**String in Java**

* Sequence of Characters,
* String is a built-in class in java.lang package.
* String objects are immutable objects. It means once the object is created then the content or data of the object can't be modified.
* When you try to modify the contents of object then new object will be created as a result.
* The reason for making strings immutable in Java is to increase efficiency, security, and thread-safety.

**Difference between String s = "hello"; and String s = new String();**

In Java, both String s = "hello"; and String s = new String(); are used to declare a String, but they differ in how they allocate memory and how they work internally.

**1. String s = "hello"; (String Literal)**

* "hello" is stored in the **String Pool** (a special memory area inside the heap).
* If another String s2 = "hello"; is declared, it will **reuse** the same object from the String Pool rather than creating a new one.
* It is more memory-efficient because Java manages the String Pool to avoid duplicate objects.

**2. String s = new String(); (Using new Keyword)**

* This explicitly creates a new String object in the **heap memory**.
* Even if the string is already present in the String Pool, Java will **not** reuse it and instead creates a new instance.
* If you assign a value like String s = new String("hello");, it will:
  1. First, check if "hello" is in the String Pool (if not, add it).
  2. Then, create a **separate object in the heap** for "hello".

**Key Differences:**

| **Feature** | **String s = "hello";** | **String s = new String("hello");** |
| --- | --- | --- |
| Memory Location | String Pool | Heap Memory |
| Object Creation | No new object if it exists | Always creates a new object |
| Performance | Faster (reuses objects) | Slower (creates new object) |
| Recommended? | Yes (for efficiency) | No (unless required) |

**Best Practice:**

* Prefer String s = "hello"; for better performance and memory efficiency.
* Use new String("hello") only when explicitly needed (e.g., creating distinct String objects for modifications).

In Java, **both String s = "hello"; and String s = new String("hello"); are immutable** because they are instances of the String class, which is **immutable by design**.

**Immutability of Strings in Java**

* Once a String object is created, its **value cannot be changed**.
* Any modification (such as concatenation, replacement, or conversion) creates a **new String object** rather than modifying the original.

**Example to Prove Immutability**

java

CopyEdit

String s1 = "hello";

s1.concat(" world"); // This creates a new "hello world" string, but s1 is unchanged.

System.out.println(s1); // Output: hello (original string remains unchanged)

String s2 = new String("hello");

s2 = s2.concat(" world"); // This creates a new string and reassigns it to s2.

System.out.println(s2); // Output: hello world (but original "hello" is still unchanged in memory)

**Key Takeaways**

* **Both String s = "hello"; and String s = new String("hello"); are immutable.**
* Modifications create **new objects** instead of modifying existing ones.
* If you need a **mutable** alternative, use StringBuilder or StringBuffer.