String vs StringBuilder vs StringBuffer in Java

A **string** is a sequence of characters. In Java, objects of String are immutable which means a constant and cannot be changed once created. In Java, String, **StringBuilder**, and **StringBuffer** are used for handling strings. The main difference is:

- **String:** Immutable, meaning its value cannot be changed once created. It is thread-safe but less memory-efficient.
- StringBuilder: Mutable, not thread-safe, and more memory-efficient compared to String. Best used for single-threaded operations.
- StringBuffer: Mutable and thread-safe due to synchronization, but less efficient than StringBuilder in terms of performance.

1. String

A String in Java is an **immutable** object. Once created, its value cannot be changed. If you modify a String, a new object is created in memory.

Ways to Create Strings:

Using String Literal:

When a String is created using a **literal**, it is stored in the **String Pool**. If a string with the same content already exists, it reuses that instance.

```
String s1 = "Hello"; // Stored in the String Pool

String s2 = "Hello"; // References the same object as s1(// Reuses the same "Hello" from String Pool)


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

Using new Keyword:

When a String is created using the new keyword, it creates a **new object in the heap**, even if the same content exists in the String Pool.

```
String str1 = new String("Hello");  // Creates a new object in the heap

String str2 = new String("Hello");  // Another new object in Heap

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

String Example:

```
String s1 = "Hello";  // Literal
s1 = s1.concat(" World"); // New object is created in String Pool
System.out.println(s1); // Output: "Hello World"

String s2 = new String("Hello"); // Using new keyword
s2 = s2.toUpperCase(); // Creates a new object in Heap
System.out.println(s2); // Output: "HELLO"
```

Methods in String:

- **length()**: Returns the length of the string.
- **charAt(int index)**: Returns the character at the specified index.
- **substring(int start, int end)**: Returns a substring.
- **concat(String s)**: Concatenates two strings.
- **toUpperCase()** and **toLowerCase()**: Converts to upper/lower case.
- **equals(Object o)**: Compares the string content.
- **trim()**: Removes leading and trailing spaces.

Example:

```
public class StringExample {

    public static void main(String[] args) {
```

```
        String s1 = "Java";            // String literal

        String s2 = "Java";            // Points to the same object as s1

        String s3 = new String("Java"); // Creates a new object



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

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

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



        String s4 = s1.concat(" Programming");

        System.out.println(s4);        // "Java Programming"

        System.out.println(s1);        // "Java" (original string remains unchanged)

    }

}
```

2. StringBuffer

A StringBuffer is **mutable** (can be modified after creation) and **thread-safe** (synchronized). It is slower compared to StringBuilder due to synchronization.

Ways to Create StringBuffer:

