String In Java

Java String –

---

1. What is a String in Java?

- A String in Java is a **sequence of characters**.
- Strings are **objects** in Java (not primitive types) and are defined in the java.lang package.
- They are **immutable**, meaning their value cannot be changed once created.

String str = "Hello"; // using string literal

---

2. Creating Strings

a. Using String Literal (stored in String Pool)

String s1 = "Java";
String s2 = "Java"; // Points to same memory location as s1

b. Using new keyword (stored in Heap Memory)

String s3 = new String("Java");

---

3. Memory Management in Strings

Method	Memory Area	Description
String s = "abc";	String Constant Pool	JVM checks pool for existing string
new String("abc")	Heap	Always creates a new object

 "abc" and new String("abc") are **not the same object**.

---

4. Immutability of Strings

- Once a string is created, its value **cannot be changed**.

- Any modification (e.g., concatenation) creates a **new string** object.

String s = "Hello";
s.concat(" World");
System.out.println(s); // Output: Hello

To update:

s = s.concat(" World");
System.out.println(s); // Output: Hello World

---

5. Commonly Used String Methods

Method	Description	Example
length()	Returns length of string	"Hello".length() → 5
charAt(int index)	Returns character at given index	"Java".charAt(1) → 'a'
substring(int)	Returns substring from given index	"Hello".substring(2) → "llo"
substring(int, int)	From start index to end-1	"Hello".substring(1,4) → "ell"
equals()	Compares strings (case-sensitive)	"a".equals("A") → false
equalsIgnoreCase()	Case-insensitive comparison	"a".equalsIgnoreCase("A") → true
contains()	Checks if substring is present	"abc".contains("b") → true
replace()	Replaces old char/string with new	"abc".replace('a','x') → "xbc"
toLowerCase()	Converts to lowercase	"Java".toLowerCase() → "java"
toUpperCase()	Converts to uppercase	"java".toUpperCase() → "JAVA"
trim()	Removes whitespace from start/end	" Hello ".trim() → "Hello"
split()	Splits string into array	"a,b,c".split(",")

---

6. String Comparison

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

System.out.println(a == b);         // false → compares references
System.out.println(a.equals(b));    // true  → compares values
```

---

7. StringBuilder vs StringBuffer vs String

Feature	String	StringBuilder	StringBuffer
Mutability	Immutable	Mutable	Mutable
Thread-safe	No	No	Yes (synchronized)
Performance	Slow	Faster	Slower (due to sync)

Example: Using StringBuilder

```
StringBuilder sb = new StringBuilder("Hello");
sb.append(" World");
System.out.println(sb); // Output: Hello World
```

---

8. String Concatenation

- Using + operator:

```
String name = "John";
String greeting = "Hello " + name;  // Efficient for small concat
```

- Using concat() method:

```
String a = "Hello";
String b = a.concat(" World");
```

- For loops/multiple strings → Use StringBuilder:

```
StringBuilder sb = new StringBuilder();
for(int i=0; i<3; i++) {
```

```
    sb.append("Repeat ");
}
System.out.println(sb);  // Output: Repeat Repeat Repeat
```

---

9. String Interning

- The .intern() method returns a canonical representation from the pool.

```
String a = new String("Java");
String b = a.intern();  // Moves to String pool
```

---

10. Convert between Data Types and Strings

```
int x = 10;
String str = String.valueOf(x);  // int to String

String s = "123";
int y = Integer.parseInt(s);     // String to int
```

How to Create String Objects in Java

In Java, you can create a String object in **two main ways**:

---

1. Using String Literals

```
String str1 = "Java";
String str2 = "Java";
```

 Behind the scenes:

- String literals are stored in a special memory area called the **String Constant Pool (SCP)**.
- If a string with the same content already exists in the pool, **no new object is created**.
- str1 and str2 will point to the **same object** in memory.

 Example:

```
public class Main {
   public static void main(String[] args) {
      String str1 = "Hello";
      String str2 = "Hello";

