String \_Builder\_Equals \_Method

Eg: String\_Builder\_Equals\_Method\_Eg5

## 🔑 Key Concept (VERY IMPORTANT)

### Why equals() does NOT compare content for StringBuilder

* String class **overrides** equals() → compares content
* StringBuilder **does NOT override** equals()
* So StringBuilder.equals() = Object.equals()
* And Object.equals() compares **references only**

📌 That’s why:

s1 == s2 → false

s1.equals(s2) → false

## 🧠 Memory Map (simple)

STACK HEAP

------------------------------------------

s1 ───────────────▶ StringBuilder("Hello")

s2 ───────────────▶ StringBuilder("Hello")

* Two different objects
* Same content, different memory locations

## 📊 Comparison Table

| **Type** | **==** | **equals()** | **Reason** |
| --- | --- | --- | --- |
| String | reference | content | equals overridden |
| StringBuilder | reference | reference | equals NOT overridden |

## ✅ Final Interview Line (remember this 🔥)

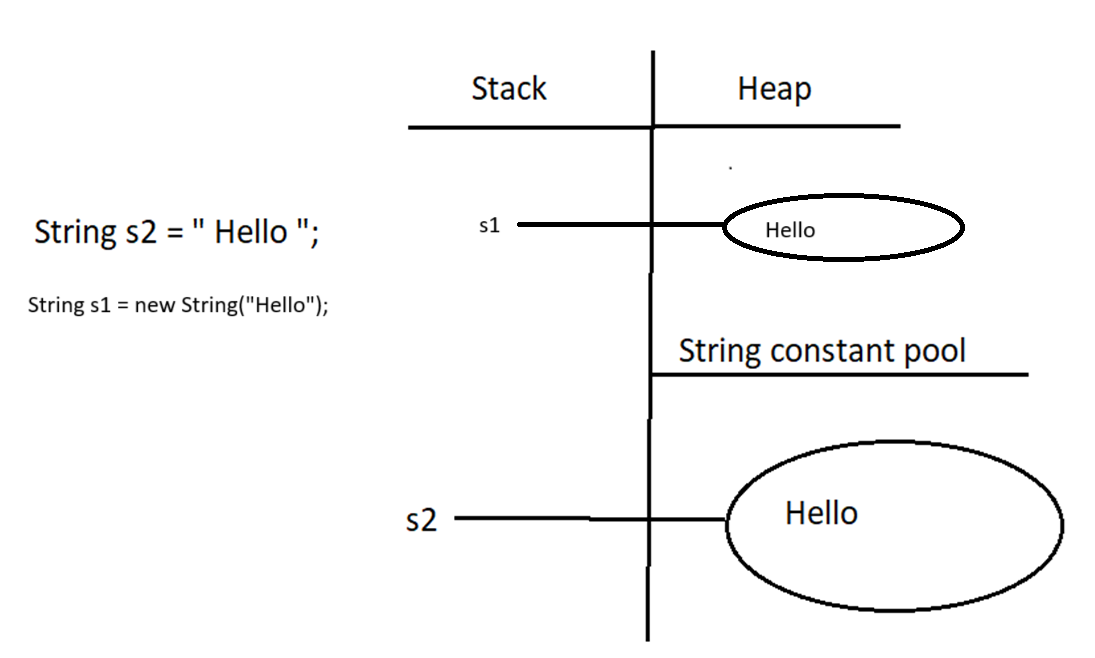
**StringBuilder does not override equals() — so it compares references, not content.**

## Eg: String\_Eg5

## Code

String s1 = new String("Hello");

String s2 = "Hello";



## Step-by-step explanation

### 🔹 Line 1

String s1 = new String

continue

Continuing the explanation 👇  
I’ll keep it **simple, exact, and memory-map based**.

