analysis

- Time Complexity: $O(n)$
 - The function iterates through each character in the string once, where (n) is the length of the string. The operations within each iteration take constant time.

- **Space Complexity:** $O(1)$
 - The function uses a constant amount of space, only storing the coordinates and the calculated result. No additional space is used that grows with the input size.

Conclusion

This solution efficiently calculates the shortest path using directions on a 2D grid. By visualizing and understanding the Cartesian plane, we ensure that the calculation remains intuitive and correct. The code simplifies the complex path into a single straight-line distance, providing the most efficient route from the start to the destination.

8. String Function: Compare Strings

String Comparison in Java

When comparing strings in Java, it's important to understand how the comparison works, especially when using the `==` operator and the `.equals()` method. Let's dive into this concept with an example.

Code Explanation

COPY

```
public class AreEqual {  
    public static void main(String[] args) {  
        String s1 = "Rohit";  
        String s2 = "Rohit";  
    }  
}
```

```
String s3 = new String("Rohit");

// Comparing strings using '=='
if (s1 == s2) {
    System.out.println("Strings are Equal");
} else {
    System.out.println("Strings are not Equal");
}

if (s1 == s3) {
    System.out.println("Strings are Equal");
} else {
    System.out.println("Strings are not Equal");
}

// Comparing strings using '.equals()'
if (s1.equals(s3)) {
    System.out.println("Strings are Equal");
} else {
    System.out.println("Strings are not Equal");
}
}
```

Output Explanation

1. Comparing `s1` and `s2` using `==` :

- Output: "Strings are Equal"

2. Comparing `s1` and `s3` using `==` :

- Output

3. Comparing `s1` and `s3` using `.equals()` :

- **Output:** "Strings are Equal"

Why Does This Happen?

1. Understanding `==` Operator:

- In Java, the `==` operator checks **reference equality** when used with objects, including strings.
- This means it checks if both variables (`s1` and `s2`) point to the **same memory location** (object).
- **Example:**
 - `s1` and `s2` are assigned the same value "Rohit" . Since they are string literals and not created using the `new` keyword, Java's **String Interning** mechanism kicks in.

2. String Interning:

- Java maintains a **string pool** in memory. When a string literal (e.g., "Rohit") is created, it checks if that string already exists in the pool. If it does, it reuses the same reference.
- Hence, `s1` and `s2` point to the same string in the pool, making `s1 == s2` return `true` .

3. `new String("Rohit")` :

- When `s3` is created using `new String("Rohit")` , it explicitly creates a **new object** in memory, even though the value is the same as `s1` and `s2` .



- As a result, `s1` and `s3` refer to **different objects** in memory, so `s1 == s3` returns `false`.

4. Using `.equals()`:

- The `.equals()` method in Java compares **content** rather than references.
- When we use `s1.equals(s3)`, it checks if the **values** stored in `s1` and `s3` are the same.
- Since both `s1` and `s3` contain "Rohit", `.equals()` returns `true`, even though they are different objects.

Why Should We Use `.equals()` for String Comparison?

- The `==` operator checks for **reference equality**, meaning it only returns `true` if both objects are the same (pointing to the same memory location).
- `.equals()` checks for **value equality**, ensuring that the actual content of the strings is compared, making it the preferred method for string comparisons in Java.

Summary

- `==`: Checks if two string references point to the same memory location (reference equality).
- `.equals()`: Checks if two strings have the same content (value equality).



- **String Interning:** String literals are stored in the **string pool**, and references to identical string literals point to the same object in memory.
-

9. String Function: Substring

Understanding Substrings in Java

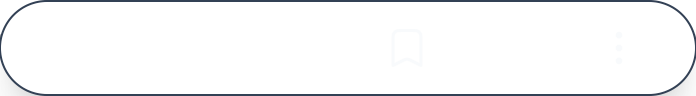
A **substring** is a sequence of characters within a string. It is essentially a smaller part or segment of the original string. Substrings are commonly used when you need to extract specific parts of a string based on given indices.

What is a Substring?

For example, if we have a string "RohitGawande", the following are examples of substrings:

- "hit"
- "Gaw"
- "Rohit"

A substring is determined by specifying two indices:

- **Starting Index (si):** The index where the substring begins.
- **Ending Index (ei):** The index where the substring ends, but it's exclusive,  at this index.