      System.out.println(str1 == str2);     // true (same reference)
      System.out.println(str1.equals(str2)); // true (same content)
   }
}
```

---

2. Using new Keyword

```
String str3 = new String("Java");
String str4 = new String("Java");
```

 Behind the scenes:

- This always creates a **new String object** in the **heap memory**, even if the value already exists in the SCP.
- str3 and str4 will refer to **different objects** even though they have the same content.

 Example:

```
public class Main {
   public static void main(String[] args) {
      String str3 = new String("Hello");
      String str4 = new String("Hello");

      System.out.println(str3 == str4);     // false (different reference)
      System.out.println(str3.equals(str4)); // true (same content)
   }
}
```

□ Comparison Table

Feature	Using String Literal	Using new Keyword
Memory Location	String Constant Pool	Heap memory
Duplicate String Handling	Avoids duplicate	Creates new every time
Reference Equality (==)	Can be true if same value	Always false for same value
Content Equality (equals)	true if values match	true if values match

Heap Object Creation vs String Constant Pool (SCP) in Java

In Java, String objects can be created in two memory areas:

1. **Heap Memory** – via new keyword
2. **String Constant Pool (SCP)** – via **string literals**

---

1. Heap Object Creation using new Keyword

```
String str1 = new String("Hello");
String str2 = new String("Hello");
```

□ What Happens in Memory?

- A new String object is **always created in the heap**, even if "Hello" already exists in SCP.
- Internally, the "Hello" literal also exists in SCP, but the object reference str1/str2 points to a **new heap object**.

□ Result:

```
System.out.println(str1 == str2);     // false (different heap objects)
System.out.println(str1.equals(str2)); // true (same content)
```

---

2. String Constant Pool (SCP) using String Literals

```
String str3 = "Hello";
String str4 = "Hello";
```

□ What Happens in Memory?

- Both str3 and str4 refer to the **same object in SCP**.
- If "Hello" already exists in SCP, it is reused.
- No new object is created for str4.

□ Result:

```
System.out.println(str3 == str4);     // true (same SCP object)
System.out.println(str3.equals(str4)); // true (same content)
```

□ Memory Behavior Comparison

Aspect	new String("Hello")	"Hello"
Memory Location	Heap + SCP (literal reused)	Only SCP
New Object Created?	Yes (in heap)	No (if already in SCP)
Duplicate Handling	No check, always new object	Checks SCP, avoids duplicates
Reference Equality (==)	false (different heap objects)	true (same SCP object)
Content Equality (equals)	true if content is same	true if content is same

String Immutability in Java

What is Immutable String?

A **String is immutable** in Java, which means:

Once a String object is created, its value cannot be changed. Any modification results in the creation of a **new String object**.

 Example:

String s = "Hello";
s.concat(" World");

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

 Even though we used concat(), the original string s did **not change**.

To actually modify it:

s = s.concat(" World"); // Now it stores the new object
System.out.println(s); // Output: Hello World

 Why is String Immutable in Java?

Here are the **reasons and benefits** behind this design choice:

---

1. Security

- Many Java classes like URL, File, Network connections, and ClassLoaders use strings.
- If a String were mutable, someone could change the contents after validation.

String url = "https://bank.com/login";

 Immutability protects such URLs from being modified maliciously.

---

2. String Pool Reusability

- Java maintains a **String Constant Pool**.
- Immutability allows **safe sharing** of string objects between different parts of code.

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

```
System.out.println(a == b); // true (same object reused)
```

---

3. Caching & Performance

- Hash code of a string is frequently used in collections like HashMap, HashSet.
- Since the content never changes, the **hashCode is cached**, improving performance.

```
String s = "data";
int h1 = s.hashCode();
int h2 = s.hashCode(); // Reused, no recomputation
```

---

4. Thread Safety

- Immutable objects are **naturally thread-safe** because they can't be changed.
- No need for synchronization when multiple threads access the same string.

---

5. Predictable Behavior

- Developers can trust that a string reference won't change accidentally.

