# 🔹 1. String in Java

**✅ What is a String?**

* A **String** is a sequence of characters.
* In Java, strings are **immutable** — once created, their values cannot be changed.

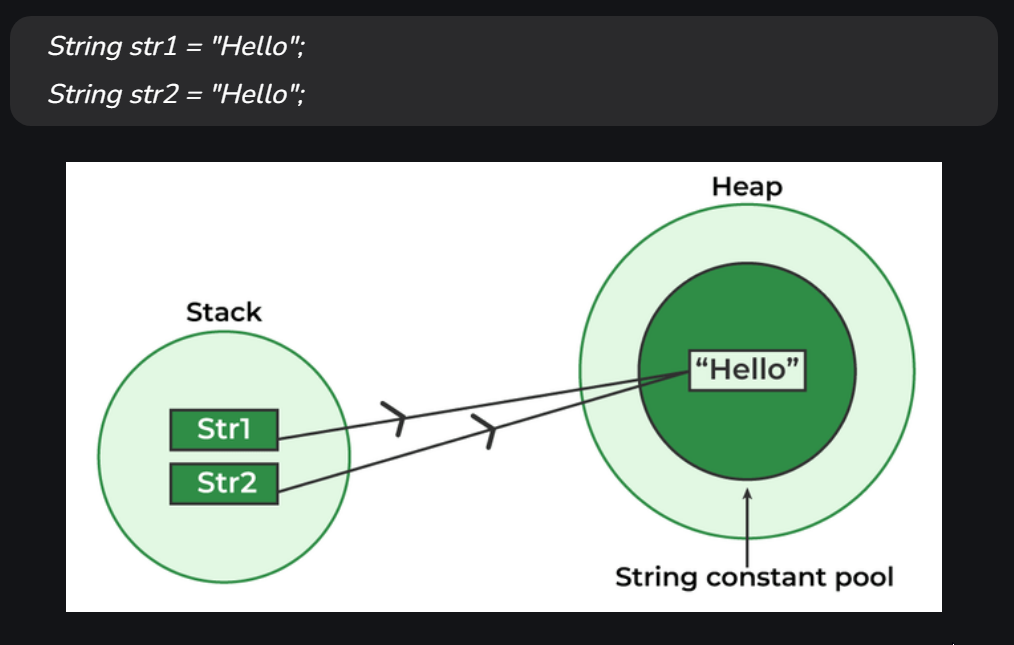
**✅ String Declaration**

String s1 = "Hello"; // Stored in String Constant Pool

String s2 = new String("Hello"); // Stored in Heap

## Memory Allocation to String in Java

the memory allocation depends on **how the string is created** — either as a **string literal** or using the **new keyword**.



A diagram of a string

AI-generated content may be incorrect.

| **Creation Type** | **Memory Area** | **Notes** |
| --- | --- | --- |
| "abc" | String Pool | Shared, reusable |
| new String("abc") | Heap + Pool(literal part) | New heap object, pool ref |
| intern() | String Pool | Forces heap string to pool |
| StringBuilder/StringBuffer | Heap | Mutable, always on heap |

## Why Is String Immutable in Java?

### 🔹 1. Security

* String is heavily used in **sensitive operations** like:
  + File paths
  + Network connections
  + Class loading (Class.forName("com.example.MyClass"))
  + DB credentials (JDBC URLs) username/password

If strings were mutable:

db.connect("jdbc:mysql://..."); // What if someone modified this string after passing?

🔐 **Immutable strings prevent such tampering.**

### 🔹 2. String Pooling (Memory Optimization)

* JVM uses **String Constant Pool** to reuse string literals.
* Immutability ensures that a pooled string **never changes**, allowing safe reuse.

String a = "Java";

String b = "Java"; // Reuses the same object from the pool

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

If String were mutable, changes to a would affect b too, breaking consistency.

### 🔹 3. Thread-Safety Without Synchronization

* Since String is immutable, multiple threads can **safely share** the same string object without synchronization.

✅ No race conditions or locks needed!

### 🔹 4. HashCode Caching

* String's hashCode() is **cached** after it's computed the first time.
* Immutability guarantees the value (and thus hash) **won’t change**, making strings efficient for:
  + HashMaps
  + HashSets

Map<String, String> map = new HashMap<>();

map.put("key", "value"); // HashCode is computed once and reused

### 🔹 5. Simplifies Design

* Immutable objects are easier to reason about, test, and debug.
* You don’t need to worry about unexpected changes to strings.