Example Substrings

Let's break down a few examples using the string "RohitGawande":

1. If we extract the substring from index 0 to 5 (`substring(0, 5)`), it includes characters at indices 0, 1, 2, 3, and 4:
 - Output: "Rohit"
2. If we extract a substring from index 3 to 7 (`substring(3, 7)`):
 - Output: "itGa"

Custom Substring Function

Here is a custom implementation to extract a substring given a start and end index:

COPY

```
public class Q2 {  
    public static String subString(String str, int si, int ei) {  
        String subst = "";  
        for (int i = si; i < ei; i++) {  
            subst += str.charAt(i);  
        }  
        return subst;  
    }  
  
    public static void main(String[] args) {  
        String str = "Hello Rohit";  
        // Using the custom function  
        System.out.println(subString(str, 2, 5)); // Output: "llo"  
        // Using Java's built-in substring method
```



```

        System.out.println(str.substring(0, 5)); // Output: "Hello"
    }
}

```

Explanation of the Code:

1. Custom Substring Function (`substring`):

- The function takes the string (`str`), a starting index (`si`), and an ending index (`ei`).
- A loop runs from `si` to `ei - 1`, and each character is added to the `subst` string.
- Finally, the function returns the substring (`subst`).

2. Using the Built-in Substring Method:

- Java provides an in-built method called `substring(int beginIndex, int endIndex)` in the `String` class.



- This method performs the same operation: it extracts a part of the string starting from `beginIndex` and ending before `endIndex`.
- Example: `str.substring(0, 5)` returns "Hello".

Benefits of Using the Built-in Method

While the custom method works fine, it's better to use Java's in-built `substring` method because:

- **Efficiency:**  ter compared to manually looping through the string.

- **Readability:** It's easier to understand and shorter in code.

Summary

- A **substring** is a portion of a string defined by a starting index and an exclusive ending index.
 - Java provides an in-built `substring` method to extract substrings efficiently.
 - Using the built-in method is recommended for better performance and code clarity.
-

10. Find the Largest String

Finding the Largest String Lexicographically

When comparing strings, the largest string lexicographically is the one that would come last if arranged in alphabetical order. Lexicographic comparison is similar to dictionary order, where:

- Strings are compared character by character.
- If two strings have the same prefix, the longer string is considered larger.

Examples:

- "apple", "man", "xyz" - "xyz" is the largest string.

- "Aaabcd" and "Aaabce" : "Aaabce" is larger because it differs in the last character and 'e' comes after 'd' .

Using compareTo() Method:

To compare strings lexicographically in Java, we use the `compareTo()` method:

- `str1.compareTo(str2)` :
 - Returns 0 if `str1` is equal to `str2` .
 - Returns a negative value if `str1` is smaller than `str2` .
 - Returns a positive value if `str1` is larger than `str2` .

For example:

- `"Rohit".compareTo("Rohit1")` will return a negative value because "Rohit" is lexicographically smaller than "Rohit1" .

Code Explanation:

COPY

```
public class Q3 {  
    public static void main(String[] args) {  
        // For a given set of strings, print the largest string le  
        // Using str.compareTo(str2) for comparison:  
        // 0: equal, <0: str1 < str2, >0: str1 > str2  
  
        String str1 = "Rohit";  
        String str2 = "Rohit1";  
        System.out.println(str1.compareTo(str2)); // Output: -1 (s1
```

```

// Array of strings
String Fruits[] = {"Mango", "Banana", "Apple"};
String Largest = Fruits[0]; // Assume the first string is the largest

// Loop through the array to find the largest string
for (int i = 0; i < Fruits.length; i++) {
    // Compare the current largest string with each string
    if (Largest.compareTo(Fruits[i]) < 0) {
        Largest = Fruits[i]; // Update if a larger string is found
    }
}
System.out.println(Largest); // Output: "Mango"
}
}

```

Explanation of the Code:

1. Comparison Using `compareTo()`:

- The code first demonstrates how `compareTo()` works by comparing "Rohit" with "Rohit1", which outputs -1 indicating that "Rohit" is smaller.

2. Finding the Largest String:

- An array `Fruits` contains "Mango", "Banana", and "Apple".
- The initial largest string is assumed to be the first element ("Mango").
- The loop iterates through each string in the array. If any string is lexicographically greater than the current `Largest`, it updates the `Largest` variable.