```
method("Hello");
```

You are guaranteed that the "Hello" won't be altered inside the method.

□ How is Immutability Achieved Internally?

```
public final class String {
   private final char value[]; // 'final' ensures immutability
}
```

- The String class is final → It **cannot be subclassed**.
- The internal char[] array is also **final and private** → No direct access or modification.

□ What If We Need Mutable Strings?

Use:

- StringBuilder – fast, **not thread-safe**
- StringBuffer – thread-safe, slower

```
StringBuilder sb = new StringBuilder("Hello");
sb.append(" World");
System.out.println(sb); // Hello World
```

StringBuffer vs StringBuilder in Java

Why use StringBuffer or StringBuilder?

- Since String objects are **immutable**, every modification (like append, insert) creates a **new object**, which is inefficient.
- To improve performance when doing **many string manipulations**, Java provides **mutable** string classes:
  - StringBuffer
  - StringBuilder

---

1. StringBuffer – Thread-safe, Synchronized

- Introduced in Java 1.0
- Used when **multiple threads** might access and modify the same string.
- All methods are **synchronized**, so **slower** than StringBuilder.

Example:

```
public class Main {
   public static void main(String[] args) {
      StringBuffer sb = new StringBuffer("Hello");
      sb.append(" World");
```

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

---

2. StringBuilder – Non-thread-safe, Faster

- Introduced in Java 1.5
- Used when **only a single thread** is accessing it.
- **Not synchronized**, hence **faster** and more efficient.

Example:

```
public class Main {
    public static void main(String[] args) {
        StringBuilder sb = new StringBuilder("Java");
        sb.append(" Rocks!");
        System.out.println(sb); // Output: Java Rocks!
    }
}
```

□ Common Methods (StringBuffer & StringBuilder)

Method	Description	Example Output
.append(str)	Adds string at end	HelloWorld
.insert(pos, str)	Inserts string at position	HeJavallo
.replace(start, end, str)	Replaces between indices	HeJava
.delete(start, end)	Deletes part of string	Heo
.reverse()	Reverses the characters	olleH
.length()	Returns current length	5
.capacity()	Current allocated buffer size	Default: 16 + length

When to Use What?

Scenario	Recommended Class
Constant strings	String
Heavy string modification (1 thread)	StringBuilder
Heavy string modification (multi-threaded)	StringBuffer

Difference Between String, StringBuffer, and StringBuilder in Java

---

□ Overview

Feature	String	StringBuffer	StringBuilder
Mutability	Immutable	Mutable	Mutable
Thread-safe	Yes (due to immutability)	Yes (synchronized methods)	No
Performance	Slow (due to immutability)	Moderate (synchronized)	Fast (no synchronization)
Introduced in	Java 1.0	Java 1.0	Java 1.5
Recommended for	Fixed constant strings	Multi-threaded string manipulation	Single-threaded string manipulation
Package	java.lang	java.lang	java.lang
Default Capacity	N/A	16 characters (expandable)	16 characters (expandable)

Code Demonstration

□ String (Immutable)

```
String s = "Hello";
s.concat(" World");
System.out.println(s);  // Output: Hello (unchanged)
```

□ StringBuffer (Mutable & Thread-Safe)

```
StringBuffer sb = new StringBuffer("Hello");
sb.append(" World");
System.out.println(sb); // Output: Hello World
```

□ StringBuilder (Mutable & Non-Thread-Safe)

```
StringBuilder sb = new StringBuilder("Hello");
sb.append(" World");
System.out.println(sb); // Output: Hello World
```

□ Internal Memory Behavior

Operation	String	StringBuffer/StringBuilder
Modification	Creates new object	Modifies the existing object
Memory Efficient?	No (more garbage)	Yes (in-place updates)

□ Use-case Recommendations

Situation	Use
Fixed value that never changes	String
Concatenation or manipulation in single-threaded app	StringBuilder

Situation	Use
Manipulation in multi-threaded environment	StringBuffer