STRING BASICS – FOUNDATION

What is a String in Java?

Definition of String

Simple definition (say this first):

A **String** in Java is an **object** that represents a **sequence of characters**.

Technical definition:

In Java, a String is an object of the `java.lang.String` class that stores characters in a sequence and is **immutable** in nature.

Example:

```
String name = "Anu";
```

Here, `"Anu"` is a String object.

Why String is SPECIAL in Java?

Strings are **not ordinary objects** in Java. They are treated specially because:

1. Very Frequently Used

- Usernames
- Passwords
- Emails
- URLs
- API responses
- File paths
- Logs

☞**Note:**

“Without String, Java applications cannot communicate with users or systems.”

2. String Constant Pool (Memory Optimization)

- Java stores String literals in a special memory area called **String Constant Pool**
- This avoids duplicate objects and **saves memory**

Example:

```
String s1 = "Java";
String s2 = "Java";
```

Both `s1` and `s2` point to the **same memory location**.

3. Immutability (Security + Performance)

- Once created, a String **cannot be changed**
- Important for:
  - Security (passwords, URLs)
  - Multithreading
  - Caching

☞ This is why Strings are used in **banking & enterprise applications**.

4. Rich Built-in Methods

Java provides **hundreds of ready-made methods**:

- `length()`
- `substring()`
- `split()`
- `replace()`
- `equals()`

☞ Reduces developer effort drastically.

Why String is NOT a Primitive?

In Java, primitives are:

```
int, float, double, char, boolean, byte, short, long
```

❌ String is **NOT** in this list.

Reasons:

1. String Has Methods

Primitives do not have methods.

```
int a = 10;       // no methods
String s = "Hi"; // has methods like s.length()
```

2. String is a Class

- String belongs to `java.lang.String`
- It supports:
    - Objects
    - Methods
    - Inheritance
    - Interfaces

3. Needs Object Features

Java needs Strings to support:

- Immutability
- Security
- Memory optimization
- Hashing

☞ These are **not possible with primitive data types**.

Where Strings Are Used in Real Life

Username & Password

```
String username = "admin";
String password = "Admin@123";
```

✓ Immutable → secure
✓ Cannot be changed accidentally

API Data (JSON / XML)

```
{
  "name": "Anu",
  "role": "Developer"
}
```

APIs **send and receive Strings**, not primitives.

File Paths

```
String path = "C:/Users/Anu/Documents/file.txt";
```

Logs

```
System.out.println("User logged in successfully");
```

Database & UI Communication

- SQL queries
- Form inputs
- Error messages
- Notifications

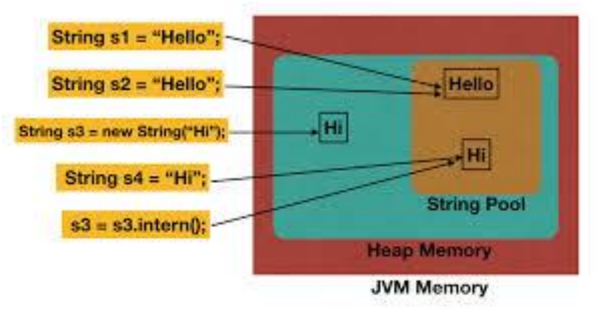
🗣 **“Almost every Java application is 60–70% string handling —
from user input, validation, database queries, APIs, to logging and security.”**

What is Heap Memory and String Pool in Java?

What is Heap Memory?

Simple Definition

Heap memory is the **runtime memory area** where **all Java objects are stored**.



◆ Key Points about Heap Memory

- Created when JVM starts
- Shared by all threads
- Stores:
  - Objects
  - Arrays
  - Class instances
- Garbage Collector works here

Example

```
Student s = new Student();
```

📌 `Student` object is stored in **heap memory**

◆ Real-Time Example

- Employee objects
- Order objects
- API response objects
- Database entity objects

What is String Pool (String Constant Pool)?

✅ Simple Definition

String Pool is a **special area inside heap memory** where **String literals are stored**.

✔ It is NOT separate from heap
✔ It is a **part of heap memory**

◆ Why String Pool Exists?