- Finally, it prints the largest string: "Mango" .

Time Complexity:

The time complexity of this solution is **$O(n)$** , where n is the number of strings in the array. This is because:

- The code iterates through the array once, comparing each string using `compareTo()` , which itself takes **$O(m)$** time, where m is the average length of the strings.
- Hence, the overall complexity is **$O(n * m)$** , where:
 - n is the number of strings.
 - m is the average length of a string.

This approach is efficient for smaller arrays and shorter strings, but for large datasets, optimizations or other approaches might be necessary.

11. Why Strings are Immutable

Let's combine both the detailed theory and memory visualization to fully explain how strings and `StringBuilder` work in Java, including the concept of string interning, memory allocation, and efficiency differences. This will include both scenarios with complete explanations and diagrams to visualize the differences.



String Storage and StringBuilder in Java

1. Understanding String Storage

In Java, strings are stored in a special area of memory called the **Heap**. Within the heap, there is a part called the **String Pool** (also known as the **Intern Pool**). When strings are created using literals (e.g., "Rohit"), Java optimizes memory usage by storing only one instance of each literal in the String Pool.

String Interning Process

- **String Literals:**
 - When we write `String str1 = "Rohit";` , the string "Rohit" is placed in the String Pool if it doesn't already exist.
 - If we write `String str2 = "Rohit";` , no new object is created. Instead, `str2` will reference the same "Rohit" object in the String Pool.
- **New String Object:**
 - If we use the `new` keyword, like `String str3 = new String("Rohit");` , Java creates a new "Rohit" object outside of the String Pool in the heap, even if an identical string already exists in the pool.
 - This is why comparing `str1` and `str3` with `==` returns `false` — they point to different objects.

Visualizatio



Stack		Heap (String Pool)
----- -----		
str1 --> "Rohit"		"Rohit" <--- Points to the same object as str1
str2 --> "Rohit"		
str3 --> New "Rohit" object (outside String Pool, unique object)		

Explanation of the Visualization:

1. Stack Area:

- Holds references for `str1`, `str2`, and `str3`.
- `str1` and `str2` point to the **same** "Rohit" object in the String Pool due to interning.
- `str3` points to a **new** "Rohit" object created with the `new` keyword, which resides outside the String Pool.

2. Heap Area:

- The **String Pool** contains only one instance of the literal "Rohit", referenced by both `str1` and `str2`.
- A separate "Rohit" object exists outside the String Pool, referenced by `str3`.

Why `==` Doesn't Work for Comparing Strings

- `==` compares references, not values.
- Since `str1` and `str2` point to the same object in the String Pool, `str1 == str2` returns `true`. However, `str1 == str3` returns `false` because `str3` points to a different object in the heap.

- To check if two strings have the same content, use `.equals()`, which compares values rather than references.

2. Why Strings Are Immutable and Inefficient for Modifications

Immutability of Strings

- Strings in Java are **immutable**, meaning once created, they cannot be modified.
- Any modification, such as concatenation (`+=`), results in the creation of a **new** string object, leaving the original unchanged.

Example: Appending Characters Using a String

COPY

```
String str = "rohit";
for (char ch = 'a'; ch <= 'z'; ch++) {
    str += ch;
}
```

Visualization:

COPY

```
Stack | Heap (String Pool)
```

```
str --> "rohit" | "rohit" (Initial String)
```

```
str --> "rohit" + "a" | New object created ("rohita")
```



```
str    --> "rohita" + "b" | New object created ("rohitab")
...    | ... (continues for each character appended)
```

- **Explanation:**

- Initially, "rohit" is stored in the String Pool.



and stored separately in the heap.

- This process repeats for every character (a to z), leading to 26 new objects being created.

Time Complexity of Using Strings for Modification:

- For each concatenation, the existing string and the new character are combined, resulting in $O(n^2)$ time complexity due to repeated object creation and copying of contents.

12. String Builder

3. Efficient String Modification Using `StringBuilder`

To avoid the inefficiencies of using strings for frequent modifications, we use `StringBuilder`.

Why `StringB`



- Unlike `String`, `StringBuilder` is mutable, meaning it can be modified without creating new objects.
- It maintains a **single buffer** where characters are appended, avoiding the overhead of creating new objects each time.

