String Immutability

@Divya

In Java, a String is immutable, which means once a String object is created, its value cannot be changed.

Swipe for more

String Immutability

When we perform operations like concatenation or replacement, a new String object is created in memory. The original String remains unchanged, and the reference variable points to the new object if reassigned.

Swipe for more

Why is String immutable?

@Divya

Java

Thread-safe: Since the value cannot change, it is safe to use in multi-threaded environments.

Secure: Strings are used for sensitive data, such as passwords and URLs. Immutability prevents accidental or malicious modification.

Memory-efficient: Java utilizes a String Constant Pool, enabling multiple references to share the same String value and conserve memory.

Swipe for more

Compile-Time vs Runtime

String Concatenation

Compile-Time Concatenation:-

Compile time is when Java converts your code into bytecode. At this time, the compiler only knows fixed values(such as String literals or final constants)

For example:

String str5 = "Hi" + "Java";

Here:

1) Both values are String literals

2) The compiler already knows the final value

So it directly creates: **String str5 = "HelloJava";**

3) Stored in String Constant Pool

Swipe for more

Compile-Time vs Runtime

String Concatenation

Runtime Concatenation

When at least one variable is involved, concatenation happens at runtime.

<u>For example:</u>

```
str3 = str3.concat("Java");
```

Here:

1) str3 is a variable (not a string literal or final). Its value can change during program execution (for example, from "Hi" to "Hello" at runtime).

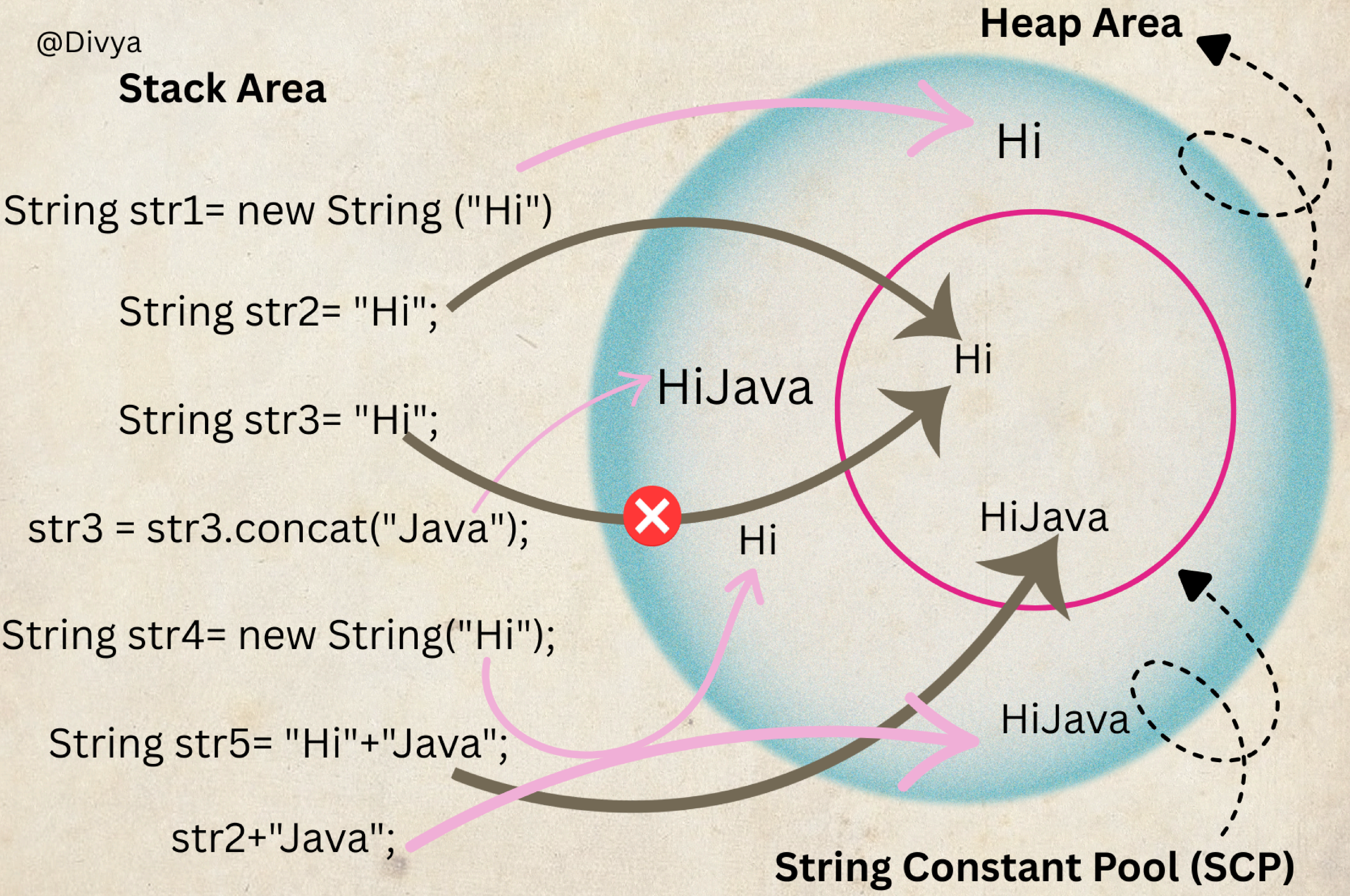
2) The compiler cannot predict the value of str3, so it must wait until runtime to perform the concatenation.