- Strings are used very frequently
- To avoid duplicate String objects
- To save memory
- To improve performance

◆ Example

```
String s1 = "Hello";
String s2 = "Hello";
```

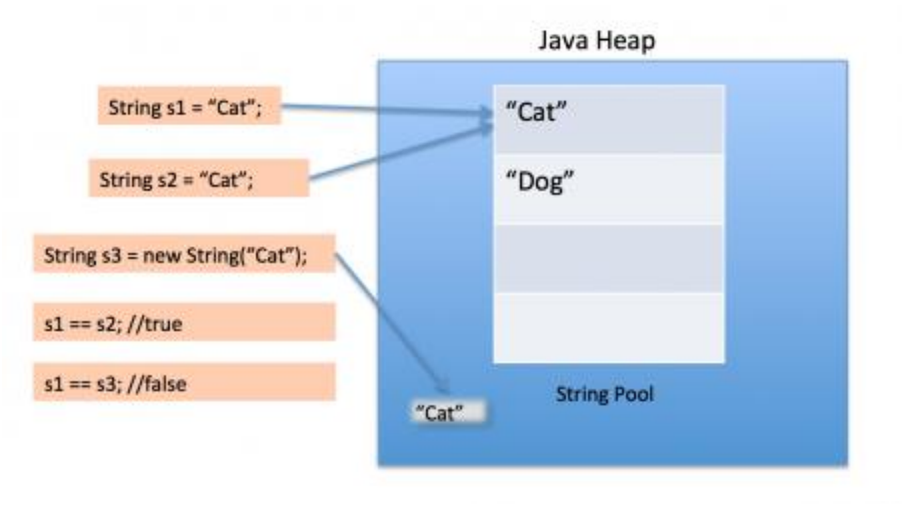
📌 Only **one "Hello" object** is created in **String Pool**

◆ Relationship Between Heap and String Pool

```
Heap Memory
│
├── String Constant Pool
│      └── "Hello"
```

```
│
├── Object Area
│       ├── new String("Hello")
│       ├── Student object
│       └── Employee object
│
```

☞ String Pool is **inside Heap**, not outside.



❖ intern() Method

Used to add heap string into SCP.

```
String interned = str1.intern();
```

✔ Improves memory usage.

Important Difference (Very Common Confusion)

Heap Memory	String Pool
Stores all objects	Stores only String literals
Large memory area	Small optimized area
Objects created using `new`	Strings created using literals
Garbage collected	Garbage collected

String Pool Migration (PermGen → Heap)

❖ Before Java 7

- String Pool stored in **PermGen**
- Limited memory (default ~64MB)

◈ After Java 7

- String Pool moved to **Heap**
- Better memory management
- Avoids `OutOfMemoryError`

Ways of Creating a Java String

In Java, **String objects can be created in multiple ways**, but the **two main ways** are:

Creating String using String Literal (String Constant Pool)

◈ Definition

When a String is created **using double quotes**, Java stores it in a **special memory area called the String Constant Pool (SCP)**.

◈ Example

```
String str = "GeeksforGeeks";
```

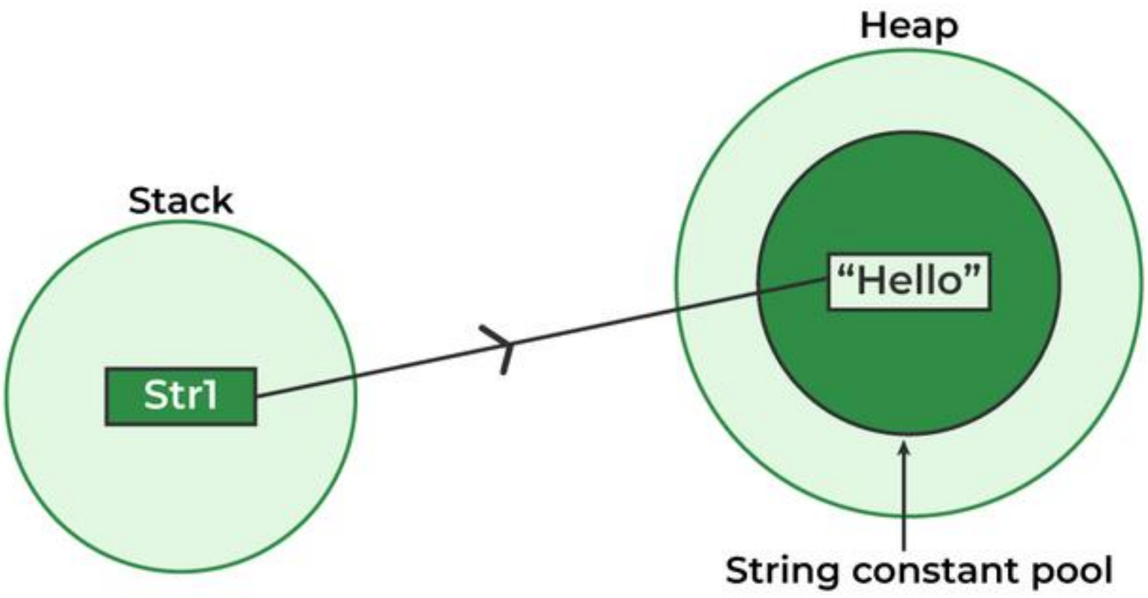
◈ What happens internally?

