What is a String in Java?

A String in Java is an object that represents a sequence of Unicode characters. It's implemented as an immutable (unchangeable) array of characters, providing various methods for string manipulation.

String Memory Architecture

1. String Storage Components

Java's string handling involves three key memory areas:

1. **Heap Memory - Where all Java objects are stored**
2. **String Pool - A special area within the heap for storing string literals**
3. **Runtime Constant Pool - Part of the method area that stores symbolic references**

2. String Pool (String Intern Pool)

The string pool is a special memory region that:

- **Stores unique string literals to save memory**
- **Maintains a hashtable-like structure for quick lookup**
- **Is part of the heap memory (since Java 7)**

Key Characteristics:

- **Contains only unique string literals**
- **Shared across entire JVM**
- **Exists to optimize memory usage**

How Strings are Created and Stored

Case 1: Using String Literal

java

Copy

String s1 = "Hello";

String s2 = "Hello";

Memory Behavior:

1. **JVM checks if "Hello" exists in string pool**
2. **If not found, creates new entry in pool**
3. **If found, returns reference to existing string**
4. **Both s1 and s2 point to the same memory location**

Case 2: Using new Keyword

java

Copy

String s3 = new String("Hello");

Memory Behavior:

1. **Creates a new String object in heap memory (outside pool)**
2. **The literal "Hello" is still added to string pool if not present**
3. **s3 refers to the heap object, not the pool object**

String Interning

The intern() method allows runtime string pooling:

java

Copy

String s4 = new String("Hello").intern();

Behavior:

- **If pool already contains equal string, returns pool reference**
- **Otherwise, adds string to pool and returns reference**
- **After interning, s4 will point to pool object**

String Immutability: Internal Implementation

Why Strings are Immutable

1. **Security: Strings used in sensitive operations (file paths, network connections)**
2. **Thread Safety: Can be shared across threads without synchronization**
3. **Hashcode Caching: Hashcode can be cached since content won't change**
4. **Performance: Enables string pooling optimization**

How Immutability is Implemented

Internally, a String contains:

java

Copy

private final char value[]; // Character array

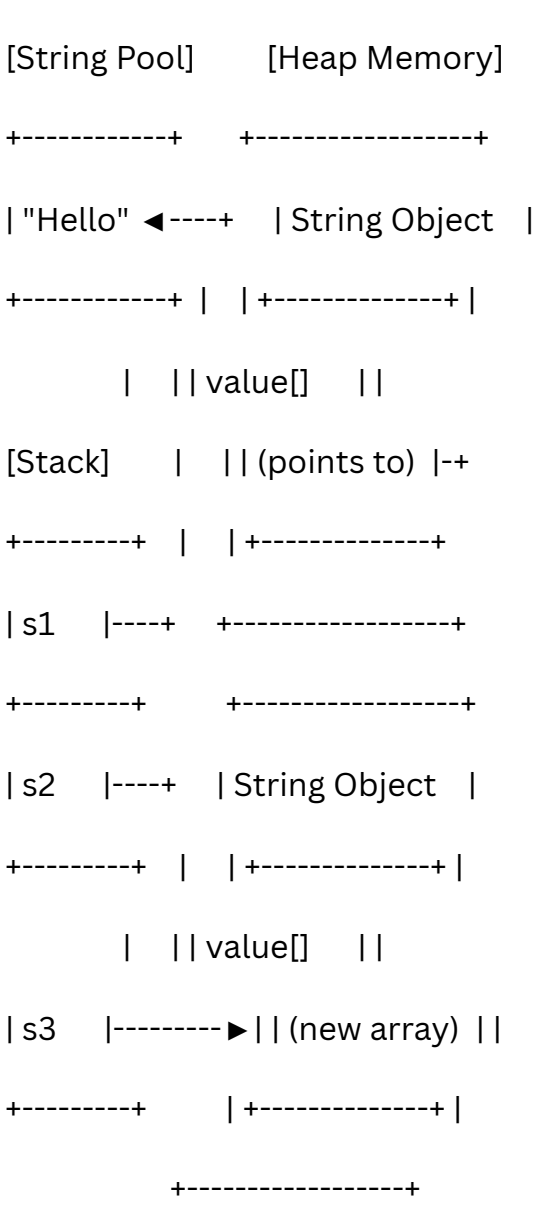
private int hash; // Cache for hashcode

The final keyword ensures:

- **Reference to the array cannot be changed**
- **Array contents can't be modified through String methods**

Memory Diagram

Copy



Important Internal Details

1. Character Storage

- **Prior to Java 9: char[] (2 bytes per character)**
- **Java 9+: byte[] with encoding flag (saves memory for ASCII strings)**

2. Hashcode Caching

- **Strings cache their hashcode after first computation**
- **Stored in private int hash (default 0)**
- **Explains why Strings make good HashMap keys**

3. Compiler Optimizations

Compile-Time Concatenation:

java

Copy

```
String s = "Hello" + "World"; // Becomes "HelloWorld" at compile time
```

StringBuilder Optimization:

java

Copy

```
String s = "";
for(int i=0; i<10; i++) {
   s += i; // Converted to StringBuilder operations by compiler
}
```

4. Garbage Collection and String Pool

- **Strings in pool are eligible for GC if no references exist**
- **Pool maintained by JVM, not part of application heap**
- **Can be tuned with JVM options (-XX:StringTableSize)**

Performance Implications

Advantages of String Pool

- **Memory efficiency (reuse of common strings)**
- **Faster comparison (can use == for pooled strings)**
- **Reduced GC overhead (fewer duplicate strings)**

Disadvantages

- **Pool maintenance overhead**
- **Potential memory leak if too many unique strings interned**
- **Not suitable for temporary strings**

Practical Examples

Example 1: String Comparison

java

Copy

```
String a = "Hello";

String b = "Hello";

String c = new String("Hello");

String d = c.intern();


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

System.out.println(a == c); // false (c is in heap)

System.out.println(a == d); // true (d is interned)
```

Example 2: Memory Impact

java

Copy

```
// Creates 1000 distinct objects in heap

for(int i=0; i<1000; i++) {

   new String("Hello");

}


// Uses single pool object

for(int i=0; i<1000; i++) {

   String s = "Hello";

}
```

Advanced Topics

1. String Deduplication (Java 8u20+)

- **JVM feature that identifies duplicate strings in heap**
- **Converts them to point to same char array**
- **Enabled with -XX:+UseStringDeduplication**

2. Compact Strings (Java 9+)

- **Stores ASCII strings as byte[] instead of char[]**
- ▢ **Reduces memory footprint by ~50% for ASCII-heavy apps**
- **Transparent to developers**

3. StringTable Size Tuning

- **Larger table reduces collisions in string pool**
- **Set with -XX:StringTableSize=N (default 60013 in Java 11)**

Best Practices

1. **Use literals for constant strings to leverage pooling**
2. **Avoid new String() unless explicitly needed**
3. **Use intern() judiciously - can help but may increase pool size**
4. **Prefer StringBuilder for complex string building**
5. **Be cautious with large substring operations - can lead to memory leaks (pre-Java 7u6)**

Understanding these internal details helps write more efficient Java code and avoid common pitfalls related to string handling.

Strings in Java: A Comprehensive Guide

Introduction to Strings in Java

In Java, a **String** is an object that represents a sequence of characters. Strings are immutable (unchangeable) sequences of Unicode characters and are one of the most commonly used classes in Java.

Key Characteristics of Strings:

- **Immutable**: Once created, cannot be modified

- **Stored in String Pool**: For memory efficiency

- **Unicode Support**: Can represent international characters

- **Object Type**: Not a primitive data type

String Class

Creating Strings

```java
// Using string literal (stored in string pool)
String str1 = "Hello";

// Using new keyword (creates new object in heap)
String str2 = new String("Hello");
```

Important String Methods

1. Basic Operations

```java
String s = "Hello World";

int length = s.length(); // 11
char ch = s.charAt(1);   // 'e'
String sub = s.substring(6); // "World"
String sub2 = s.substring(0, 5); // "Hello"
```