## Code

String s1 = new String("Hello");

String s2 = "Hello";

## 🔹 Line 1 explanation

String s1 = new String("Hello");

What happens internally:

1. "Hello" is a **String literal**
   * Stored in **String Constant Pool (SCP)**
   * If already present, JVM reuses it
2. new String("Hello")
   * Creates a **NEW String object in HEAP**
   * Copies the content "Hello" from SCP into heap object
3. s1
   * Reference variable (stack)
   * Points to the **heap object**

## 🔹 Line 2 explanation

String s2 = "Hello";

What happens:

1. JVM checks SCP
2. "Hello" already exists in SCP
3. No new object is created
4. s2 points directly to **SCP object**

## 🧠 Memory Map

STACK HEAP SCP

------------------------------------------------------------

s1 ───────────────▶ "Hello" (heap obj) "Hello"

s2 ─────────────────────────────────────────▶ "Hello"

## 🔹 Comparison results

s1 == s2 ❌ false

s1.equals(s2) ✅ true

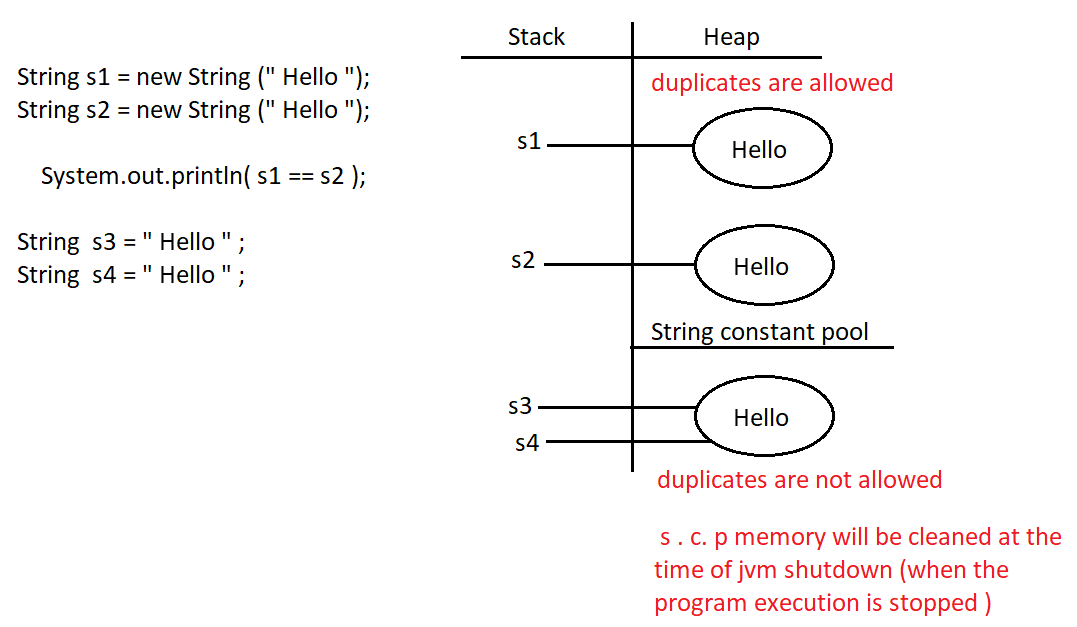
### Why?

* == compares **references**
  + heap vs SCP → different memory locations
* equals() compares **content**
  + both contain "Hello"

## 🔑 One-line interview answer

new String("Hello") creates a heap object,  
"Hello" without new refers to SCP.

Eg: Strings\_Eg6



## 🔑 Key concepts shown by this example

### 1️⃣ new String("Hello")

* Always creates a **new object in heap**
* Even if same content already exists

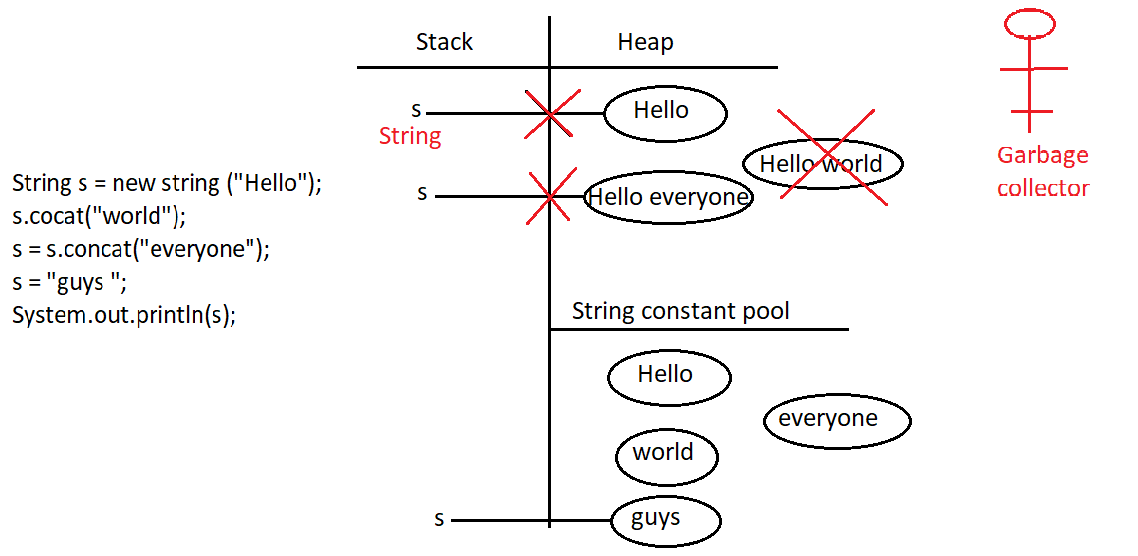
### 2️⃣ "Hello" (String literal)

