**Strings**:

Definition: A **String** in Java is immutable — once created, its value cannot be changed.

Example:

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

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

Tricky Interview Question:  
Q: Why is String immutable in Java?

Answer:

* Security: used in class loading, file paths, networking, etc.
* Thread-safety: No unexpected changes in multi-threaded environments.
* String pool optimization: Immutable strings can be shared.

**Real-world example:**If URLs or database credentials are passed as Strings, immutability ensures they’re not altered during program execution.

**🧠 What is the String Pool?**

The String Pool (also called String Intern Pool or String Constant Pool) is a special memory area in Java inside the heap, where JVM stores String literals.

If two string literals have the same content, they both point to the same object in the pool (for memory optimization).

🟩 Why? To save memory.

Since Strings are immutable, the JVM reuses common string literals instead of creating new ones every time.

✅ Example:

String s1 = "Java"; String s2 = "Java";

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

System.out.println(s1.equals(s2)); // true — same value

👉 Because both s1 and s2 point to the same object in the String pool.

🟥 Now check this one:

String s3 = new String("Java");

System.out.println(s1 == s3); // false — different reference

System.out.println(s1.equals(s3)); // true — same value

Why false for ==?

Because new String() forces the creation of a new object in heap memory, not reused from the pool.

**what if we want to manually move s3 into the pool**?

Use **intern**():

String s3 = new String("Java");

String s4 = s3.intern();

System.out.println(s1 == s4); // true — both point to the pool object

💬 **Interview Question:**

**Q: What happens in memory when we do:**

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

Answer:

* a refers to the pooled literal "Hello"
* b refers to a new object in the heap with value "Hello"
* b.intern() will return a

**What is a String Literal?**

A String literal is a sequence of characters enclosed in double quotes — like:

String s = "Java";

When you write a String like that, it’s called a literal and is automatically stored in the String Pool by the JVM.

🧪 Example:

String s1 = "Hello"; String s2 = "Hello";

System.out.println(s1 == s2); // true (both refer to the same literal in the String Pool)

System.out.println(s1.equals(s2)); // true

| **Way of creating String** | **Stored In** | **Reused from Pool?** | **== comparison** |
| --- | --- | --- | --- |
| String s = "Hello"; | String Pool | ✅ Yes | true |
| String s = new String("Hello"); | Heap Memory | ❌ No | false |

✅ Why true for == here? Because "Hello" is a literal, so JVM stores it once in the pool and both variables point to the same reference.

🎯 **Interview Question:**

Q: **How many objects are created in memory**?

String s = new String("Java");

Answer:  
2 objects:

1. "Java" literal in the String pool
2. A new object in the heap using new

🧱 . **StringBuffer (**Mutable & Thread-safe)

Definition: A mutable sequence of characters. Thread-safe (methods are synchronized).

Example:

StringBuffer sb = new StringBuffer("Hello");

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

**Tricky Interview Question**:  
Q: Why might StringBuffer be slower in a single-threaded environment?

Answer: Because its methods are synchronized, which adds overhead even when not needed.

  ⚡ . **StringBuilder** (Mutable & NOT Thread-safe)

Definition: Like StringBuffer but faster and not synchronized — not thread-safe.

Example:

StringBuilder sb = new StringBuilder("Fast");

sb.append(" & Furious");

System.out.println(sb); // Output: Fast & Furious

**Tricky Interview Question:**Q: When should you use StringBuilder over StringBuffer?

Answer: In a single-threaded context or when thread-safety is not a concern, use StringBuilder for better performance.

**🧠 Why is StringBuilder not thread-safe?**

Because none of its methods are synchronized — meaning multiple threads can access and modify a StringBuilder object at the same time, potentially causing inconsistent or unpredictable results.

| **Feature** | **String** | **StringBuffer** | **StringBuilder** |
| --- | --- | --- | --- |
| Mutable? | ❌ No | ✅ Yes | ✅ Yes |
| Thread-safe? | ✅ Yes | ✅ Yes | ❌ No |
| Performance | Medium | Slow (synchronized) | Fast (non-synchronized) |
| Use case | Constant data | Multi-threaded concat | Single-threaded concat |

| **Class** | **Thread Safety** | **Use When** |
| --- | --- | --- |
| StringBuffer | ✅ Yes | In multi-threaded code |
| StringBuilder | ❌ No | In single-threaded code |
| String | ✅ Immutable | When content won’t change |

🔥 Tricky & Scenario-Based Questions on Strings and Builders

Q1: What will be the output?

String s1 = "Java"; String s2 = "Ja" + "va";

System.out.println(s1 == s2);

📝 Answer: true  
Explanation: Both are compile-time constants, so s2 gets optimized and points to the same reference in the String Pool.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

🧩 Q2: What will be the output?

String a = "Hello"; String b = "Hel"; String c = b + "lo";

System.out.println(a == c);

📝 Answer: false  
Explanation: b is a variable, so b + "lo" is computed at runtime → creates a new String object, not from the pool.

🧠 Pro Tip: Compile-time concatenation is pooled; runtime is not.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

🧩 Q3: What happens when you do this?