2. Searching

```java
int index = s.indexOf('o'); // 4
int lastIndex = s.lastIndexOf('o'); // 7
boolean contains = s.contains("World"); // true
```

3. Comparison

```java
boolean eq1 = s.equals("hello world"); // false
boolean eq2 = s.equalsIgnoreCase("hello world"); // true
int cmp = s.compareTo("Hello"); // positive number
boolean starts = s.startsWith("Hello"); // true
boolean ends = s.endsWith("World"); // true
```

4. Modification (returns new string)

```java
String lower = s.toLowerCase(); // "hello world"
String upper = s.toUpperCase(); // "HELLO WORLD"
String trimmed = " hello ".trim(); // "hello"
String replaced = s.replace('o', 'x'); // "Hellx Wxrld"
String concat = s.concat("!!"); // "Hello World!!"
```

5. Conversion

```java
```

```
char[] chars = s.toCharArray();

byte[] bytes = s.getBytes();

String[] parts = s.split(" "); // ["Hello", "World"]

String joined = String.join("-", "Hello", "World"); // "Hello-World"

String formatted = String.format("Value: %d", 10); // "Value: 10"
```

String Immutability

```java
String s1 = "Hello";

String s2 = s1.concat(" World"); // Creates new string

System.out.println(s1); // Still "Hello"

System.out.println(s2); // "Hello World"
```

String Pool Concept

```java
String a = "Hello"; // Goes to string pool

String b = "Hello"; // Reuses from string pool

String c = new String("Hello"); // Creates new object in heap

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

System.out.println(a == c); // false (different references)

System.out.println(a.equals(c)); // true (same content)
```

```

## StringBuilder Class

**StringBuilder** is a mutable sequence of characters, used when you need to modify strings frequently.

### Why Use StringBuilder?

- More efficient than String for multiple modifications

- Avoids creating multiple intermediate String objects

- Better performance for concatenation in loops

### Creating StringBuilder

```java

StringBuilder sb = new StringBuilder(); // Initial capacity 16

StringBuilder sb2 = new StringBuilder(100); // Custom capacity

StringBuilder sb3 = new StringBuilder("Hello"); // With initial value

```

Important StringBuilder Methods

1. Appending

```java

sb.append("Hello");

sb.append(" ");

sb.append("World"); // "Hello World"

```

2. Inserting

```java

sb.insert(5, ","); // "Hello, World"

```

3. Deleting

```java

sb.delete(5, 6); // "Hello World"

sb.deleteCharAt(4); // "Hell World"

```

4. Replacing

```java

sb.replace(5, 6, ","); // "Hello,World"

```

5. Other Operations

```java

sb.reverse(); // "dlroW ,olleH"

sb.setCharAt(0, 'D'); // "DlroW ,olleH"

int cap = sb.capacity(); // Current capacity

sb.ensureCapacity(100); // Ensure minimum capacity

sb.trimToSize(); // Trims to current length

```

Example: Efficient String Concatenation

```java
// Inefficient with String
String result = "";
for (int i = 0; i < 100; i++) {
    result += i; // Creates new String each time
}

// Efficient with StringBuilder
StringBuilder sb = new StringBuilder();
for (int i = 0; i < 100; i++) {
    sb.append(i);
}
String result = sb.toString();
```

StringBuffer Class

StringBuffer is similar to StringBuilder but is thread-safe (synchronized). Use it when thread safety is required.

Key Differences:

- StringBuffer: Synchronized, thread-safe, slightly slower
```

- StringBuilder: Not synchronized, faster, use when thread safety isn't needed

Methods

Same as StringBuilder, but all methods are synchronized.

StringTokenizer Class

StringTokenizer is a legacy class (from Java 1.0) for breaking strings into tokens.