Example: Appending Characters Using `StringBuilder`


COPY

```
StringBuilder sb = new StringBuilder();
for (char ch = 'a'; ch <= 'z'; ch++) {
    sb.append(ch);
}
System.out.println(sb);
```

Visualization:

COPY

Stack		Heap
----- -----		
sb	-->	StringBuilder object ("abcdefghijklmnopqrstuvwxyz")

- **Explanation:**
 - Only **one** `StringBuilder` object is created in the heap.
 - The loop appends each character (a to z) directly to this buffer, w objects.

Time Complexity of Using `StringBuilder` :

- The process runs in **$O(n)$** time, as it only iterates once and modifies the existing buffer.

Summary

Operation	Data Structure	Time Complexity
String Concatenation (<code>+=</code>)	<code>String</code>	$O(n^2)$
Append (<code>append()</code>)	<code>StringBuilder</code>	$O(n)$

Conclusion:

- For frequent modifications or dynamic string changes, always prefer using `StringBuilder` to maintain efficiency and reduce memory overhead.
- Use strings (`String`) when the content is fixed and doesn't require multiple changes.

By understanding these differences and visualizing memory usage, it's clear why `StringBuilder` is preferred for string manipulations and how Java manages strings efficiently through interning.



13. Convert Letters to Uppercase

Understanding Strings and String Manipulation

Strings in Java are immutable, meaning once they are created, they cannot be changed. This immutability ensures that strings are safe to use in multi-threaded environments and are efficient in memory usage when using the same values repeatedly.

However, when we need to manipulate strings frequently, such as converting characters or modifying parts of the string, creating new string objects repeatedly is inefficient. For such scenarios, Java provides a more efficient option called `StringBuilder`. Unlike regular strings, `StringBuilder` is mutable, allowing us to modify its content without creating new objects.

Problem: Converting the First Letter of Each Word to Uppercase

Suppose we have a string like `"hi, i am rohit"`. We want to convert the first letter of each word to uppercase so that the output becomes `"Hi, I Am Rohit"`.

1. String Structure:

- The given string consists of multiple words separated by spaces.
- Each word may start with a lowercase letter that needs to be converted to uppercase

2. Approach:

- We first convert the first character of the string to uppercase, as it is the beginning of the first word.
- We then iterate through the rest of the string.
- Whenever we encounter a space, it indicates that the next character is the start of a new word, which should be converted to uppercase.

3. In-Built Function for Character Conversion:

- Java provides `Character.toUpperCase(char ch)` to convert a given character to uppercase.
- This method is useful when we need to selectively modify certain characters while iterating through the string.

Visualization: Memory Representation Using StringBuilder

COPY

Input String in Memory (Heap Area):

"hi, i am rohit"

StringBuilder Initialization:

```
+-----+
| h | i | , | ... (Dynamic Capacity)
+-----+
```

1. Convert 'h' to 'H' and append:

```
+-----+
| H | i | , | ...
+-----+
```



2. Iterate through the string:

- Character at index 3: Space (' ')
- Find next character ('i') and convert it to 'I'
- Append the space and 'I'

```
+-----+
| H | i | , | | I | ...|
+-----+
```

3. Repeat the process for each space and the character following it

Final StringBuilder in Memory:

```
+-----+
| H | i | , | | I | | ... |
+-----+
```

Converted String: "Hi, I Am Rohit"

Code: Converting the First Letter of Each Word to Uppercase

COPY

```
package STRINGS;

public class Q4 {
    // Function to convert the first letter of each word to uppercase
    public static String convertToUppercase(String str) {
        // Use a StringBuilder to build the new string
        StringBuilder sb = new StringBuilder("");
    }
```

```
// Convert the first character of the string to uppercase &
char ch = Character.toUpperCase(str.charAt(0));
sb.append(ch);

// Loop through the rest of the string starting from the second character
for (int i = 1; i < str.length(); i++) {
    // If the character is a space and the next character is not a space
    if (str.charAt(i) == ' ' && i < str.length() - 1) {
        sb.append(str.charAt(i)); // Append the space
        i++; // Move to the next character
        sb.append(Character.toUpperCase(str.charAt(i))); // Append the next character in uppercase
    } else {
        // Otherwise, append the current character as it is
        sb.append(str.charAt(i));
    }
}

// Convert StringBuilder to string and return the final result
return sb.toString();
}

public static void main(String[] args) {
    // Given string
    String str = "hi, i am rohit";

    // Calling the function and displaying the result
    String result = toUpperCase(str);
    System.out.println("Original String: " + str);
    System.out.println("String after converting first letter of each word to uppercase: " + result);
    System.out.println(result);
}
}
```

