

# Tabel of Content

Unit No.	Topic
1	Introduction to Strings in Java
2	String Literal vs String Object
3	StringBuffer in Java
4	StringBuilder in Java
5	String vs StringBuffer vs StringBuilder
6	String Topics for Product-Based Companies
7	LeetCode Practice Problems
8	Java-Specific Tips & Best Practices



## 1 What is a String in Java

A **String** in Java is a **sequence of characters**, treated as an object of the String class in the java.lang package.

Java Strings are **immutable**, meaning once created, their values **cannot be changed**.

### Real-World Analogy

Think of a **string** as a **word written in ink** on paper — once written (created), you can read, compare, or copy it, but **can't change the ink directly** (immutable). To change it, you create a new paper (new String).

### String Declaration and Initialization

```
// Using string literal (stored in string pool)
String s1 = "Hello";

// Using new keyword (stored in heap)
String s2 = new String("World");
```

### Common String Methods

METHOD	DESCRIPTION	EXAMPLE
LENGTH()	Returns the number of characters	s.length()
CHARAT(INT INDEX)	Returns character at a specific index	s.charAt(1)
TOUPPERCASE()	Converts to uppercase	s.toUpperCase()
TOLOWERCASE()	Converts to lowercase	s.toLowerCase()
EQUALS()	Compares content (case-sensitive)	s1.equals(s2)
EQUALSIGNORECASE()	Compares ignoring case	s1.equalsIgnoreCase(s2)
CONTAINS()	Checks if string contains substring	s.contains("text")
SUBSTRING(START, END)	Extracts substring	s.substring(1, 4)
REPLACE(A, B)	Replaces characters	s.replace("a", "b")
SPLIT(" ")	Splits string into array	s.split(" ")
TRIM()	Removes leading/trailing spaces	s.trim()



## String Immutability Explained

```
String s = "Hello";  
s.concat(" World"); // does NOT change original string  
System.out.println(s); // Output: Hello
```

To reflect the change:

```
s = s.concat(" World");  
System.out.println(s); // Output: Hello World
```

## String Comparison

```
String s1 = "Hello";  
String s2 = "Hello";  
String s3 = new String("Hello");  
  
System.out.println(s1 == s2); // true (same object in pool)  
System.out.println(s1 == s3); // false (different object)  
System.out.println(s1.equals(s3)); // true (same content)
```

## Example Program

```
public class StringExample {  
    public static void main(String[] args) {  
        String name = "Java Programming";  
  
        System.out.println("Length: " + name.length());  
        System.out.println("Upper: " + name.toUpperCase());  
        System.out.println("First char: " + name.charAt(0));  
        System.out.println("Contains 'Java': " + name.contains("Java"));  
    }  
}
```

## Best Practices

- Prefer string **literals** for memory efficiency.
- Use equals() for comparison, **not** ==.
- Use StringBuilder for heavy string modifications (e.g., in loops).
- Avoid unnecessary string concatenations — it's memory-expensive.



## Summary

TOPIC	KEY POINT
<b>IMMUTABLE</b>	Once created, cannot be changed
<b>STORAGE</b>	String Pool (literal), Heap (new)
<b>METHODS</b>	Powerful built-in methods for processing
<b>COMPARISON</b>	equals() for content, == for reference check
<b>USE CASE</b>	Widely used in file I/O, user input, APIs, etc.

## 2 String Literal Vs String Object

### What is a String Literal?

A **String literal** is any sequence of characters enclosed in double quotes, e.g.:

```
String s1 = "Java";
```

- Stored in the **String Constant Pool (SCP)** inside the **Method Area** of JVM memory.
- If "Java" already exists in the SCP, it **does not create a new object** — it just returns a reference to the existing one.