### ✅ Advantages of String Immutability

| **Advantage** | **Description** |
| --- | --- |
| 🔐 Security | Prevents tampering with sensitive information |
| 🧠 Thread-safety | Multiple threads can use the same string safely |
| 💾 Memory efficiency | Enables pooling and reuse of string literals |
| ⚡ Performance | Enables hashCode caching and faster lookups |
| 🧹 GC friendliness | Fewer changes → less garbage → less GC overhead |
| 🛠️ Simpler APIs | Methods like substring(), replace() return new strings, ensuring original is unchanged |

### 🔁 Real-world analogy:

Think of a string like a **permanent marker on paper** — once written, it cannot be changed. If you want a different sentence, you use a **new sheet**.

# 🔹 2. String Class

* java.lang.String is a final class (cannot be subclassed).
* It implements:
  + Serializable (for object serialization)
  + Comparable<String> (for natural ordering)
  + CharSequence (for character sequence access)
* Thread Safe

**Key Methods:**

* length(), charAt(), substring(), equals(), hashCode(), toLowerCase(), toUpperCase(), trim(), split(), replace(), etc.

**Immutability Example:**

String s = "Hello";

s.concat(" World"); // This won't change 's'

System.out.println(s); // Output: Hello

# 🔹 3. String Constant Pool

**What is it?**

* A special memory region inside the heap (part of **method area** in JVM) used to store string literals.
* When you create String s1 = "hello";, JVM checks the pool — if it exists, it reuses it.

**Example:**

String s1 = "Java";

String s2 = "Java";

System.out.println(s1 == s2); // true (same reference from pool)

## ✅ Advantages of String Pool in Java

The **String Constant Pool** (also known as the **String Intern Pool**) is an optimization technique used by the JVM to manage memory more efficiently. Here's how and why it benefits your application:

### 🔹 1. Memory Efficiency

* When you create strings using literals (e.g., "Hello"), Java stores them in the pool.
* If the same literal is used again, it reuses the existing object rather than creating a new one.

🔸 Example:

String s1 = "Java";

String s2 = "Java";

System.out.println(s1 == s2); // true — both point to the same memory

✅ **Advantage**: Saves memory by avoiding duplication of string objects.

### 🔹 2. Improved Performance

* String comparison using == is faster than .equals() because it compares references.
* Since pool strings share the same reference, you can use == safely for comparison.

✅ **Advantage**: Faster string comparisons when working with interned strings.

### 🔹 3. String Immutability + Pool = Safe Reuse

* Strings in the pool are **immutable** — they cannot be changed.
* This immutability ensures safe reuse without unexpected changes.

✅ **Advantage**: Reduces bugs and supports thread-safe reuse of string values.

### 🔹 4. Reduces Garbage Collection Overhead

* Fewer duplicate strings mean fewer objects on the heap.
* This leads to reduced pressure on the garbage collector.

✅ **Advantage**: Improves JVM performance over time, especially in string-heavy applications.

### 🔹 5. Helps in Compiler Optimization

* The compiler can replace repeated literals with shared references at compile-time.

✅ **Advantage**: Results in leaner bytecode and faster runtime performance.

### ✳️ Summary of Advantages:

| **Benefit** | **Description** |
| --- | --- |
| ✅ Memory Efficiency | Reuses same string objects to save heap space |
| ✅ Fast Comparisons | Reference comparison (==) becomes reliable |
| ✅ Thread Safety | Immutable strings can be shared safely |
| ✅ Less GC Work | Fewer objects created, reducing GC frequency |
| ✅ Compiler Friendly | Enables compile-time optimizations |

# 🔹 4. StringBuffer

* A **mutable** version of String.
* **Thread-safe** (synchronized), hence slower in performance.
* Ideal when strings are modified frequently in multi-threaded environments.

**Example:**

StringBuffer sb = new StringBuffer("Hello");

sb.append(" World");

System.out.println(sb); // Output: Hello World

# 🔹 5. StringBuilder

* Also mutable like StringBuffer.
* **Not thread-safe** (non-synchronized), hence **faster**.
* Suitable for single-threaded use cases.

**Example:**

StringBuilder sb = new StringBuilder("Hello");

sb.append(" World");

System.out.println(sb); // Output: Hello World

# 🔹 6. Difference Between String, StringBuffer, and StringBuilder

| **Feature** | **String** | **StringBuffer** | **StringBuilder** |
| --- | --- | --- | --- |
| Mutability | Immutable | Mutable | Mutable |
| Thread-safe | Yes (immutable) | Yes (synchronized) | No |
| Performance | Slow (due to immutability) | Slower | Faster |
| Use-case | Fixed data | Multi-threaded data change | Single-threaded data change |

# 🔹 7. Important Concepts

## 🔸 String Interning

* The process of storing only one copy of each distinct string value in the string constant pool.