* Stored in **String Constant Pool (SCP)**
* JVM reuses the same object

### 3️⃣ == operator

* Compares **memory references**
* Does NOT compare content

Eg: Strings\_Eg7



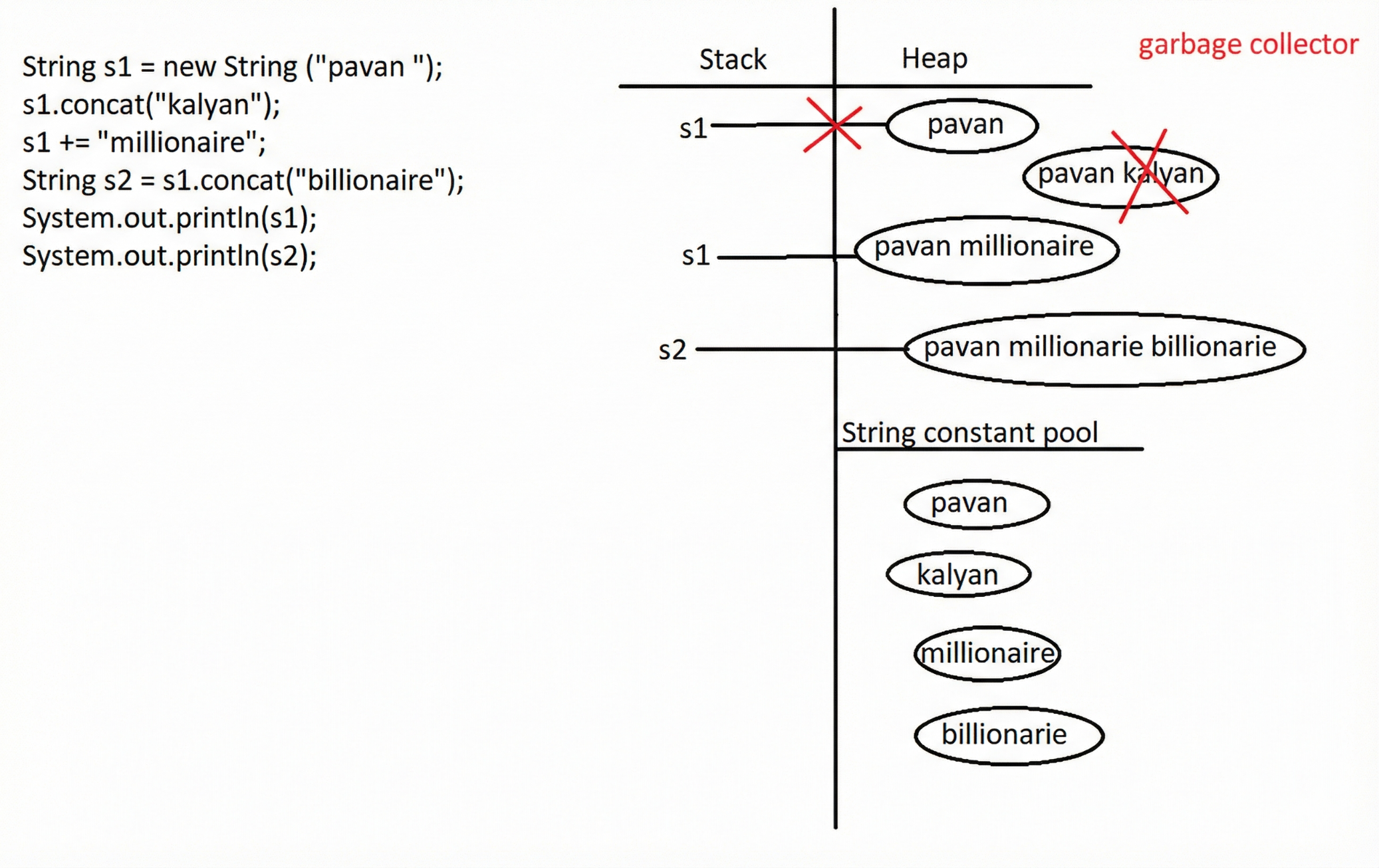
## 🔑 Key rules demonstrated

1️⃣ **String is immutable**  
→ concat() never modifies existing object

2️⃣ **Unreferenced objects**  