### What is a String Object?

You can also create a String using the new keyword:

```
String s2 = new String("Java");
```

- Creates a **new object in the Heap** memory.
- Also refers to "Java" in the **SCP** (for internal character storage).
- So this creates **two objects**:  
→ One in Heap (via new)  
→ One in SCP (if not already present)

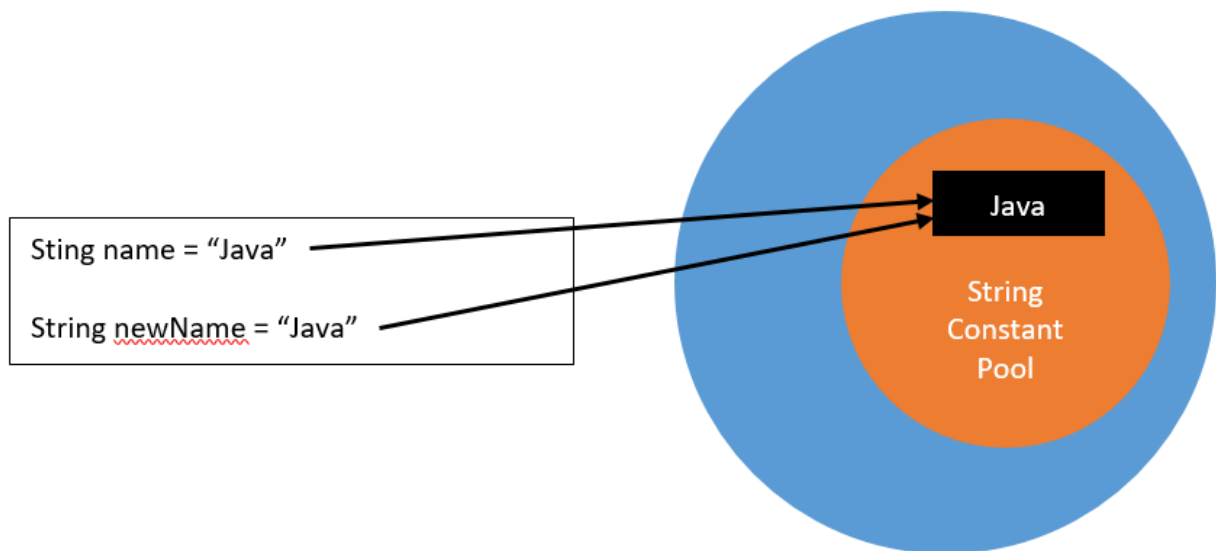
### Key Differences: Literal vs Object

FEATURE	STRING LITERAL ("JAVA")	STRING OBJECT (NEW STRING("JAVA"))
<b>MEMORY LOCATION</b>	String Constant Pool (SCP)	Heap + reference to SCP
<b>REUSE</b>	Reused if already exists	Always a new object
<b>EFFICIENCY</b>	Memory efficient	Less efficient (creates duplicate)
<b>COMPARISON USING ==</b>	Might return true	Always returns false
<b>EXAMPLE</b>	String s = "Java";	String s = new String("Java");