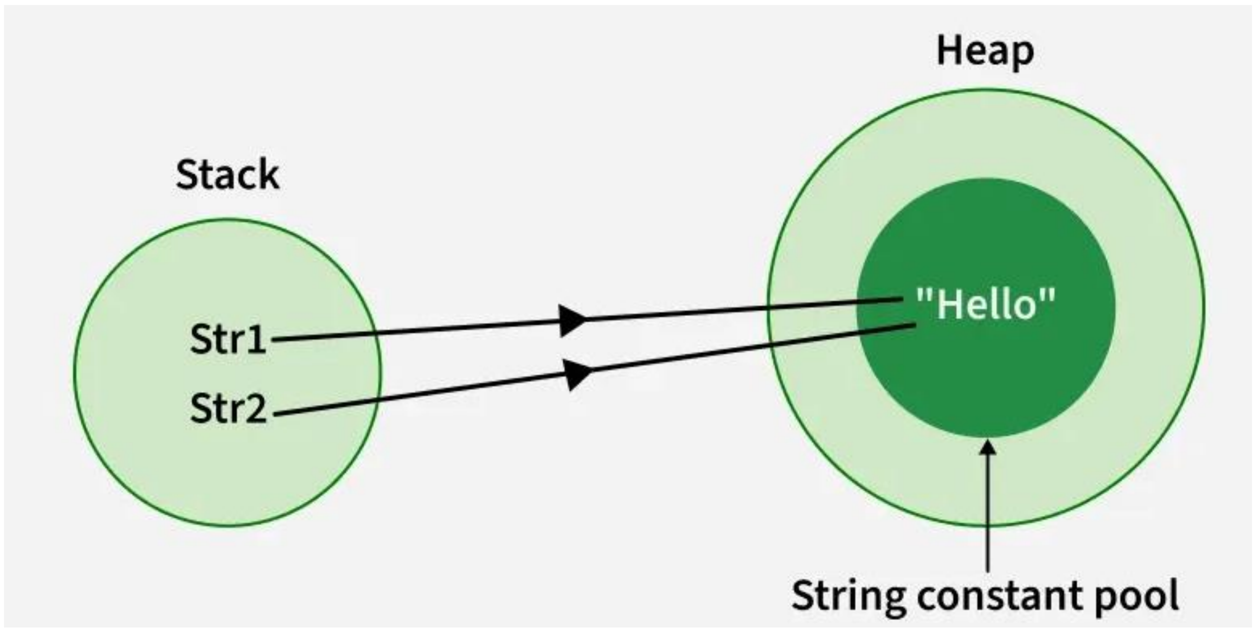
1. JVM checks **String Constant Pool**
2. If `"GeeksforGeeks"` already exists → **reuse it**
3. If not → **create new object in SCP**

✓ No duplicate objects
✓ Memory efficient
✓ Faster performance

◈ Note

"String literals are cached in the String Constant Pool to save memory."


Heap
Stack
"Hello"
Str1
String constant pool


Heap
Stack
"Hello"
Str1
Str2
String constant pool

Creating String using new Keyword (Heap Memory)

◆ Definition

When we use `new`, Java **always creates a new String object in heap memory**, even if the same value exists in SCP.

