L -48: Introduction to Java Strings | Java plus DSA Placement | FAANG

Time Interval: 00:00:00 - 00:23:18

Introduction to Strings

The video begins with the instructor introducing the concept of **strings** in Java. The first question asked is: **"What is a string?"** The explanation follows that a string is nothing but a **sequence of characters**.

Characters Recap: Before understanding strings, it's essential to understand **characters**. A **character** in Java is a single entity enclosed within **single quotes** (`' '`).
Example:char ch = 'A';
char ch2 = 'B';

- Here, `ch` and `ch2` are **character variables**.

String Definition: A **string** is a **sequence of characters** enclosed in **double quotes** (`" "`).
Example: String str = "Hello";

- Here, `"Hello"` is a **string** because it consists of multiple characters enclosed within double quotes.

Creating a String in Java

There are multiple ways to **create a string** in Java. The simplest and most common way is:

String name = "John";

Here, `"John"` is stored in memory, and the variable `name` is referring to it.

Example - String Storage in Memory

If we print `System.out.println(name);`, it will display:

John

This example highlights that strings are **easy to create and use** in Java.

Understanding String Storage in Java

Java handles strings differently compared to primitive data types. The instructor explains **how strings are stored in memory**.

- When we create a string using **double quotes**, Java **stores it in a special memory area called the "String Constant Pool."**
- If the same string already exists in the **pool**, Java does **not create a new copy**; instead, it **reuses the existing one**.
- This optimization **saves memory** by preventing duplicate strings.

Example:

String str1 = "Hello";
String str2 = "Hello";

Here, both `str1` and `str2` **point to the same memory location** in the **String Constant Pool**, meaning only **one copy of `"Hello"` exists**.

String Objects and "new" Keyword

Another way to create strings in Java is by using the `new` keyword:

String str3 = new String("Hello");

This method **forces Java to create a new string object** in **Heap memory**, even if `"Hello"` already exists in the String Constant Pool.

Key Difference:
String str1 = "Hello"; // Stored in String Constant Pool
String str2 = new String("Hello"); // Stored in Heap Memory

- Even though `str1` and `str2` **have the same value**, they point to **different memory locations**.

String Comparison (`==` vs `.equals()`)

Java provides two ways to compare strings:

1. **Using `==` Operator**: Compares the **memory location** (reference comparison).
2. **Using `.equals()` Method**: Compares the **actual content** (value comparison).

Example:

```
String a = "Java";
String b = "Java";
String c = new String("Java");

System.out.println(a == b); // true (same reference)
System.out.println(a == c); // false (different objects)
System.out.println(a.equals(c)); // true (same value)
```

- `a == b` returns **true** because both refer to the same object in the **String Constant Pool**.
- `a == c` returns **false** because `c` was created using `new`, which places it in **Heap Memory**.
- `a.equals(c)` returns **true** because `.equals()` checks the **actual string content**, not the memory address.

String Immutability in Java

Strings in Java are immutable, meaning once a string is created, it **cannot be changed**. Any modification creates a **new string object**.

Example:

```
String s1 = "Hello";
s1 = s1 + " World";
System.out.println(s1);
```

Output:

```
Hello World
```

Explanation: Instead of modifying `"Hello"`, Java creates a **new string `"Hello World"`** and assigns it to `s1`.

Memory Optimization in Java Strings

The **String Constant Pool** ensures that duplicate strings **do not take extra memory**.

Example:

```
String x = "Code";
String y = "Code";
System.out.println(x == y); // true (same object)
```

Here, `"Code"` is stored **only once**, and both `x` and `y` point to the **same memory location**.

However, when using `new`:

```
String z = new String("Code");
System.out.println(x == z); // false (different objects)
```

A **new object is created in Heap Memory**, making the `==` comparison **false**.

Garbage Collection and String Deallocation

When a string **loses all references**, Java's **Garbage Collector (GC)** automatically removes it.

Example:

```
String temp = "Garbage";
temp = "Collection";
```

Here, `"Garbage"` is **no longer referenced**, so the **Garbage Collector removes it**.

Important Interview Question: How Many Objects Are Created?

Consider the following code:

```
String s1 = new String("ABC");
String s2 = "ABC";
```

- `"ABC"` is **stored in the String Constant Pool**.
- `s1` **creates a new object in Heap Memory**, meaning **two objects are created**:
    1. One in the **String Constant Pool**
    2. One in **Heap Memory**

Thus, the correct answer is **two objects**.

Using `.equalsIgnoreCase()` and `.compareTo()`

- `.equalsIgnoreCase()` compares two strings **ignoring case sensitivity**.
- `.compareTo()` compares strings lexicographically.

Example:

```
String s1 = "Hello";
String s2 = "hello";
System.out.println(s1.equalsIgnoreCase(s2)); // true
System.out.println(s1.compareTo(s2)); // Non-zero value (case-sensitive comparison)
```

Conclusion

- Strings in Java are **immutable** and stored in a **String Constant Pool** to optimize memory.
- The `==` operator checks **references**, while `.equals()` checks **values**.
- Using `new String("text")` **creates a new object** instead of reusing an existing one.
- **Garbage Collector** removes unreferenced strings from memory.
- **String comparison and memory allocation** are crucial topics in **Java interviews**.

Example Exploratory Questions

1. **E1**: Why are Java strings immutable?
2. **E2**: How does Java optimize string storage with the String Constant Pool?
3. **E3**: What is the difference between `==` and `.equals()` for strings?