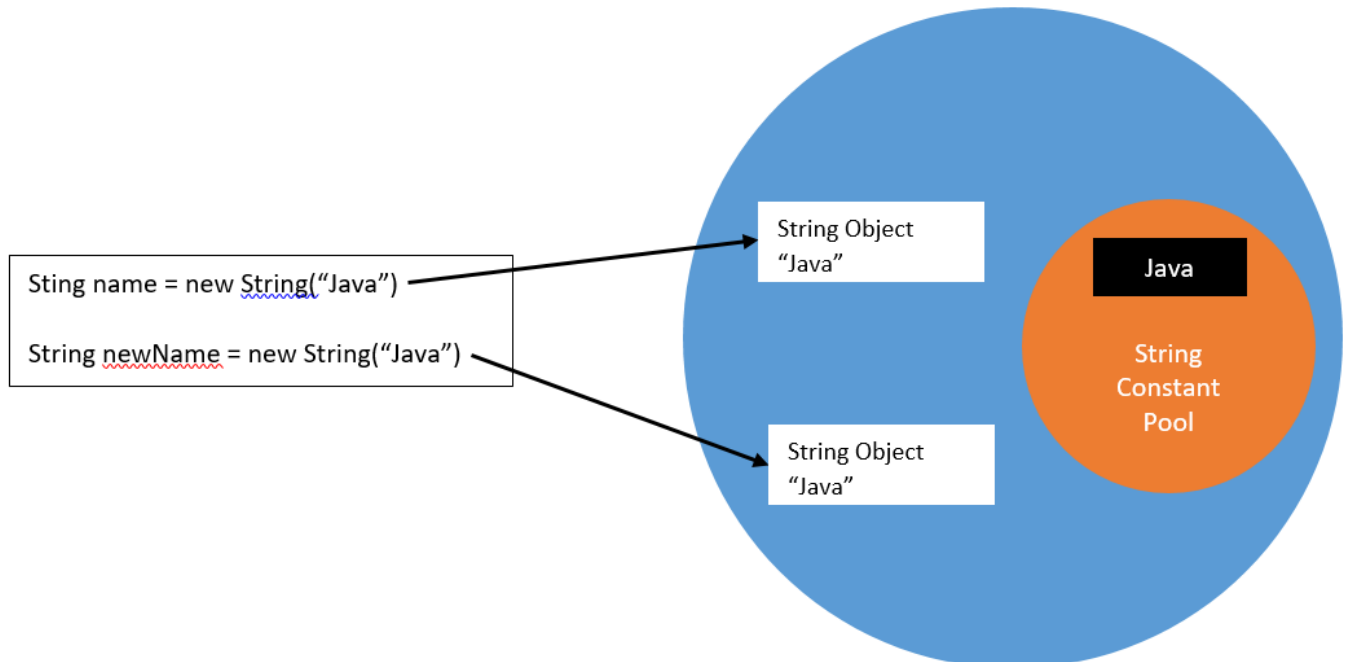


## Memory Diagram

### String Literal



### String Object



## Comparison Example

```
public class StringMemoryDemo {
    public static void main(String[] args) {
        String s1 = "Java";
        String s2 = "Java";
        String s3 = new String("Java");

        System.out.println(s1 == s2);      // true (same SCP object)
        System.out.println(s1 == s3);      // false (heap vs SCP)
        System.out.println(s1.equals(s3)); // true (content comparison)
    }
}
```

### Output:

```
true
false
true
```

## Real-World Analogy

Imagine the **SCP** as a **library**:

- When you ask for a book titled "Java":
  - If it already exists on the shelf, you're **given the same copy** (literal).
  - If you say, "I want a brand new one" (using `new`), then the library **prints a fresh copy** and gives it to you.

## Best Practices

- Use string **literals** when possible to save memory.
- **✗** Avoid unnecessary use of `new String()` unless you need a **separate object**.
- Always use `.equals()` to compare strings (not `==`).

### Bonus: `intern()` Method

You can force a string object to refer to SCP:

```
String s4 = new String("Java").intern();
```

Now `s4` will point to "Java" in SCP — just like a literal.



## Summary

TERM	MEANING
SCP	String Constant Pool — stores unique string literals
HEAP	General object storage area in memory
NEW STRING()	Always creates a new object in Heap
INTERN()	Moves or refers a string to the SCP
==	Compares reference (address)
EQUALS()	Compares content

## 3 StringBuffer in Java

### 1. Overview / Explanation

- StringBuffer is a **mutable** sequence of characters (unlike String, which is immutable).
- Part of java.lang package.
- Used when you need to **modify strings frequently** (e.g., appending, inserting, deleting).
- **Thread-safe** – methods are **synchronized**, so safe to use in multi-threaded environments.