Creating StringTokenizer

```java
StringTokenizer st = new StringTokenizer("Hello World"); // Default delimiter is whitespace
StringTokenizer st2 = new StringTokenizer("Hello,World", ","); // Custom delimiter
StringTokenizer st3 = new StringTokenizer("Hello,World", ",", true); // Return delimiters as tokens
```

Important Methods

```java
while (st.hasMoreTokens()) {
    String token = st.nextToken();
    System.out.println(token);
}
```

int count = st.countTokens(); // Number of remaining tokens

```

Note: For new code, prefer `String.split()` or `Scanner` class.

## StringJoiner Class (Java 8+)

**StringJoiner** is used to construct a sequence of characters separated by a delimiter.

### Creating StringJoiner

```java
StringJoiner sj = new StringJoiner(","); // Comma delimiter
StringJoiner sj2 = new StringJoiner(",", "[", "]"); // With prefix and suffix
```

Methods

```java
sj.add("Apple");
sj.add("Banana");
sj.add("Orange");
String result = sj.toString(); // "Apple,Banana,Orange"

// With prefix/suffix: "[Apple,Banana,Orange]"
```
```

String vs StringBuilder vs StringBuffer

Feature	String	StringBuilder	StringBuffer
Mutability	Immutable	Mutable	Mutable
Thread Safety	Yes	No	Yes
Performance	Fast	Very Fast	Fast
Storage	String Pool	Heap	Heap
When to Use	Constants	Single-thread	Multi-thread

Best Practices

1. Use String for constants and when immutability is desired
2. Use StringBuilder for string manipulation in single-threaded environments
3. Use StringBuffer for string manipulation in multi-threaded environments
4. For simple concatenation, + operator is fine (compiler may optimize to StringBuilder)
5. For complex or looped concatenation, always use StringBuilder
6. Prefer String.split() over StringTokenizer for new code
7. Use StringJoiner for joining strings with delimiters (Java 8+)

Performance Considerations

- String concatenation in loops creates many temporary objects
- StringBuilder/Buffer initial capacity should be set if final size is known
- String methods like substring(), toUpperCase() etc. create new objects

- String literals are reused from string pool, but new String() creates new objects

Java 11 Additions

```java
// isBlank() - checks if string is empty or only whitespace
" ".isBlank(); // true

// lines() - returns stream of lines
"line1\nline2".lines().count(); // 2

// repeat() - repeats string
"abc".repeat(2); // "abcabc"

// strip(), stripLeading(), stripTrailing() - better than trim()
" hello ".strip(); // "hello"
```

This comprehensive guide covers all major aspects of Strings and related classes in Java. The choice between these classes depends on your specific requirements regarding mutability, thread safety, and performance.

 # **Strings in Java: DSA Perspective**

****(Optimized for Notes - Clear Formatting)****

1️⃣ Basic String Operations

1.1 Iterating Over a String

```java
// METHOD 1: Standard for-loop
String s = "hello";
for (int i = 0; i < s.length(); i++) {
    char c = s.charAt(i);
    System.out.print(c + " ");  // Output: h e l l o
}

// METHOD 2: Enhanced for-loop (char array)
for (char c : s.toCharArray()) {
    System.out.print(c + " ");  // Output: h e l l o
}
```

2️⃣ Recursion with Strings

2.1 Reverse a String (Recursive)

```java
public static String reverse(String s) {
    if (s.isEmpty()) return s;  // Base case
    return reverse(s.substring(1)) + s.charAt(0);  // Recursive call
}
```

➤ Logic Breakdown:

1. Breaks problem into smaller subproblems

2. `"hello" → reverse("ello") + "h" → reverse("llo") + "e" + "h" → ... → "olleh"`

2.2 Check Palindrome (Recursive)

```java
public static boolean isPalindrome(String s, int left, int right) {
    if (left >= right) return true;  // Base case
    if (s.charAt(left) != s.charAt(right)) return false;  // Mismatch found
    return isPalindrome(s, left + 1, right - 1);  // Move pointers inward
}
```

➤ Usage Example:

```java
System.out.println(isPalindrome("madam", 0, 4));  // true
System.out.println(isPalindrome("racecar", 0, 6)); // true
```

3️⃣ Essential DSA Problems

3.1 First Unique Character

```java
public static int firstUniqChar(String s) {
    int[] freq = new int[26];  // Assuming lowercase English letters
    for (char c : s.toCharArray()) freq[c - 'a']++;  // Build frequency map
```

```
    for (int i = 0; i < s.length(); i++) {
        if (freq[s.charAt(i) - 'a'] == 1) return i;  // Found unique
    }
    return -1;  // No unique characters
}
```