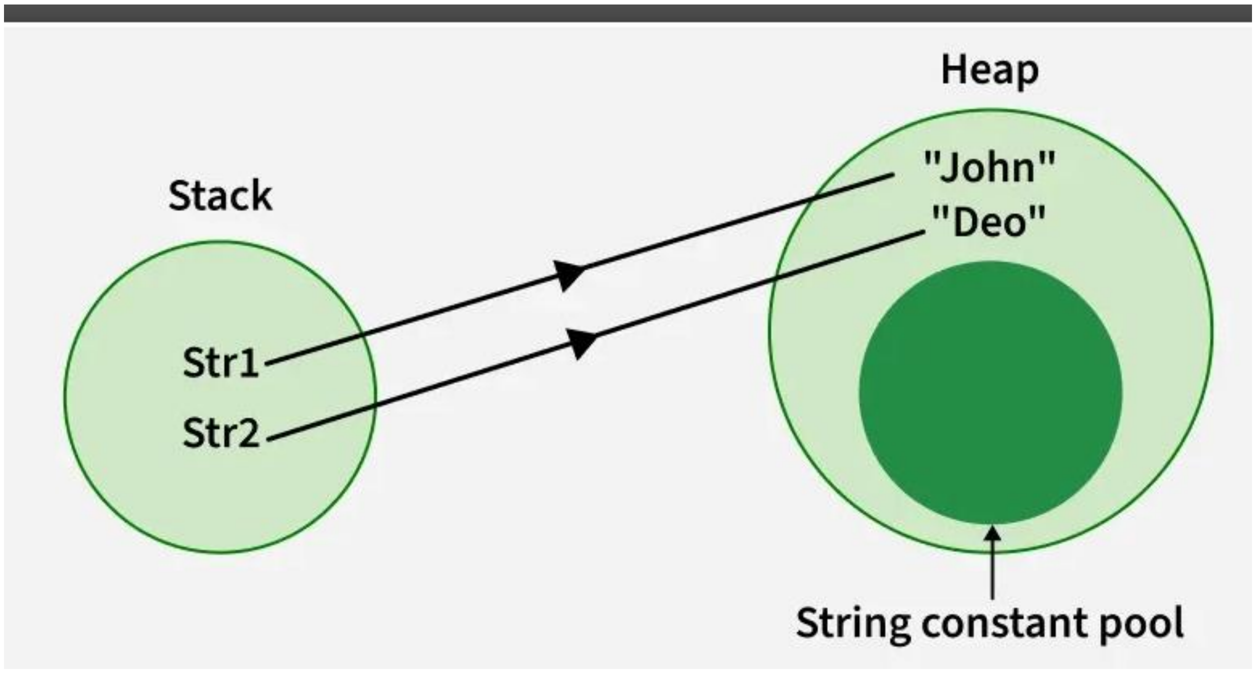
◆ Example

```
String s = new String("Welcome");
```

◆ What happens internally?

1. JVM checks SCP for `"Welcome"`
2. If not found → creates it in SCP
3. JVM creates **another object in heap**
4. Reference `s` points to **heap object**, not SCP

✓ Two objects may exist
✕ Not memory efficient



VISUAL COMPARISON (Explain like this)

String Literal

```
SCP:
"Hello"  ← s1, s2
```

new String()

```
SCP:         HEAP:
"John"       "John" ← s1
"Doe"        "Doe"  ← s2
```

Interview Trick Question

```
String a = "Hello";
String b = new String("Hello");

a == b ?   ❌ false
a.equals(b) ? ✅ true
```

✓ Content same
❌ Memory location different

Visual Representation (Explain in Seminar)

```
String Constant Pool:
"Hello"

Heap:
"Hello" ← s1
"Hello" ← s2
```

Interfaces and Classes in Strings

CharSequence Interface

◆ What is CharSequence?

`CharSequence` is an **interface** that represents a **sequence of characters**.