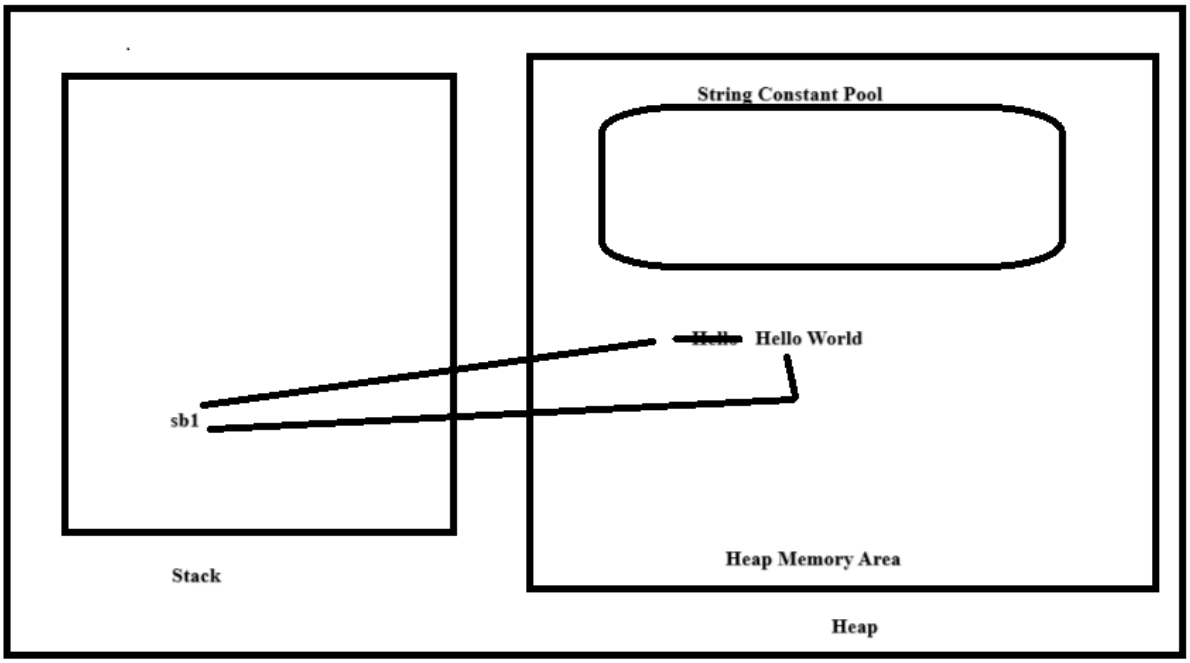
Unlike String, StringBuffer is mutable. It does not reuse instances or go into the String Pool. Whether created using literals or the new keyword, StringBuffer objects reside in heap memory and can be modified.

Using Constructor:

```
StringBuffer sb1 = new StringBuffer("Hello");  // Mutable object in Heap
sb1.append(" World");  // Modifies the same object
System.out.println(sb1); // Output: "Hello World"
```

Default Constructor:

```
StringBuffer sb2 = new StringBuffer();  // Empty buffer with default capacity 16
sb2.append("Hello");
System.out.println(sb2); // Output: "Hello"
```

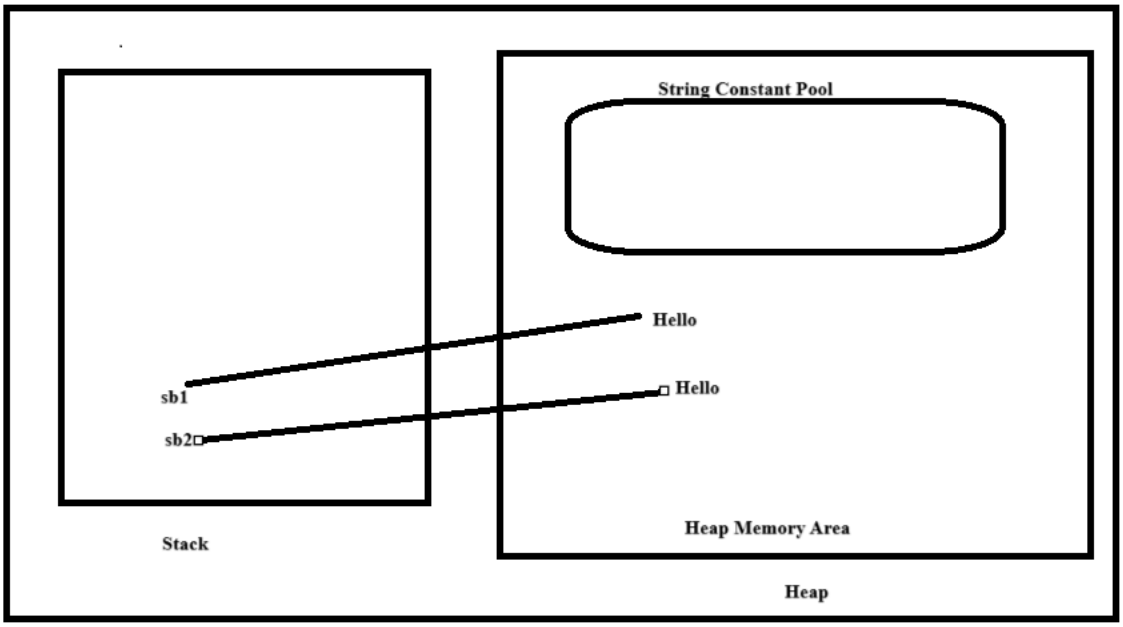


Important Example:

1. == **operator**:
    - The == operator compares **references** (memory locations), not the contents of the objects.
    - If you use == to compare two StringBuffer objects, it will return true only if both objects refer to the exact same memory location (i.e., the same object).

      Example:

      StringBuffer sb1 = new StringBuffer("hello");
      StringBuffer sb2 = new StringBuffer("hello");
      System.out.println(sb1 == sb2); // false, because they are two different objects



2. **.equals() method**:
    - The .equals() method is inherited from Object and, by default, compares **references** (like ==), unless it is overridden.
    - StringBuffer does **not** override the .equals() method, so it behaves like == for reference comparison.
    - If you want to compare the contents of two StringBuffer objects, you can use .toString() to convert them into String objects and then use .equals().

      Example:
      StringBuffer sb1 = new StringBuffer("hello");
      StringBuffer sb2 = new StringBuffer("hello");

```
System.out.println(sb1.toString().equals(sb2.toString())); // true, because the content is the same
```

In summary:

- Use == if you want to check if two StringBuffer objects refer to the same object.
- Use .toString().equals() if you want to compare the content of the StringBuffer objects.

Methods in StringBuffer:

- **append(String s)**: Appends the string to the buffer.
- **insert(int offset, String s)**: Inserts a string at the specified position.
- **replace(int start, int end, String s)**: Replaces characters in a range.
- **delete(int start, int end)**: Deletes characters in a range.
- **reverse()**: Reverses the characters in the buffer.

Example:

```
public class StringBufferExample {

    public static void main(String[] args) {

        StringBuffer sb = new StringBuffer("Hello");



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

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



        sb.insert(6, "Java "); // Insert
```

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



        sb.replace(6, 10, "Python"); // Replace

        System.out.println(sb); // "Hello Python World"



        sb.reverse(); // Reverse

        System.out.println(sb); // "dlroW nohtyP olleH"

    }

}
```

3. StringBuilder

A StringBuilder is similar to StringBuffer, but it is **not thread-safe** (unsynchronized). It is faster compared to StringBuffer when used in a single-threaded environment.

Ways to Create StringBuilder:

Similar to StringBuffer, StringBuilder is mutable but **not synchronized** (not thread-safe). It is faster than StringBuffer in a single-threaded environment.

Using Constructor:

```
StringBuilder sb1 = new StringBuilder("Hello");


sb1.append(" World");  // Modifies the same object
System.out.println(sb1); // Output: "Hello World"
```

Default Constructor:

```
StringBuilder sb2 = new StringBuilder(); // Empty buffer with default capacity 16


sb2.append("Java");
System.out.println(sb2); // Output: "Java"
```

Important Example:

The behavior of .equals() and == in StringBuilder is very similar to StringBuffer in Java:

1. **== operator**:
    - The == operator compares **references** (memory locations), not the content.
    - If you use == to compare two StringBuilder objects, it will return true only if both objects refer to the exact same memory location (i.e., the same object).

      Example:
      ```
      StringBuilder sb1 = new StringBuilder("hello");
      StringBuilder sb2 = new StringBuilder("hello");
      System.out.println(sb1 == sb2); // false, because they are two different objects
      ```

2. **.equals() method**:
    - Similar to StringBuffer, StringBuilder does **not override** the .equals() method, so it behaves like == for reference comparison.
    - To compare the contents of two StringBuilder objects, you can convert them to String using .toString() and then use .equals().

      Example:
      ```
      StringBuilder sb1 = new StringBuilder("hello");
      StringBuilder sb2 = new StringBuilder("hello");
      System.out.println(sb1.toString().equals(sb2.toString())); // true, because the content is the same
      ```

In summary:

- Use == to check if two StringBuilder objects refer to the same memory location.
- Use .toString().equals() to compare the content of two StringBuilder objects.

Methods in StringBuilder:

Both StringBuilder and StringBuffer have the same core methods for string manipulation because they are part of the same class hierarchy. The primary difference between them is that StringBuffer is synchronized, making it thread-safe, while StringBuilder is not.

The methods are identical to those in StringBuffer, such as:

- append()
- insert()
- replace()
- delete()
- reverse()

Example:

Here's an example demonstrating how to use some common methods from both StringBuilder and StringBuffer in one code snippet:

