L -49: String Functions, String Builder vs String Buffer | Java plus DSA Placement | FAANG

Time Interval: 00:00:00 - 00:17:59

Introduction

The video starts with a quick **revision of string comparison methods** in Java, specifically:

- `==` **operator**: Compares reference memory locations.
- `.equals()` **method**: Compares the actual values of strings.

Before diving into **string functions**, the instructor demonstrates **how to take user input in Java using the `Scanner` class**.

Taking User Input in Java

To take **string input from the user**, we use `Scanner`:

```
Scanner sc = new Scanner(System.in);
String word = sc.next();
String line = sc.nextLine();
```

Here,

- `next()` takes a **single word input**.
- `nextLine()` takes an **entire line of text**.

Issue Demonstrated:
If we use `next()` before `nextLine()`, the **newline character** left in the buffer prevents `nextLine()` from working properly.

Solution: Swap the order of input statements, using `nextLine()` first.

Important String Functions in Java

The instructor explains several **built-in string functions**, along with live coding examples.

1️⃣ `toLowerCase()` and `toUpperCase()`

These functions convert a string to **lowercase** or **uppercase**, respectively.

Example:

```
String str = "Hello";
System.out.println(str.toLowerCase()); // hello
System.out.println(str.toUpperCase()); // HELLO
```

- These methods **return a new string** without modifying the original string.

2️⃣ `charAt(index)`

Retrieves the character at a **specific index** in a string.

Example:

```
String name = "Java";
System.out.println(name.charAt(1)); // Output: 'a'
```

- Strings in Java are **zero-indexed**, meaning the first character is at **index 0**.

3️⃣ `indexOf(char)`

Finds the **first occurrence** of a character.

Example:

```
String text = "Java Programming";
System.out.println(text.indexOf('a')); // Output: 1
```

- If the character **does not exist**, it returns `-1`.

4️⃣ `lastIndexOf(char)`

Finds the **last occurrence** of a character.

Example:

```
String text = "Java Programming";
System.out.println(text.lastIndexOf('a')); // Output: 10
```

5️⃣ `length()`

Returns the **total number of characters** in a string.

Example:

```
String message = "Hello World";
System.out.println(message.length()); // Output: 11
```

- The last index in a string is always `length - 1`.

Converting Strings to Character Arrays

Java allows converting a string into a **character array**:

```
String word = "Hello";
char[] chars = word.toCharArray();
```

Each character of `"Hello"` is stored as an array element.

String Comparison Using `compareTo()`

The `compareTo()` method is used to **lexicographically compare two strings**:

```
String s1 = "Apple";
String s2 = "Banana";
System.out.println(s1.compareTo(s2));
```

Possible outputs:

- **0** → Strings are equal.
- **Positive number** → `s1` is lexicographically greater than `s2`.
- **Negative number** → `s1` is lexicographically smaller than `s2`.

Example:

- `"Apple".compareTo("Banana")` returns a **negative value** since `"Apple"` comes before `"Banana"` alphabetically.
- `"Banana".compareTo("Apple")` returns a **positive value**.

Removing Extra Spaces Using `trim()`

The `trim()` function **removes leading and trailing spaces** from a string:

```
String sentence = "  Hello World  ";
System.out.println(sentence.trim()); // Output: "Hello World"
```

Checking Prefix and Suffix in Strings

1️⃣ `startsWith(prefix)`

Checks if a string **begins** with a given substring.

Example:

```
String text = "Java Programming";
System.out.println(text.startsWith("Java")); // Output: true
```

2️⃣ `endsWith(suffix)`

Checks if a string **ends** with a given substring.

Example:

```
System.out.println(text.endsWith("ming")); // Output: true
```

Understanding Substrings

A **substring** is a portion of a string.

Example:

```
String text = "Hello World";
System.out.println(text.substring(0, 5)); // Output: "Hello"
```

- The substring method follows the rule: **start index (inclusive), end index (exclusive).**

Mutable Strings: `StringBuilder` vs `StringBuffer`

Java provides **mutable** string classes:

1. `StringBuilder` (Faster, but not thread-safe).
2. `StringBuffer` (Thread-safe but slightly slower).

Why are Strings Immutable?

- Strings are **immutable** to **optimize memory** and **ensure security** in Java.

Using `StringBuilder` for Faster String Modification

```
StringBuilder sb = new StringBuilder("Hello");
sb.append(" World");
System.out.println(sb); // Output: Hello World
```

- Unlike `String`, modifications happen **in-place** without creating a new object.

Using `StringBuffer` for Thread-Safety

```
StringBuffer sb = new StringBuffer("Hello");
sb.append(" World");
System.out.println(sb); // Output: Hello World
```

- **Key Difference**: `StringBuffer` is **synchronized**, making it safe for **multi-threaded environments**.

Conclusion

- Java provides many **built-in functions** for working with strings.
- `compareTo()` is used for **lexicographic comparison**.
- `trim()` removes **extra spaces** from a string.
- **Mutable alternatives** (`StringBuilder`, `StringBuffer`) allow **efficient string modifications**.
- `StringBuilder` is **faster**, while `StringBuffer` is **thread-safe**.

Example Exploratory Questions

1. **E1**: Why is `StringBuffer` thread-safe while `StringBuilder` is not?
2. **E2**: How does `compareTo()` determine string order?
3. **E3**: What are real-world use cases of `StringBuilder`?