◆ Methods provided:

- `length()`
- `charAt()`
- `subSequence()`
- `toString()`

Classes that implement CharSequence

1. String
2. StringBuffer
3. StringBuilder

String Class

◈ Definition

- String is an **immutable class**
- Once created, its value **cannot be changed**

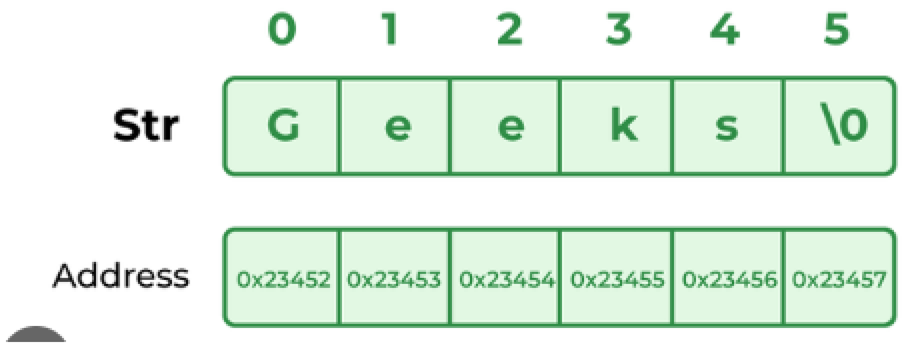
◈ Example

```
String str = "geeks";
```

or

```
String str = new String("geeks");
```

✓ Any modification creates **new object**



For a String, why do we say it has one address?”

Short Answer (Seminar-ready)

A **String has one reference address**, but **internally it contains multiple character addresses**.

Both are correct — they refer to **different levels**.

What is the one address we talk about?

In Java:

```
String str = "Geeks";
```

- `str` is a **reference variable**
- It stores **ONE memory address**
- That address points to the **String object**

☞ This is why we say:

“A String has one address.”

Then what are those multiple addresses in the diagram?

Your diagram shows something like:

```
Index:    0   1   2   3   4
Chars:    G   e   e   k   s
Address:  A1  A2  A3  A4  A5
```

What this actually means:

- Internally, a String stores characters in a **character array**
- Each character occupies a **separate memory location**
- These are **internal addresses**, not the String's reference

---

Think in TWO LEVELS 👇

Level 1: String Reference (ONE address)

```
str  ──▶  String Object
```

- `str` holds **one address**
- That address points to the String object in heap / SCP

Level 2: Internal Character Storage (MANY addresses)

```
String Object
   └── char[] → ['G','e','e','k','s']
                  ↑   ↑   ↑   ↑   ↑
                A1  A2  A3  A4  A5
```

- Characters are stored **internally**
- Each character has its **own memory slot**
- These addresses are **not visible to Java developers**

Why We Say "One Address" in Interviews/Seminar

Because:

- Java variables **never point to individual characters**
- They point to the **object as a whole**
- Internal structure is **abstracted**

☞ So we always talk about **object reference**, not internal array addresses.

🗣 “A String variable holds a single reference address, but internally the String object stores characters in multiple memory locations.”

Why You Cannot Change a Character Directly?

```
str.charAt(0) = 'M';   // ❌ Not allowed
```

Because:

- Characters are stored internally
- String is **immutable**
- No direct access to internal array

StringBuffer

◈ Definition

- Mutable
- Thread-safe
- Used in **multi-threaded environments**

◈ Example

```
StringBuffer sb = new StringBuffer("GeeksforGeeks");
sb.append(" Java");
```