➤ Key Points:

- Time Complexity: **O(n)**
- Space Complexity: **O(1)** (fixed-size array)

3.2 Longest Substring Without Repeats

```java
public static int lengthOfLongestSubstring(String s) {
    Set<Character> window = new HashSet<>();
    int max = 0, left = 0;

    for (int right = 0; right < s.length(); right++) {
        while (window.contains(s.charAt(right))) {  // Remove duplicates
            window.remove(s.charAt(left++));  // Shrink window from left
        }
        window.add(s.charAt(right));  // Expand window
        max = Math.max(max, right - left + 1);  // Update max length
    }
    return max;
}
```

```

**➤ Visualization:**

```

Input: "abcabcbb"

Window States:

[a] → [a,b] → [a,b,c] → [b,c,a] → [c,a,b] → [a,b,c] → [b] → [b]

Max Length: 3

```

---

## **4️⃣ StringBuilder Techniques**

### **4.1 Efficient String Building**

```java
// ❌ Inefficient (creates new String objects)
String result = "";
for (int i = 0; i < 1000; i++) {
    result += i;  // O(n²) time!
}

// ✅ Optimized (mutable buffer)
StringBuilder sb = new StringBuilder();
for (int i = 0; i < 1000; i++) {
    sb.append(i);  // O(n) time
}
String output = sb.toString();
```
```

```

### **4.2 Reverse Words in String**

```java
public static String reverseWords(String s) {
    String[] words = s.trim().split("\\s+");  // Split by whitespace
    StringBuilder sb = new StringBuilder();

    for (int i = words.length - 1; i >= 0; i--) {
        sb.append(words[i]).append(" ");  // Append in reverse order
    }
    return sb.toString().trim();  // Remove trailing space
}
```

➤ Example:

Input: `"the sky is blue"` → Output: `"blue is sky the"`

5️⃣ Advanced Problems

5.1 String Permutations (Backtracking)

```java
public static List<String> generatePermutations(String s) {
    List<String> result = new ArrayList<>();
    backtrack(s.toCharArray(), 0, result);
    return result;
```
```

```
}

private static void backtrack(char[] arr, int index, List<String> result) {
    if (index == arr.length) {
        result.add(new String(arr));
        return;
    }
    for (int i = index; i < arr.length; i++) {
        swap(arr, index, i);  // Choose
        backtrack(arr, index + 1, result);  // Explore
        swap(arr, index, i);  // Unchoose (backtrack)
    }
}
```

➤ Recursion Tree:

```
          abc
       /   |   \
   a(bc)  b(ac)  c(ba)
    / \    / \    / \
  ab(c) ac(b) ba(c) bc(a) ...
```

6️⃣ Quick Reference Snippets

6.1 Count Vowels/Consonants

```java
String input = "Hello World";
long vowels = input.toLowerCase().chars()
        .filter(c -> "aeiou".indexOf(c) != -1)
        .count();
```

6.2 Check Anagram

```java
public static boolean isAnagram(String s1, String s2) {
   if (s1.length() != s2.length()) return false;
   int[] counts = new int[26];
   for (char c : s1.toCharArray()) counts[c - 'a']++;
   for (char c : s2.toCharArray()) if (--counts[c - 'a'] < 0) return false;
   return true;
}
```

📝 Summary Table

Concept	Time Complexity	Key Method
Iteration	O(n)	`charAt()`/`toCharArray()`
Recursion	O(n)	Substring decomposition
Sliding Window	O(n)	Two pointers + HashSet

| StringBuilder | O(n) | `append()` for concatenation |

****🔗 Recommended Next Steps:****

- Practice on LeetCode (Tag: String)

- Implement KMP for pattern matching

- Explore Trie data structure for dictionary problems

****Formatted for perfect copy-pasting into notes!**** 📑✨

Pretty Printing & Format Specifiers in Java

(Concise Notes with Single-Line Examples)

📌 String Formatting Methods

1. `String.format()`

```java
String formatted = String.format("Name: %s, Age: %d", "Alice", 25);
// Output: "Name: Alice, Age: 25"
```