String s1 = new String("hello");

String s2 = s1.intern();

### 🧠 The Golden Rule:

**String.intern() returns a reference to the string in the pool.**

* If the value **already exists in the pool** → returns the pooled object
* If the value **doesn't exist** → adds the current object (x) to the pool and returns it

System.out.println(s2 == "hello"); // true

## 🔸 String Comparison

* == checks **reference** equality.
* .equals() checks **value** equality.

## 🔸 Mutable vs Immutable

* String is immutable — every change creates a new object.
* StringBuilder and StringBuffer are mutable — change modifies same object.

# Interview Questions 🡪

### Program

String s1 = "Java"; // Stored in String Constant Pool  
String s2 = new String("Java"); // New object in Heap  
String s3 = s2.intern(); // s3 points to the pooled "Java"  
  
System.out.println(s1 == s2); // ❌ false — different objects (pool vs heap)  
System.out.println(s1 == s3); // ✅ true — both point to pooled object

### Program

String a = "hello";  
String b = "he" + "llo";  
String c = "he";  
String d = c + "llo";  
  
System.out.println(a == b);  
System.out.println(a == d);  
System.out.println(a.equals(d));

String a = "hello";

String b = "he" + "llo"; // 👈 Compiler optimizes this to "hello" → pooled

String c = "he";

String d = c + "llo"; // 👈 Runtime concatenation → new object on heap

Output – true false true

Compile-time string concatenation goes to **pool**,  
Runtime concatenation results in a **new object on the heap**.

### Program

StringBuilder sb1 = new StringBuilder("abc");  
StringBuilder sb2 = new StringBuilder("abc");  
  
System.out.println(sb1 == sb2);  
System.out.println(sb1.equals(sb2));

sb1.toString().equals(sb2.toString()) // ✅ true

* sb1 == sb2 → false → clearly different objects in heap
* sb1.equals(sb2) → ❌ false  
  👉 Because StringBuilder does **not override equals()**, so it behaves like Object.equals() (i.e., checks reference)

⚠️ Unlike String, StringBuilder/StringBuffer don’t compare content using .equals() — unless you override it manually.

### Program

String part1 = "Hel";  
String part2 = "lo";  
  
String s1 = "Hello";  
String s2 = part1 + part2;  
String s3 = s2.intern();  
  
System.out.println(s1 == s2);  
System.out.println(s1 == s3);

String part1 = "Hel"; // in pool

String part2 = "lo"; // in pool

String s1 = "Hello"; // in pool

String s2 = part1 + part2; // runtime concat → new object in heap

String s3 = s2.intern(); // s3 → reference to "Hello" in pool

### Program

* String a = new String("hello");  
  String b = "hello";  
    
  String c = a.intern();  
  String d = "hel" + "lo";  
  String e = new String("hel" + "lo");  
    
  System.out.println(a == b); // 1  
  System.out.println(b == c); // 2  
  System.out.println(b == d); // 3  
  System.out.println(b == e); // 4  
  System.out.println(e.intern() == b); //e.intern() returns reference from pool (i.e., b), not the heap one

Output – false true true false true

### Program

String x = new StringBuilder("ja").append("va").toString();  
String y = x.intern();  
String z = "java";  
  
System.out.println(x == y); // 1  
System.out.println(y == z); // 2  
System.out.println(x == z); // 3

Output - false true false

Some common strings (like "java", "true", "false", "null", etc.) are **interned by default**.

### Program

String x = new String("springboot");  
String y = x.intern();  
String z = "springboot";  
  
System.out.println(x == y); // 1  
System.out.println(y == z); // 2  
System.out.println(x == z); // 3

Output – True True True

**✅ JVM Behavior (Edge Case):**

* x = new String("springboot");
  + "springboot" literal is **not yet in the pool**.
  + So:
    - "springboot" is added to pool **when intern() is called**
    - But since the value already exists **on the heap** (x), JVM **adds x to the pool**.

y = x.intern(); // returns x itself and puts it in the pool

**So Now:**

| **Variable** | **Points To** |
| --- | --- |
| x | heap object "springboot" |
| y | same heap object (because intern() returns x) |
| z | refers to the **same object** — because z = "springboot" finds it already interned |