StringBuilder sb1 = new StringBuilder("Java");

StringBuilder sb2 = new StringBuilder("Java");

System.out.println(sb1 == sb2);

System.out.println(sb1.equals(sb2));

📝 Answer:

* sb1 == sb2 → false (different objects)
* sb1.equals(sb2) → false (StringBuilder doesn’t override equals — so it behaves like Object.equals) but String can support

 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

🧩 Q4: Which is faster?

String result = ""; for (int i = 0; i < 1000; i++) { result += i; }

📝 Answer: Very slow!  
Explanation: Every += creates a new String object → performance hit due to immutability

✅ Recommended:

StringBuilder sb = new StringBuilder();

for (int i = 0; i < 1000; i++) { sb.append(i); }

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

🧩 Q5: How many objects are created?

String s = new String("Java");

📝 Answer: Two  
Explanation:

* "Java" → String literal in pool
* new String("Java") → New object in heap

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Q6: Can a StringBuilder be added to the String pool?

📝 Answer: No  
Only Strings can be interned using .intern(). StringBuilder must be converted to String first:

StringBuilder sb = new StringBuilder("abc");

String s = sb.toString().intern();

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Q7: Will this compile?

StringBuilder sb = new String("abc"); // ??

📝 Answer: ❌ Compile-time error  
Because String cannot be passed to StringBuilder constructor in that way.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Q8. Can String be null or empty? What’s the difference?

✅ Yes:

* null: variable doesn’t reference any object.
* empty: variable refers to an object, but that object has no characters.
* Examples:
* String s1 = null;  
  String s2 = ""; // s2.length() = 0

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

🧠 Q9. Can you change the contents of a String?

❌ No — Strings are immutable.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Q10. What’s the default capacity of a new StringBuilder?

✅ 16 characters  
So new StringBuilder() creates buffer with 16-char capacity. If it exceeds, capacity is doubled.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

🧠 Q11. Will this compile?

StringBuffer sb = "hello";

❌ No — String literal can’t be assigned to StringBuffer.

Use:

StringBuffer sb = new StringBuffer("hello");

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

🧠 Q12. What is the difference between equals() and == with String?

* == compares reference (memory address)
* equals() compares values (overridden in String)

 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

🧠 Q13. Can StringBuilder be used in a multithreaded program?

❌ Not safely. Use StringBuffer instead if you need synchronization.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

🧠 Q14. What is the capacity vs length in StringBuilder?

* length(): actual number of characters stored.
* capacity(): size of internal buffer (defaults to 16 or auto-resizes).

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

🧠 Q15. Is StringBuilder faster than String in loops?

✅ Yes! Especially for concatenation inside loops — use StringBuilder to avoid creating too many temporary String objects.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

🧠 Advanced & Tricky Interview Questions on String & Related Concepts:

1. ❓ What is the difference between: String s = "abc"; and String s = new String("abc");

✅ Answer: First stores in String pool; second creates a new object in the heap (plus a reference in pool if not present already).

1. ❓ Can String be subclassed? ✅ Answer: No. String is final. You cannot extend it.
2. ❓ Is this valid?

String s1 = null; if (s1.equals("hello")) { }

✅ Answer: No! NullPointerException. Always call equals on constant: "hello".equals(s1)

1. ❓ What is the output?

String a = "Hello"; String b = "Hel" + "lo"; System.out.println(a == b); // ?

✅ Answer: true — because both are compile-time constants and refer to same object in pool.

1. ❓ What is the output?

String a = "Hello"; String b = "Hel"; String c = b + "lo"; System.out.println(a == c); // ?

✅ Answer: false — because b is a variable, so b + "lo" is computed at runtime → new object.

1. ❓ Does StringBuilder override hashCode()?

✅ Answer: No — it uses Object’s hashCode, based on reference. So two builders with same content have different hash codes.

1. ❓ Can StringBuilder be used as a key in HashMap?

✅ Answer: Not recommended, unless you override equals() and hashCode() properly — because it’s mutable and may break map integrity.

1. ❓ What is the output?

StringBuilder sb = new StringBuilder("test"); sb.insert(2, "abc"); System.out.println(sb);

✅ Answer: teabcst

1. ❓ What happens when you call reverse() on StringBuilder?

✅ Answer: The sequence of characters is reversed in-place.

1. ❓ What if you exceed the default capacity of StringBuilder?

✅ Answer: Capacity increases automatically — it’s resized internally using the formula: newCapacity = (oldCapacity + 1) \* 2

1. ❓ How can you make StringBuilder thread-safe?

✅ Answer: Use synchronization manually or use StringBuffer instead.

1. ❓ Can two StringBuilders with the same characters be considered equal?

✅ No, unless you compare them using sb1.toString().equals(sb2.toString())

1. ❓ Can final StringBuilder be modified?

✅ Yes. Final means reference can’t change — but internal content is still mutable.

Example: final StringBuilder sb = new StringBuilder("Hi"); sb.append(" there"); // valid

1. ❓ Difference between concat() and + operator for String?

✅ Both result in new Strings — concat() only works with strings, + operator can handle nulls and non-strings by converting them.

1. ❓ Why does using += on String inside loop lead to performance issues?

✅ Because each += creates a new object, leading to high memory usage and poor performance. Use StringBuilder instead.