→ eligible for Garbage Collection

3️⃣ **Reference reassignment**  
→ same variable can point to different objects

Eg: String\_Eg8



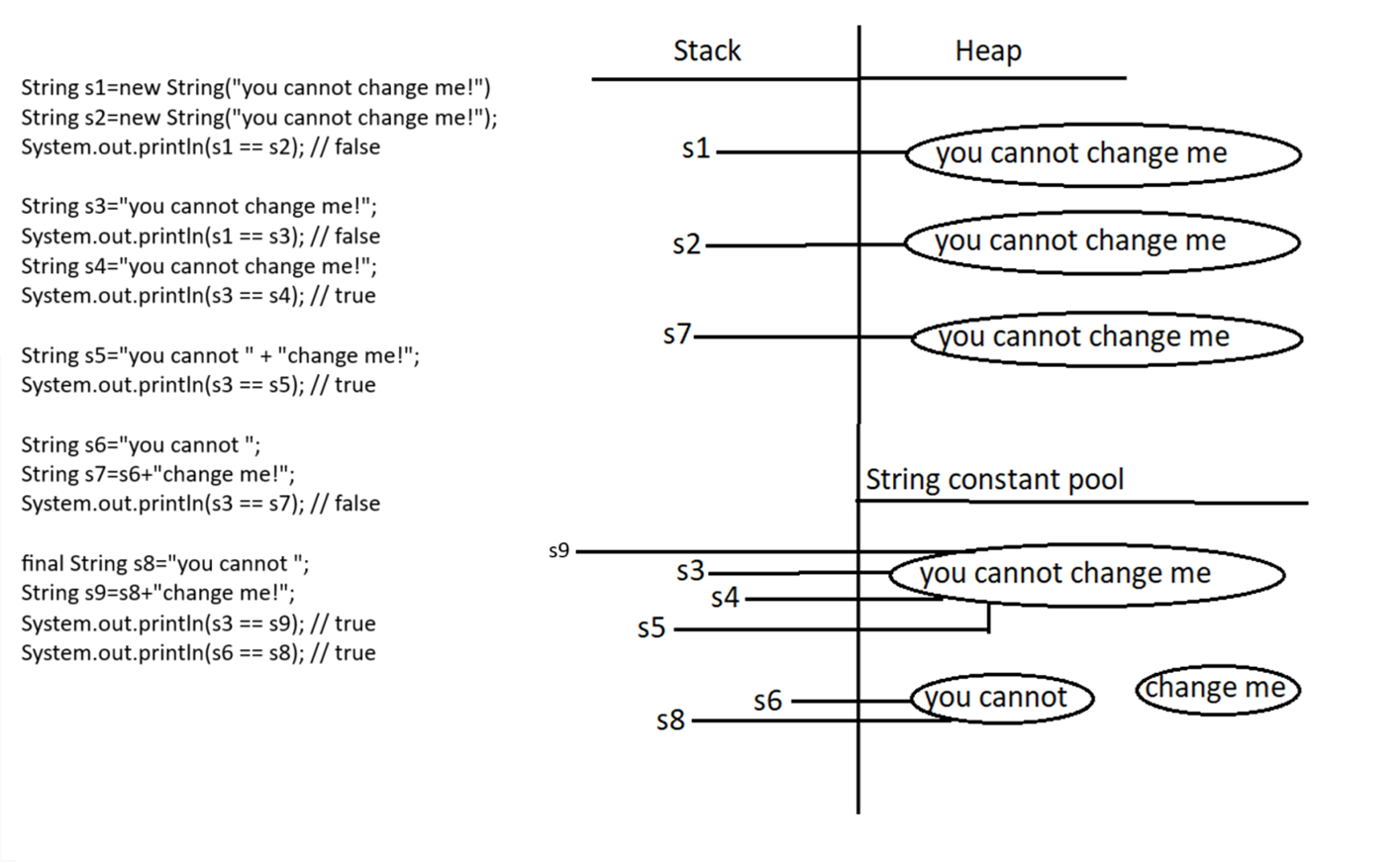
## 🔑 Key rules demonstrated

1️⃣ **String is immutable**  
→ every change creates a new object

2️⃣ **concat() without assignment**  
→ object is created but lost (GC eligible)