2. `System.out.printf()`

```java
System.out.printf("Price: $%.2f", 19.99);
// Output: "Price: $19.99" (prints directly)
```

3. `Formatter` Class

```java
new Formatter().format("Hex: %x", 255).toString();  // "Hex: ff"
```

🔢 Common Format Specifiers

Specifier	Type	Example	Output
`%s`	String	`"%s".formatted("Java")`	`"Java"`
`%d`	Integer	`String.format("%04d", 42)`	`"0042"` (4 digits)
`%f`	Float/Double	`"%.1f".formatted(3.14159)`	`"3.1"` (1 decimal)
`%x`	Hexadecimal	`"%x".formatted(255)`	`"ff"`
`%c`	Character	`"%c".formatted('A')`	`"A"`
`%b`	Boolean	`"%b".formatted(true)`	`"true"`
`%n`	Newline	`"Line1%nLine2"`	`"Line1\nLine2"`

🎨 Pretty Printing Techniques

1. Column Alignment

```java
System.out.printf("%-10s %5d%n", "Apple", 50);  // Left-align string, right-align number
// Output: "Apple       50"
```

```

### **2. Number Formatting**

```java
String.format("%,d", 1000000);  // "1,000,000" (thousand separator)
```

3. Date Formatting

```java
String.format("%tF", new Date());  // "2023-11-15" (ISO date)
```

4. Scientific Notation

```java
"%.2e".formatted(1234567.89);  // "1.23e+06"
```

💡 Pro Tips

1. **Precision Control**
   ```java
   "%.3f".formatted(Math.PI);  // "3.142" (3 decimal places)
   ```

2. **Padding with Zeros**
   ```

```java
"%05d".formatted(42);  // "00042"
```

3. **Argument Indexing**

```java
String.format("%2$s %1$s", "World", "Hello");  // "Hello World"
```

4. **Format Flags**

```java
"%+d".formatted(25);   // "+25" (force sign)
"%,.2f".formatted(1234.5);  // "1,234.50" (comma + 2 decimals)
```

📝 Summary Cheat Sheet

```java
// BASIC TEMPLATE:
String result = String.format("Format %s %d %.2f", text, number, decimal);

// COMMON USES:
String tableRow = String.format("%-15s | %4d | %6.2f", "Item", 100, 19.99);
String hexValue = String.format("0x%08X", 255);  // "0x000000FF"
```

Formatted for easy copy-pasting into notes! ✨

Comprehensive String Interview Questions & Answers

1. Basic String Concepts

Q1: What is a String in Java?

A: In Java, a String is an immutable sequence of Unicode characters represented by the `String` class. It's stored in the String Pool (a special memory area in the heap) for reusability.

Q2: Why are Strings immutable in Java?

A: Strings are immutable because:

- **Security:** Prevents modification in sensitive operations (e.g., database connections).

- **Thread Safety:** Can be shared across threads without synchronization.

- **Hash Caching:** Hashcode can be cached since the content won’t change.

- **Memory Efficiency:** Enables String Pool optimization.

Q3: What is the String Pool?

A: The String Pool is a special area in the heap where Java stores unique String literals to save memory.

Example:

```java
String s1 = "Hello";  // Goes to String Pool

String s2 = new String("Hello");  // Creates a new object in heap

System.out.println(s1 == s2);  // false (different references)
```

```

### **Q4: What is `intern()` method?**

**A:** `intern()` adds a String to the pool (if not already present) and returns its reference.

**Example:**

```java
String s3 = new String("Hello").intern();
System.out.println(s1 == s3);  // true (both point to pool)
```

---

2. String Manipulation & Operations

Q5: How to reverse a String?

Method 1: Using `StringBuilder`

```java
String reversed = new StringBuilder("Hello").reverse().toString();  // "olleH"
```

Method 2: Using Recursion

```java
public String reverse(String s) {
   return s.isEmpty() ? s : reverse(s.substring(1)) + s.charAt(0);
}
```
```

Q6: Check if a String is a Palindrome

```java
boolean isPalindrome(String s) {
   int left = 0, right = s.length() - 1;
   while (left < right) {
      if (s.charAt(left++) != s.charAt(right--)) return false;
   }
   return true;
}
```

Q7: Find the first non-repeating character

```java
int firstUniqChar(String s) {
   int[] freq = new int[26];
   for (char c : s.toCharArray()) freq[c - 'a']++;
   for (int i = 0; i < s.length(); i++) {
      if (freq[s.charAt(i) - 'a'] == 1) return i;
   }
   return -1;
}
```

---

3. String Performance & Optimization

Q8: Why use `StringBuilder` over `String` for concatenation?

