MASTERING STRING
IN JAVA

Take a closer look at Java's String class —
how it works, why it's immutable, and the
methods every dev should know.

Swipe for more

In Java, a String is a sequence of characters use to represent a text. It is one of the mostly use classes in Java and it is the part of java.lang package.

Unlike many other languages where strings are just arrays of characters, Java treats strings as objects of the String class. This means they come with a rich set of built-in methods that make text manipulation easy and powerful.

One of the key characteristics of a Java String is that it's immutable — once created, its value cannot be changed. Any modification like concatenation or replacement actually creates a new String object under the hood.

Immutability of Java Strings

One of the most important concepts to understand about String in Java is that it is immutable. The values of String cannot be changed or modified once the String object is created.

🔍 What Does Immutability Mean?

When you modify a string — for example, by concatenating or replacing characters — you're not actually changing the original object. Instead, Java creates a new String object in memory with the updated value.

```java
String name = "Kiran";
name = name + " Khanal";

System.out.println(name); // Output: Kiran Khanal
```

In the above example,
- Kiran is the original String.
- Kiran Khanal is the new String object.
- The original String Kiran remains unchanged in the memory.

What's happening behind the scene?

At first, the variable name points to the string "Kiran". When you concatenate name + " Khanal", Java doesn't change the original "Kiran" string. Instead, it creates a brand new string: "Kiran Khanal". Now, the variable name points to this new string. The original "Kiran" string is still in memory — it just isn't being used anymore (unless something else refers to it). Java garbage collector manages the unused objects.

🧪 Think of It Like This:

Imagine writing "Kiran" on a piece of paper. Instead of erasing and adding " Khanal" to the same paper, Java writes "Kiran Khanal" on a new piece of paper and gives you that instead.
The first one still exists — but you're now working with the new one.

💡 Why Are Strings Immutable?

- Security: Strings are often used in sensitive operations — like file paths, network connections, or credentials. Making them immutable ensures they can't be changed unexpectedly.

- Thread Safety: Since String objects can't be changed, they are naturally safe to use in multi-threaded environments.

- Performance (String Pooling): Java maintains a String pool to save memory. Immutable strings can be shared safely, which reduces object creation and memory usage.

🔲 What is the Java String Pool?

In Java, all objects — including strings — are stored in a region of memory called the Heap. But strings are a bit special.

To optimize memory usage, Java uses a specific area within the heap known as the String Pool (or String Constant Pool).
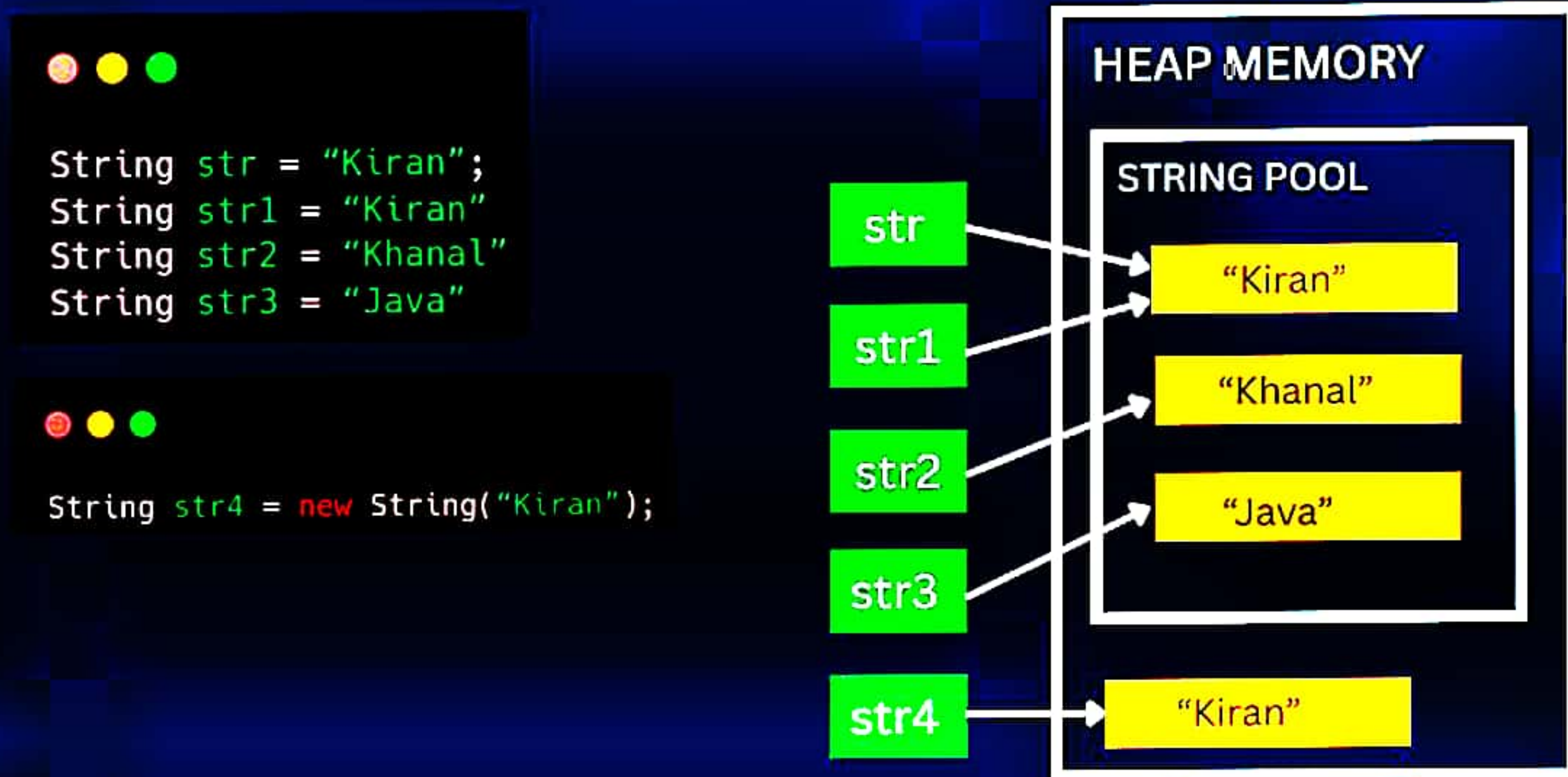
🧠 How It Works