3) As a result, a new String object is created and stored in the Heap as: **str3 = "HiJava"**

4) This is slower compared to compile-time concatenation.

Swipe for more

@Divya



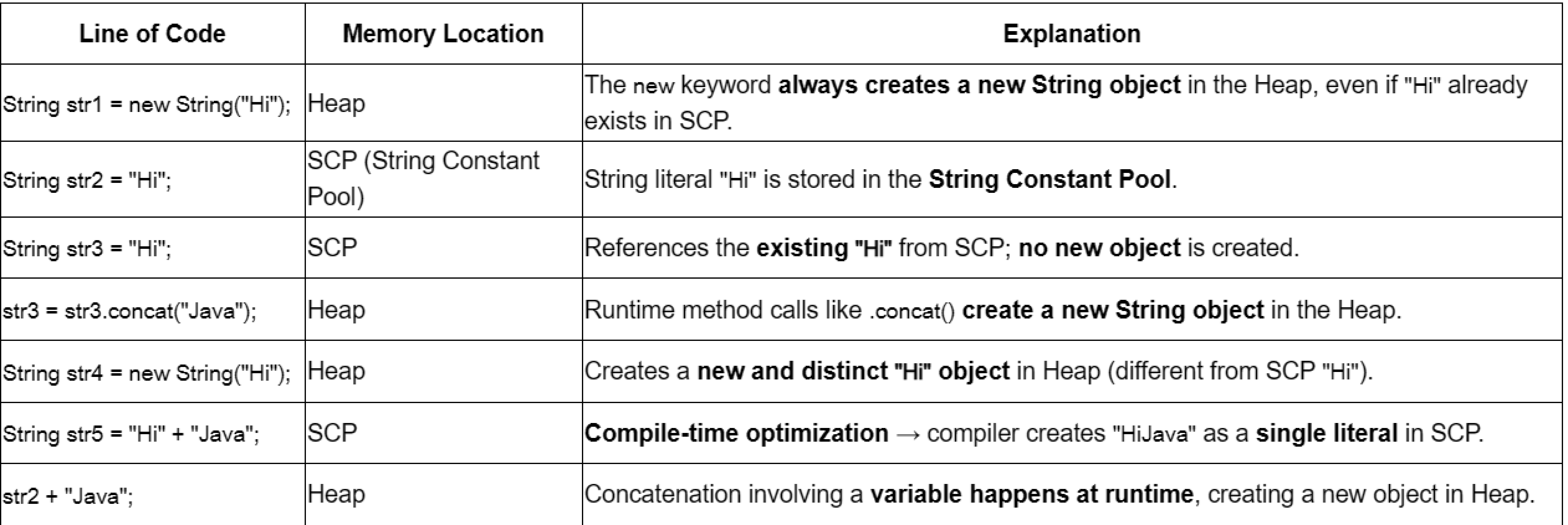
Stack Area

Heap Area

```
String str1= new String ("Hi")

String str2= "Hi";

String str3= "Hi";

str3 = str3.concat("Java");

String str4= new String("Hi");

String str5= "Hi"+"Java";

str2+"Java";
```

String Constant Pool (SCP)

Note : ❌ str3 no longer points to 'Hi' after concatenation'



Line of Code	Memory Location	Explanation
String str1 = new String("Hi");	Heap	The new keyword **always creates a new String object** in the Heap, even if "Hi" already exists in SCP.
String str2 = "Hi";	SCP (String Constant Pool)	String literal "Hi" is stored in the **String Constant Pool**.
String str3 = "Hi";	SCP	References the **existing "Hi"** from SCP; **no new object** is created.
str3 = str3.concat("Java");	Heap	Runtime method calls like .concat() **create a new String object** in the Heap.
String str4 = new String("Hi");	Heap	Creates a **new and distinct "Hi" object** in Heap (different from SCP "Hi").
String str5 = "Hi" + "Java";	SCP	**Compile-time optimization** → compiler creates "HiJava" as a **single literal** in SCP.
str2 + "Java";	Heap	Concatenation involving a **variable happens at runtime**, creating a new object in Heap.

@Divya

Garbage Collection & Concatenation Rules

1) The garbage collector will **remove objects from heap memory** if they are no longer referenced.

2) However, it will **not remove string literals in the String Constant Pool**, even if they are unused.

3) If the **compiler is 100% sure** about the value, concatenation happens at compile time.

4) If there is any **uncertainty**, concatenation happens **at runtime**. for this one