A:

- **String:** Immutable → Each concatenation creates a new object (O(n²) time).

- **StringBuilder:** Mutable → Modifies internal buffer (O(n) time).

Example:

```java
// ❌ Inefficient
String result = "";
for (int i = 0; i < 1000; i++) result += i;

// ✅ Efficient
StringBuilder sb = new StringBuilder();
for (int i = 0; i < 1000; i++) sb.append(i);
String output = sb.toString();
```

Q9: Difference between `StringBuilder` and `StringBuffer`?

StringBuilder	**StringBuffer**
Not thread-safe	Thread-safe (synchronized)
Faster	Slower due to synchronization
Java 5+	Java 1.0

---

4. Advanced String Problems

Q10: Longest Substring Without Repeating Characters

```java
int longestSubstring(String s) {
   Set<Character> set = new HashSet<>();
   int max = 0, left = 0;
   for (int right = 0; right < s.length(); right++) {
      while (set.contains(s.charAt(right))) {
         set.remove(s.charAt(left++));  // Shrink window
      }
      set.add(s.charAt(right));
      max = Math.max(max, right - left + 1);
   }
   return max;
}
```

Q11: Check if two Strings are Anagrams

```java
boolean isAnagram(String s1, String s2) {
   if (s1.length() != s2.length()) return false;
   int[] counts = new int[26];
   for (char c : s1.toCharArray()) counts[c - 'a']++;
```

```
    for (char c : s2.toCharArray()) {
        if (--counts[c - 'a'] < 0) return false;
    }
    return true;
}
```

Q12: Implement `strStr()` (Needle in Haystack)

```java
int strStr(String haystack, String needle) {
    if (needle.isEmpty()) return 0;
    for (int i = 0; i <= haystack.length() - needle.length(); i++) {
        if (haystack.substring(i, i + needle.length()).equals(needle)) {
            return i;
        }
    }
    return -1;
}
```

---

5. Tricky String Questions

Q13: How to count vowels and consonants?

```java
```

```
long vowels = "Hello".chars().filter(c -> "aeiou".indexOf(c) != -1).count();  // 2
```

Q14: Reverse words in a String

```java
String reverseWords(String s) {
   String[] words = s.trim().split("\\s+");
   return IntStream.range(0, words.length)
                .mapToObj(i -> words[words.length - 1 - i])
                .collect(Collectors.joining(" "));
}
```

Q15: What is the output of `"Hello" == new String("Hello")`?

A: `false` because `==` checks references, and `new String()` creates a new object in the heap.

---

6. Real-World Scenarios

Q16: How to parse a CSV String?

```java
String csv = "Java,Python,C++";
String[] languages = csv.split(",");  // ["Java", "Python", "C++"]
```

Q17: How to format a phone number?

```java
String phone = "1234567890";
String formatted = String.format("(%s) %s-%s",
   phone.substring(0, 3),
   phone.substring(3, 6),
   phone.substring(6));  // (123) 456-7890
```

Summary Table

Topic	**Key Methods/Concepts**
Immutability	`String`, `intern()`, String Pool
Reversal/Palindrome	`StringBuilder.reverse()`, Two Pointers
Substring Problems	Sliding Window, `Set`/`Map`
Concatenation	`StringBuilder` vs `+`
Formatting	`String.format()`, `printf()`

🔥 Pro Tip: Memorize time complexities:

- `charAt()`, `length()` → O(1)
- `substring()` → O(n) (pre-Java 7u6), O(1) (later versions)
- `equals()` → O(n) (char-by-char comparison)

Practice these on LeetCode for mastery! 🚀