3️⃣ **+= creates new String object**  
→ reference is reassigned

Eg: String\_Eg9



# 🟢 STEP 1 — s1 and s2 (new String())

String s1 = new String("you cannot change me");

String s2 = new String("you cannot change me");

### What happens

* Literal "you cannot change me" → **SCP**
* new String() → **ALWAYS creates a new HEAP object**
* Happens **twice**

### Memory

STACK HEAP SCP

---------------------------------------------------------------------------

s1 ───────────────▶ "you cannot change me" (heap obj #1)

s2 ───────────────▶ "you cannot change me" (heap obj #2)

"you cannot change me"

s1 == s2 → false

# 🟢 STEP 2 — s3 literal

String s3 = "you cannot change me";

### What happens

* JVM finds literal already in SCP
* No new object created
* s3 points to SCP

s1 == s3 → false (heap vs SCP)

# 🟢 STEP 3 — s4 literal reuse

String s4 = "you cannot change me";

### What happens

* SCP object reused
* s3 and s4 point to SAME object

s3 == s4 → true

# 🟢 STEP 4 — Compile-time concatenation (s5)

String s5 = "you cannot" + " change me";

### What happens

* Both operands are **string literals**
* Compiler joins them **at compile time**
* Result placed in **SCP**

Internally becomes:

String s5 = "you cannot change me";

s3 == s5 → true

# 🟢 STEP 5 — Runtime concatenation (s7)

String s6 = "you cannot";

String s7 = s6 + " change me";

### What happens

* s6 is **NOT final**
* Concatenation happens **at runtime**
* JVM uses StringBuilder
* Result created in **HEAP**

Internally:

new StringBuilder()

.append(s6)

.append(" change me")

.toString();

s3 == s7 → false (SCP vs HEAP)

# 🟢 STEP 6 — final enables compile-time optimization (s9)

final String s8 = "you cannot";

String s9 = s8 + " change me";

### What happens

* s8 is final
* Value known at **compile time**
* Compiler folds expression
* Result stored in **SCP**

Internally becomes:

String s9 = "you cannot change me";

s3 == s9 → true

# 🟢 STEP 7 — Same literal comparison

System.out.println(s6 == s8);

* Both point to same SCP literal "you cannot"

true

# ✅ FINAL COMPLETE MEMORY MAP (ALL TOGETHER)

STACK HEAP SCP

---------------------------------------------------------------------------

s1 ───────────────▶ "you cannot change me" (heap #1)

s2 ───────────────▶ "you cannot change me" (heap #2)

s7 ───────────────▶ "you cannot change me" (heap #3)

s3 ───────────────────────────────────────────────▶ "you cannot change me"

s4 ───────────────────────────────────────────────▶ "you cannot change me"

s5 ───────────────────────────────────────────────▶ "you cannot change me"

s9 ───────────────────────────────────────────────▶ "you cannot change me"

s6 ───────────────────────────────────────────────▶ "you cannot"

s8 ───────────────────────────────────────────────▶ "you cannot"

SCP contains ONLY:

- "you cannot change me"

- "you cannot"

- " change me"

# 🔥 FINAL CONFIRMATION (your understanding)

✅ Heap objects → **s1, s2, s7 ONLY**  
✅ SCP objects → literals ( “hello”) + compile-time constants

## 1️⃣ It **must be** final

### ✅ Compile-time constant

final int a = 10;

### ❌ Not a compile-time constant

int b = 10; // not final

## 2️⃣ It must be a **primitive type or String**

### ✅ Compile-time constants

final int x = 5;

final double d = 2.5;

final String s = "Hello";

### ❌ Not compile-time constants

final Integer i = 10; // wrapper class

final StringBuilder sb = new StringBuilder("Hello");

## 3️⃣ It must be **initialized at declaration**

### ✅ Compile-time constant

final int p = 100;