**Use Case:** When building dynamic strings in a loop or multithreaded app – e.g., processing input, generating reports.

### 2. Declaration and Instantiation

```
StringBuffer sb1 = new StringBuffer();           // Empty buffer
StringBuffer sb2 = new StringBuffer("Hello");    // Initialized
StringBuffer sb3 = new StringBuffer(50);         // With capacity
```

### 3. Common Methods with Examples

#### append()

Adds text at the end.

```
sb1.append("Java");
System.out.println(sb1); // Java
```

#### insert()

Inserts text at a specific index.

```
sb1.insert(4, " Programming");
System.out.println(sb1); // Java Programming
```



**replace()**

Replaces part of the string between start and end index.

```
sb1.replace(0, 4, "Python");
System.out.println(sb1); // Python Programming
```

**delete()**

Deletes characters between start and end index.

```
sb1.delete(0, 7);
System.out.println(sb1); // Programming
```

**reverse()**

Reverses the entire content.

```
sb1.reverse();
System.out.println(sb1); // gnimmargorP
```

**length() and capacity()**

```
System.out.println(sb1.length()); // No. of characters
System.out.println(sb1.capacity()); // Buffer capacity
```

**charAt() and setCharAt()**

```
char ch = sb1.charAt(0);
sb1.setCharAt(0, 'X');
```

**4. Why Use StringBuffer Over String?**

OPERATION	STRING	STRINGBUFFER
<b>MUTABILITY</b>	Immutable	Mutable
<b>THREAD-SAFE</b>	Not thread-safe	Yes
<b>PERFORMANCE</b>	Slower in loops	Faster in loops

**5. StringBuffer vs StringBuilder**

FEATURE	STRINGBUFFER	STRINGBUILDER
<b>THREAD-SAFETY</b>	Yes (synchronized)	No
<b>PERFORMANCE</b>	Slower	Faster (in single-thread)
<b>USE CASE</b>	Multithreaded apps	Single-thread apps





## 4 StringBuilder in Java

### 1. Overview / Explanation

- StringBuilder is a **mutable** sequence of characters, just like StringBuffer.
- **Not thread-safe**, but **faster** than StringBuffer in single-threaded applications.
- Part of java.lang package.
- Ideal when you're performing **lots of modifications to strings** in a **single-threaded** context.

**Use Case:** Building or modifying strings inside loops, parsing files, generating HTML reports, etc.

### 2. Declaration and Instantiation

```
StringBuilder sb1 = new StringBuilder();           // Empty buffer
StringBuilder sb2 = new StringBuilder("Hello");    // With initial value
StringBuilder sb3 = new StringBuilder(50);         // With specific capacity
```

### 3. Common Methods with Examples

#### append()

```
sb1.append("Java");
System.out.println(sb1); // Java
```

#### insert()

```
sb1.insert(4, " World");
System.out.println(sb1); // Java World
```

#### replace()

```
sb1.replace(0, 4, "Hello");
System.out.println(sb1); // Hello World
```

#### delete()

```
sb1.delete(5, 11);
System.out.println(sb1); // Hello
```

#### reverse()

```
sb1.reverse();
System.out.println(sb1); // olleH
```

#### length() and capacity()

```
System.out.println(sb1.length()); // Number of characters
System.out.println(sb1.capacity()); // Total buffer size (default is 16 + initial content length)
```



**charAt() and setCharAt()**

```
char ch = sb1.charAt(0);
sb1.setCharAt(0, 'M');
System.out.println(sb1); // Ml...
```

**4. StringBuilder vs String vs StringBuffer**

FEATURE	STRING	STRINGBUILDER	STRINGBUFFER
<b>MUTABILITY</b>	✗ Immutable	Mutable	Mutable
<b>THREAD-SAFE</b>	✗ No	✗ No	Yes
<b>PERFORMANCE</b>	✗ Slower	Fastest	⚠ Slower (sync)
<b>BEST FOR</b>	Constant text	Fast updates (1 thread)	Multithreading

**5 String vs StringBuffer vs StringBuilder**

FEATURE	STRING	STRINGBUFFER	STRINGBUILDER
MUTABILITY	✗ Immutable	Mutable	Mutable
THREAD-SAFE	✗ No	Yes (all methods are synchronized)	✗ No
PERFORMANCE	✗ Slowest (new object per change)	⚠ Slower (due to thread-safety overhead)	Fastest (no sync overhead)
SYNCHRONIZATION	✗ Not applicable	Synchronized	✗ Not synchronized
USE CASE	Constant/fixed string content	Multi-threaded environment	Single-threaded environment
PACKAGE	java.lang	java.lang	java.lang
INTRODUCED IN	JDK 1.0	JDK 1.0	JDK 1.5
METHODS FOR CHANGE	N/A (strings can't be modified)	append(), insert(), delete(), replace()	append(), insert(), delete(), replace()
MEMORY EFFICIENT?	✗ No (creates many objects)	Yes	Yes



