**Core String Interview Questions with Answers**

**1. Why is String immutable in Java? What are the benefits?**

**Answer:**

* String immutability ensures:
  + **Thread safety**: No synchronization needed.
  + **Security**: Used in class loaders, file paths, etc.
  + **Hashcode caching**: Speeds up map lookups.
  + **String pooling**: Saves memory.
  + Prevents data manipulation in sensitive areas (e.g., database URLs).

**Cross-question:**

* *How does immutability affect memory usage in high-throughput systems?*

**2. Explain String, StringBuilder, and StringBuffer in terms of performance and thread safety.**

**Answer:**

| **Feature** | **String** | **StringBuilder** | **StringBuffer** |
| --- | --- | --- | --- |
| Mutability | Immutable | Mutable | Mutable |
| Thread-Safe | Yes (due to immutability) | No | Yes (synchronized) |
| Performance | Low (new object on change) | High (single-threaded) | Moderate (sync overhead) |

**Cross-question:**

* *How would you design a thread-safe StringBuilder without synchronized keyword?*

**3. How does the Java String pool work internally?**

**Answer:**

* JVM maintains a special memory called the **String Constant Pool**.
* String literals are stored here; if a new literal is created, JVM checks the pool first.
* String s1 = "test"; String s2 = "test"; // s1 == s2

**Cross-question:**

* *What happens when we use new String("test") instead of a literal?*

**4. What are the performance implications of + operator in loops for String concatenation?**

**Answer:**

* + creates a new StringBuilder each time.
* Leads to many temporary objects → **heap pressure & GC overhead**.
* Use StringBuilder/StringBuffer for performance.

**Cross-question:**

* *How would you optimize string concatenation of 1000 strings?*

**5. What happens internally when you call s.concat("world") on String s = "Hello"?**

**Answer:**

* A new String object is created; s remains unchanged.
* concat() does: return new StringBuilder(this).append("world").toString();

**6. Can you override equals() and hashCode() for String class? Why or why not?**

**Answer:**

* No, because String is a final class.
* Its equals() checks value equality; hashCode() is overridden to reflect character content.

**7. How does intern() method work? When would you use it?**

**Answer:**

* intern() places a string in the **string pool** if not already present.
* Use it to save memory when many strings have the same value.

java

CopyEdit

String a = new String("test").intern();

**Cross-question:**

* *What are the downsides of excessive use of intern()?*

**8. What is the difference between == and .equals() with Strings?**

**Answer:**

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

**Cross-question:**

* *Can you give a real-time bug example caused by using == with Strings?*

**9. Is String thread-safe? Why?**

**Answer:**

* Yes, because it is **immutable**.
* Shared access doesn’t risk modification.

**10. What is the output of the following snippet?**

java

CopyEdit

String a = "Hello";

String b = new String("Hello");

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

System.out.println(a.equals(b)); // ?

**Answer:**

* a == b → false (different references)
* a.equals(b) → true (same value)

**🔷 Advanced String Questions**

**11. What is the memory structure of String in Java (Heap, Pool, PermGen/MetaSpace)?**

**Answer:**

* String literals go to **String Constant Pool** (stored in heap since Java 7).
* Interned Strings go to the pool.
* new String() objects are stored in **heap memory**.

**12. Explain String deduplication in Java 8+.**

**Answer:**

* JVM option: -XX:+UseStringDeduplication
* Enables GC to detect duplicate character arrays and point them to a single char[].
* Useful in high-duplicate environments (e.g., large logs, XML parsing).

**13. How does StringJoiner work internally? When would you use it?**

**Answer:**

* Introduced in Java 8.
* More efficient alternative for joining delimited strings.
* Internally uses StringBuilder.

**14. What is the difference between String.split() and using Pattern.split()?**

**Answer:**

* String.split() uses regex under the hood → slower for large-scale.
* Pattern.split() reuses the compiled pattern → **faster for repeated use**.

**Example:**

java

CopyEdit

Pattern p = Pattern.compile(",");

p.split("a,b,c");

**15. How can you reverse a String in Java efficiently?**

**Answer:** java

CopyEdit

String input = "Hello";

String reversed = new StringBuilder(input).reverse().toString();

**🔷 Design/Real-world Questions**

**16. How would you implement a custom ImmutableString class?**

**Answer:**

* Final class.
* Final internal char array.
* Deep copy on construction and on getter.
* No mutating methods.

**17. How would you store millions of repeating strings efficiently in memory?**

**Answer:**

* Use:
  + **intern()**
  + **String deduplication**
  + **Custom canonicalization using Map<String, String>**

**18. How do internationalization (Unicode, encoding) issues affect String operations?**

**Answer:**

* Java String is UTF-16 encoded.
* Length ≠ number of characters (because of surrogate pairs).
* charAt() can mislead in multibyte characters.

**19. What is the difference between substring() in Java 6 vs Java 7+?**

**Answer:**

* Java 6: substring() shares the original char[] → **memory leak risk**.
* Java 7+: substring() creates new char[].

**20. What is the behavior of String.valueOf(null)?**

**Answer:**

* Returns "null" (string literal), doesn’t throw NullPointerException.