✓ Same object modified
❌ Slower due to synchronization

StringBuilder

◈ Definition

- Mutable
- NOT thread-safe

- Faster than StringBuffer
- Used in **single-threaded applications**

◈ Example

```
StringBuilder sb = new StringBuilder();
sb.append("GFG");
```

✓ High performance
✓ No synchronization overhead

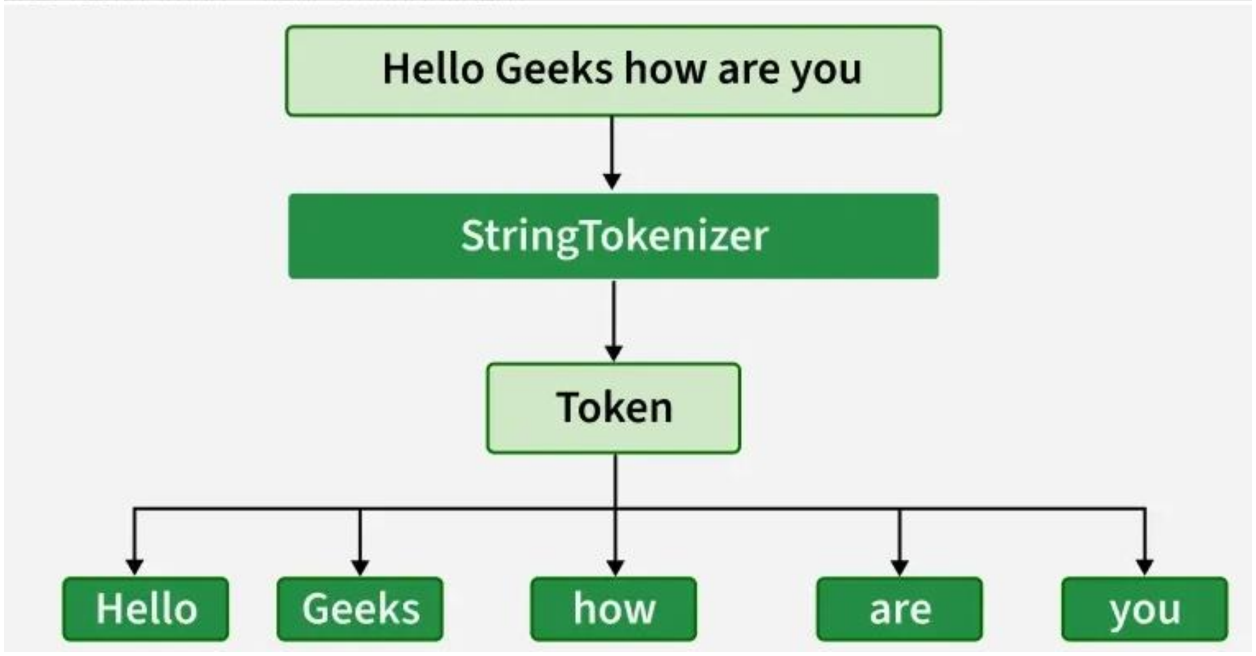
StringTokenizer

◈ Definition

Used to **split a string into tokens** based on delimiters.

◈ Example

```
StringTokenizer st = new StringTokenizer("Java String Example");
```



✓ Maintains current position
✓ Returns tokens one by one

Immutable String in Java

❖ Definition

In Java, **String objects are immutable**, meaning once created, their value **cannot be changed**.

❖ Example

```
String s = "Sachin";
s.concat(" Tendulkar");
System.out.println(s);
```

❖ Output

```
Sachin
```

❖ Explanation

- `"Sachin"` remains unchanged
- `"Sachin Tendulkar"` is created as a **new object**
- Reference `s` still points to `"Sachin"`

❖ Explicit Assignment Example

```
String name = "Sachin";
name = name.concat(" Tendulkar");
System.out.println(name);
```

❖ Output

```
Sachin Tendulkar
```

✔ Reference updated to new object

Why String is Immutable? (MOST IMPORTANT PART)

1. Security Reason

Strings are used in:

- Passwords
- URLs
- File paths
- Class loading

☞ If String were mutable:

- A hacker could change values **after validation**
- Serious security risk

Note:

"Immutability protects sensitive data like passwords and URLs."

---

2. Performance (Caching)

- Java uses **String Constant Pool (SCP)**
- Same literal is reused

```
String a = "Hello";
String b = "Hello";
```

✓ One object reused
✓ Faster access

☞ If String were mutable, caching would be **unsafe**.

---

3. Memory Optimization

- Thousands of same values → **one object**
- Saves heap memory
- Less garbage creation

---

4. Thread Safety

- Immutable objects are **inherently thread-safe**
- Multiple threads can share same String

- No synchronization required

Seminar line:

“Because Strings are immutable, they are automatically thread-safe.”

---

5. HashCode Consistency

- Strings are commonly used as **keys in HashMap**
- HashCode is **cached**
- If value changed → HashMap would break

☞ Immutability guarantees **stable hashCode**.

What Happens Internally When We Modify a String?

Example

```
String s = "Sachin";
s = s.concat(" Tendulkar");
```

🔍 Internal Steps

1. `"Sachin"` is created (SCP)
2. `concat()` is called
3. JVM creates **new object** → `"Sachin Tendulkar"`
4. Reference `s` is updated to new object
5. Old `"Sachin"` remains unchanged