### ❌ Not compile-time constant

final int q;

q = 100; // initialized later

## 4️⃣ The value must be a **constant expression**

### ✅ Constant expressions

final int a = 10 + 20;

final String s = "Java" + " World";

### ❌ Not constant expressions

final int b = getValue(); // method call

final String s2 = new String("Hi"); // runtime object

## 🔥 All rules satisfied → Compile-time constant

final String msg = "Hello" + " Java";

✔ final  
✔ String  
✔ initialized at declaration  
✔ constant expression

## ❌ Looks final but NOT a compile-time constant

final String name = getName();

❌ method call → runtime value

## 🧠 One-line summary

**Compile-time constant = final + primitive/String + initialized immediately + constant expression**

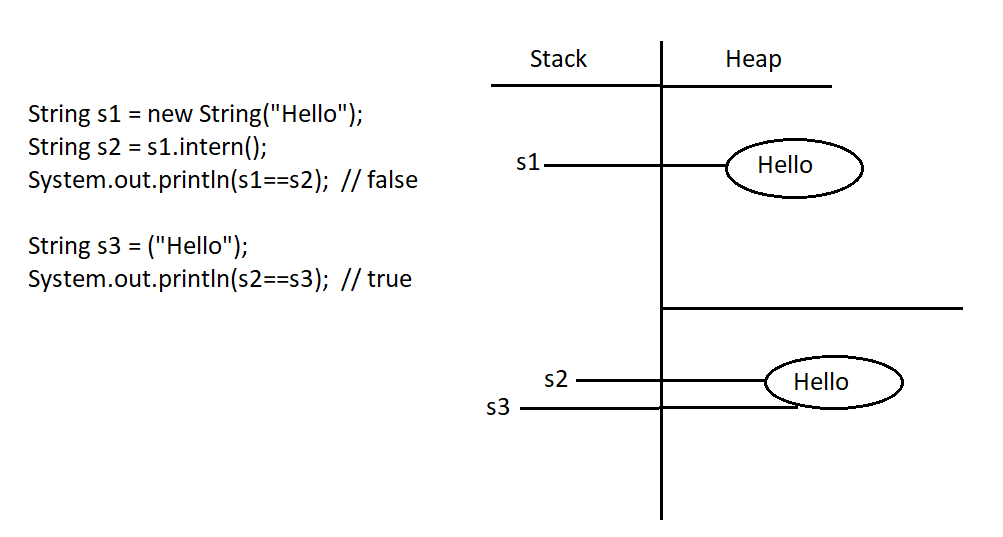
✅ + does **not** always mean runtime  
✅ final enables compile-time folding  
✅ Same text ≠ same memory