Sample Output

[COPY](#)

Original String: hi, i am rohit

String after converting first letter of each word to uppercase:

Hi, I Am Rohit

Explanation of the Code

1. StringBuilder Initialization:

- We use `StringBuilder` to build the modified string efficiently.
- We start by converting the first character of the string to uppercase using `Character.toUpperCase(str.charAt(0))` and appending it.

2. Loop Through the String:

- We iterate from index `1` to the end of the string.
- If the character at the current index is a space and there is another character following it, we convert that next character to uppercase.
- We append the space and the converted uppercase character to our `StringBuilder`.
- For all other characters, we append them as they are.

3. Return the Modified String:

- Finally, we return the modified string by calling the `toString()` method on the `StringBuilder`.

Time Complexity Analysis

- The time complexity of this approach is $O(n)$, where n is the length of the string. The `StringBuilder` allows for efficient manipulation without creating multiple string objects, making the process optimized for performance.
-

14. String Compression

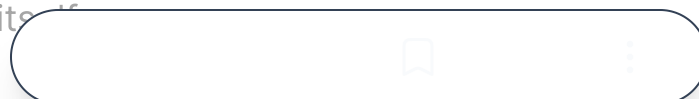
String compression is a process where consecutive repeated characters in a string are replaced with the character followed by the count of its occurrences. The goal is to reduce the size of the string by summarizing repeating patterns, making the string more compact.

For example:

- **Input:** "aaabbccdd"
- **Output:** "a3b2c3d2"

Rules for String Compression:

1. If a character appears consecutively, replace it with the character followed by the number of times it appears.
 - Example: "aaa" becomes "a3".
2. If a character appears only once, it should still be shown as the character itself.



3. If the compressed version of the string is not shorter than the original, you can opt to return the original string (for efficiency).

Problem Clarification:

Given a string, you need to compress it based on the rules mentioned. For example:

- **Example 1:** "aaaabbccccc" → "a4b2c5"
- **Example 2:** "abcd" should remain as "abcd" since there are no consecutive repetitions.

Why Use `StringBuilder`?

In Java, strings are immutable, meaning that every time we concatenate, a new string object is created. If we concatenate repeatedly, this can lead to inefficient memory usage and slower performance. Using `StringBuilder` allows us to modify the string efficiently, as it is mutable and designed for such operations.

Visualization: Memory Representation Using `StringBuilder`

COPY

Input String in Memory:
"aaabbccdd"

`StringBuilder` Initialization:



1. Start iterating through the string:
 - Character at index 0: 'a'
 - Count consecutive 'a's: 3
 - Append 'a' and '3' to StringBuilder

```
+-----+
|  a  | 3  |          | ...
+-----+
```

2. Move to character 'b':
 - Count consecutive 'b's: 2
 - Append 'b' and '2'

```
+-----+
|  a  | 3  |  b  | 2  | ...
+-----+
```

3. Repeat for all characters until the end of the string.

Final Compressed String:

"a3b2c3d2"

Code: String Compression Using StringBuilder

COPY

```
import java.util.*;
```

```
public class
```

```
// Function to compress the given string
```

```
public static String compress(String str) {  
    // Using StringBuilder for efficient string manipulation  
    StringBuilder compressed = new StringBuilder();  
  
    // Iterate through the string  
    for (int i = 0; i < str.length(); i++) {  
        // Start counting occurrences of the current character  
        int count = 1;  
        while (i < str.length() - 1 && str.charAt(i) == str.charAt(i+1)) {  
            count++;  
            i++;  
        }  
  
        // Append the character  
        compressed.append(str.charAt(i));  
  
        // Append the count only if it's greater than 1  
        if (count > 1) {  
            compressed.append(count);  
        }  
    }  
  
    // Convert StringBuilder to string and return  
    return compressed.toString();  
}  
  
public static void main(String[] args) {  
    // Given string  
    String str = "aaabbccdd";  
  
    // Calling the compress function and printing the result  
    System.out.println("Original String: " + str);  
    System.out.println("Compressed String: " + compress(str));  
}
```

```
}  
}
```

Sample Output

[COPY](#)

Original String: aaabbccdd

Compressed String: a3b2c3d2

Explanation of the Code:

1. `StringBuilder` Initialization:

- We use `StringBuilder` to create an efficient way to build the compressed string.