Whenever a string is created using double quotes,

```
String lang = "Java";
```

Java checks the String Pool to see if a string with the same content already exists.
- ✅ If it exists, it reuses the same object from the pool.
- ❌ If it doesn't, it creates a new object and stores it in the pool.

📝 Pictorial Representation of String Pool

```
String str = "Kiran";
String str1 = "Kiran"
String str2 = "Khanal"
String str3 = "Java"
```

```
String str4 = new String("Kiran");
```

HEAP MEMORY

STRING POOL

str	"Kiran"
str1	"Khanal"
str2	"Java"
str3	
str4	"Kiran"

🔬 What's Really Happening?

🌕 str and str1 point to the same object in the String Pool Because both were created using string literals, Java reuses the existing value.

But,

🚨 When you use the new keyword:
📝 Java skips the pool and creates a new object in the heap
— even if "Kiran" already exists in the pool.
📍 So now, str4 is a completely separate object in memory.

🔗 What is intern() in Java?

The intern() method is used to manually add a String to the String Pool, or return the existing pooled version if it already exists.

It's like telling Java:

> "Hey, I want this string to behave like a literal. Please store it in the pool if it's not there."

🩹 Example

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

System.out.println(a == b); // ❌ false

String c = a.intern();

System.out.println(c == b); // ✅ true
```

If you create a string using the new keyword, it won't automatically go into the String Pool — it gets stored in the heap. But when you call .intern() on that string, Java checks the pool for an existing match. If found, it returns the pooled object; otherwise, it adds the string to the pool.

🔗 String Comparison in Java

Comparing strings in Java might seem simple, but there's a key difference between checking if two strings have the same content and if they are the exact same object in memory.

✅ Using .equals()

If you want to compare the content of two strings — whether they hold the same sequence of characters — use the .equals() method.

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

System.out.println(a.equals(b)); // true ✅
```

Even if the two strings are different objects, .equals() checks what's inside them — the actual characters.

⚠️ Using ==

The == operator checks if both variables point to the exact same object in memory (i.e., same reference), not just the content.

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

System.out.println(a == b); // false ❌
System.out.println(a.equals(b)); // true ✅
```

Here, a and b have the same content, but == returns false because a was created using new, so it's a different object in the heap.

💬 == Might Return true Sometimes

If both strings are created as literals, Java uses the String Pool, and both variables might point to the same object:

```java
String a = "Java";
String b = "Java";

System.out.println(a == b); // true ✅
```

But don't rely on this — always use .equals() for comparing string values.

🟦 .equalsIgnoreCase()

If you want to compare two strings without caring about case, use .equalsIgnoreCase():

```java
String name1 = "Kiran";
String name2 = "kiran";

System.out.println(name1.equalsIgnoreCase(name2)); // true ✅
```