## 🧠 ONE-LINE MASTER RULE (INTERVIEW GOLD)

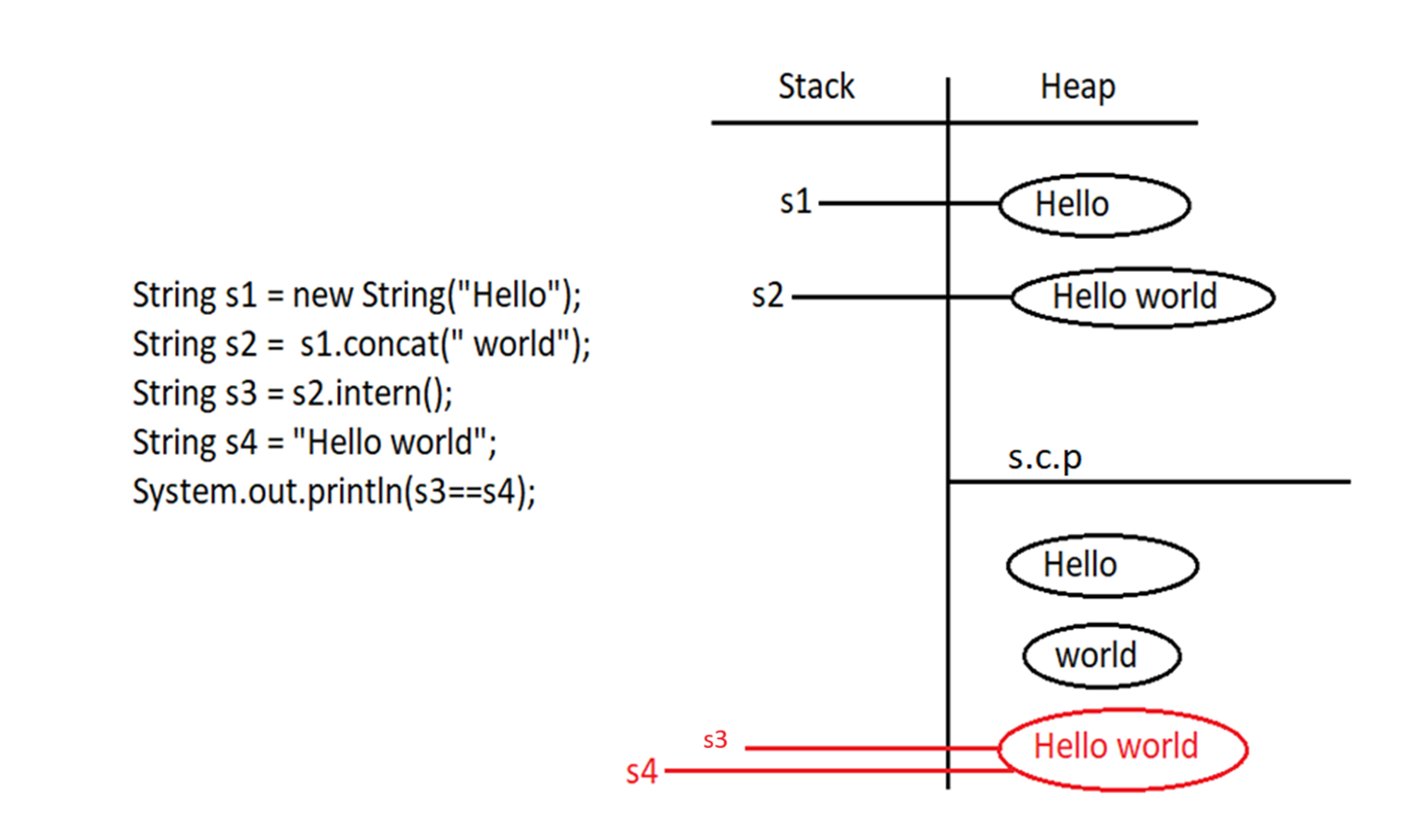
**Creation decides memory, not content.  
Runtime → heap, compile-time → SCP.**

Eg: String\_Eg10

intern() checks the String Constant Pool (SCP).  
If an equal string already exists there, it returns the reference to that SCP string.  
If it does NOT exist, it adds the string to SCP and returns that reference.



Eg: Strings\_Eg11



## Note: Main Point (Very Important)

**In the String Constant Pool (SCP), even a space character counts.**

"Hello world" and "Helloworld" are **two completely different strings**,  
so they are stored as **two different objects in SCP**.

## 📌 Importance (Need) of String Constant Pool (SCP)

### 1️⃣ Memory optimization and performance improvement

In real-world applications, the **same String values** (such as city names, country names, status values, etc.) are used repeatedly.  
If a new String object were created every time with the same content, it would:

* Waste memory
* Reduce performance due to excessive object creation

The **String Constant Pool (SCP)** solves this by storing **only one copy of each unique String literal** and reusing it wherever required.

### 2️⃣ Reusability of String objects

In SCP:

* A **single String object** can have **multiple references**
* This reuse significantly **reduces memory usage**
* It also **improves application performance**

This is the **primary purpose** of the String Constant Pool.

### 3️⃣ Immutability makes SCP safe (very important)

Since multiple references point to the **same String object**, allowing modification would be dangerous.  
If Strings were mutable:

* A change made using one reference would affect **all other references**
* This could corrupt shared data across the application

To prevent this, **String objects are immutable**.

✔ **Immutability is NOT a disadvantage**  
✔ **Immutability is the solution that makes SCP possible and safe**

### 4️⃣ What happens when a String value “changes”?

When a program appears to change a String value (for example, changing a city name):

* The original String object is **never modified**
* A **new String object** is created instead
* If an identical String already exists, it is **reused**
* Existing references remain **completely unaffected**

This guarantees **data safety and consistency**.

## 🧠 Simple real-life understanding

* Many users can share the same String value (example: "Bengaluru")
* Only **one object** is stored in memory
* When someone needs a different value (example: "Hyderabad"), a **new object** is used
* Shared values are never altered

## 🔥 One-line interview answer

**The String Constant Pool exists to reuse immutable String objects safely, improving memory utilization and performance.**