## Example Comparison

```
// String (immutable)
String s = "Hello";
s = s + " World"; // Creates a new String object

// StringBuffer (mutable, thread-safe)
StringBuffer sb = new StringBuffer("Hello");
sb.append(" World"); // Modifies original object

// StringBuilder (mutable, not thread-safe)
StringBuilder sb2 = new StringBuilder("Hello");
sb2.append(" World"); // Modifies original object
```

## When to Use What?

SITUATION	RECOMMENDED TYPE
<b>SIMPLE, UNCHANGING TEXT</b>	String
<b>MANY STRING CHANGES IN MULTITHREADED CODE</b>	StringBuffer
<b>MANY STRING CHANGES IN SINGLE-THREADED CODE</b>	StringBuilder

## 6 String Topics for Product-Based Companies

### 1. Core String Handling (Basic - Intermediate)

Subtopic	Concepts & Java APIs
<b>String declaration &amp; immutability</b>	String, memory model, equals() vs ==
<b>StringBuilder / StringBuffer</b>	Mutability, performance
<b>Substring &amp; concatenation</b>	substring(), concat(), +
<b>String comparison</b>	equalsIgnoreCase(), compareTo()
<b>Character operations</b>	charAt(), toCharArray(), Character methods
<b>Trimming, replacing, splitting</b>	trim(), replace(), split()



## 2. String Pattern-Based Problems (High Priority)

Subtopic	Techniques
Palindrome checks	Two-pointer, recursion
Anagram check & grouping	Frequency arrays, HashMap, sorting
Isomorphic strings	Two HashMaps or int[256]
Pangram check	Set or array of 26 chars
Valid parentheses	Stack usage
Longest common prefix	Vertical scan, binary search

## 3. Hashing + Sliding Window (High Frequency)

Subtopic	Techniques
Longest substring without repeating chars	HashSet + sliding window
Minimum window substring	HashMap, sliding window
Count distinct substrings	Rolling hash, Trie
Find all anagrams in a string	Sliding window + frequency map
Substring with concatenation of all words	Advanced window & map

## 4. Advanced Pattern Matching & DP

Subtopic	Techniques
KMP Algorithm	Prefix function array
Rabin-Karp	Rolling hash
Longest Palindromic Substring	Dynamic Programming / Expand around center
Edit Distance	DP on substrings
Regex-style pattern match	Recursion + Memoization
Wildcard Matching	DP with * and ?



## 5. Backtracking & Recursion on Strings

Subtopic	Example Problems
String permutations	Backtracking with visited[]
Generate parentheses	Recursion, backtracking
Restore IP addresses	Recursive split with constraints
Word break (I, II)	DP + recursion + memoization
Letter combinations of phone number	DFS tree traversal

## 6. String + Data Structures

Subtopic	Java Tools
Trie (Prefix Tree)	Custom TrieNode class
Suffix Tree/Array (Optional)	String compression algorithms
Custom sorting of strings	Comparator, lambda in Java
Implement strStr() / indexOf()	Naive, KMP

## Sample Problems for Practice

Problem Type	LeetCode Problems
Substring without repeats	LeetCode 3
Longest Palindromic Substring	LeetCode 5
Group Anagrams	LeetCode 49
Word Break	LeetCode 139
Regex Match	LeetCode 10
Wildcard Match	LeetCode 44
Implement strStr()	LeetCode 28

## Concepts to Emphasize in Java

- Immutability of Strings vs StringBuilder
- Efficient use of HashMap, HashSet, Array for frequency
- Two-pointer and sliding window templates
- Writing clean recursive/backtracking functions
- Java 8 features: Comparator, Streams (for advanced cases)