2. Iterating Through the String:

- We iterate through each character in the string.
- For each character, we check if it is repeated consecutively.
- If it is, we count the repetitions and move the index accordingly.

3. Appending to `StringBuilder`:

- We append the character to the `StringBuilder`.
- If the count of the character is greater than 1, we append the count.

4. Return the Compressed String:

- We convert the `StringBuilder` to a string and return it as the final result.

Time Complexity Analysis

- The time complexity of this approach is **$O(n)$** , where **n** is the length of the string. The use of `StringBuilder` ensures that the method is efficient in terms of both time and space.

Points to Note:

- This solution handles cases where characters are not repeated (e.g., "abcd") and keeps the original format.
- It efficiently compresses strings with repeated characters and does not create new string objects repeatedly, ensuring optimal performance.

By using this approach and code, you can efficiently compress strings while maintaining the integrity of the original input and avoiding unnecessary memory usage.

15. Practice Questions

1. **Reverse a String:** Write a function to reverse a string.
2. **Count Vowels:** Write a function to count the number of vowels in a string.



3. **Longest Common Prefix:** Find the longest common prefix among an array of strings.
-

Conclusion

Strings are a fundamental concept in Java and understanding them is crucial to solving many problems in DSA. We explored string operations, methods, and practical questions that will help you navigate the world of string manipulation in programming. Continue practicing these concepts to enhance your skills and stay tuned for the next chapter in our DSA journey!

Happy Coding!

Related Posts

1. Chapter 1: Variables and Data Types

A quick guide on Java variables and data types, covering primitives, non-primitives, and type conversions with examples.

2. Chapter 21: LinkedList (Part 1)

An introduction to LinkedLists in Java, including implementation, types (singly and doubly), and basic operations.

Other Series



1. **Full Stack Java Development:** Comprehensive tutorials and projects to build and deploy Java applications.
2. **Full Stack JavaScript Development:** Covers front-end and back-end JavaScript technologies for building dynamic web apps.

Connect With Me

- **LinkedIn:** Follow me
- **GitHub:** Check out my projects
- **LeetCode:** See my coding progress

Stay connected for updates and new posts!

Rohit Gawande

Full Stack Java Developer, Code Enthusiast, Blogger

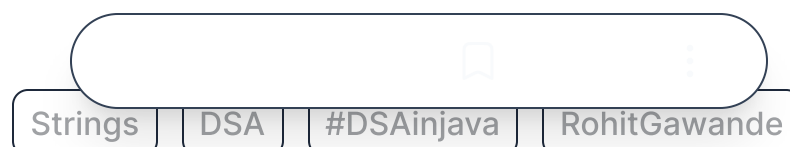
Subscribe to my newsletter

Read articles from **Rohit Gawande's blog** directly inside your inbox.

Subscribe to the newsletter, and don't miss out.

rohitgawande2004@gmail.cc

SUBSCRIBE



Written by



Rohit Gawande

"Web Developer | JavaScript Enthusiast | Passionate about building responsive, user-friendly websites. On a mission to master front-end development, one line of code at a time. Sharing my journey through tutorials, projects, and insights. Let's connect and grow together in the world of tech!"

ARTICLE SERIES

DSA(Data Structure and Algorithm) In JAVA

1

Chapter 32: Heaps

Introduction to Priority Queues and Heaps Introduction to Priority Queues (PQ) A Priority Queue (PQ)...

2

Chapter 2: Operators in Java

Operators are symbols that perform operations on variables and values. The

[Show all 32 posts](#)

35

Chapter 31: Binary Search Trees(Part 2)

Related Posts in My Series: DSA in Java Series: Chapter 2: Operators in Java – Learn about the diff...

36

Chapter 44: Graphs (Part 2)

In the last part of the DSA series, we explored the basic concepts of graphs, such as types of graph...

©2024 Rohit Gawande's blog

[Archive](#) • [Privacy policy](#) • [Terms](#)



Powered by Hashnode - Build your developer hub.

[Start your blog](#)[Create docs](#)