```
public class StringBuilderStringBufferExample {

    public static void main(String[] args) {

        // Using StringBuilder (non-synchronized, faster in single-threaded scenarios)

        StringBuilder sb = new StringBuilder("Hello");

        // Append

        sb.append(" World");

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

```
// Insert

sb.insert(6, "Java ");

System.out.println("After insert: " + sb);  // Output: Hello Java World



// Delete

sb.delete(6, 11);

System.out.println("After delete: " + sb);  // Output: Hello World



// Set character at index

sb.setCharAt(6, 'J');

System.out.println("After setCharAt: " + sb);  // Output: Hello Jorld



// Reverse

sb.reverse();

System.out.println("After reverse: " + sb);  // Output: droJ olleH
```

```
// Convert to string

String str = sb.toString();

System.out.println("ToString: " + str);  // Output: droJ olleH



// Using StringBuffer (synchronized, thread-safe)

StringBuffer sbf = new StringBuffer("Welcome");



// Append

sbf.append(" to Java");

System.out.println("After append: " + sbf);  // Output: Welcome to Java



// Insert

sbf.insert(8, "Programming ");

System.out.println("After insert: " + sbf);  // Output: Welcome Programming to Java



// Delete
```

```
        sbf.delete(8, 21);
        System.out.println("After delete: " + sbf);  // Output: Welcome to Java

        // Set character at index
        sbf.setCharAt(11, 'J');
        System.out.println("After setCharAt: " + sbf);  // Output: Welcome to Java

        // Reverse
        sbf.reverse();
        System.out.println("After reverse: " + sbf);  // Output: avaJ ot emocleW
    }
}
```

—----------

Methods in StringBuilder and StringBuffer:

1. **append()**:

- Adds a string (or other types) to the end of the current StringBuilder or StringBuffer.
- Example: sb.append(" World");

2. **insert()**:
    - Inserts a string (or other types) at a specified index.
    - Example: sb.insert(5, " Java");
3. **delete()**:
    - Removes a substring from the current StringBuilder or StringBuffer.
    - Example: sb.delete(5, 10);
4. **deleteCharAt()**:
    - Removes the character at a specific index.
    - Example: sb.deleteCharAt(4);
5. **reverse()**:
    - Reverses the content of the StringBuilder or StringBuffer.
    - Example: sb.reverse();
6. **toString()**:
    - Converts the StringBuilder or StringBuffer to a String.
    - Example: String str = sb.toString();
7. **capacity()**:
    - Returns the current capacity (the amount of storage) of the StringBuilder or StringBuffer.
    - Example: int cap = sb.capacity();
8. **length()**:
    - Returns the current length (number of characters) in the StringBuilder or StringBuffer.
    - Example: int len = sb.length();
9. **charAt()**:
    - Returns the character at a specified index.
    - Example: char c = sb.charAt(2);
10. **setCharAt()**:

- Sets the character at a specific index.
- Example: sb.setCharAt(3, 'X');

11. **ensureCapacity()**:
    - Ensures that the StringBuilder or StringBuffer has at least the specified capacity.
    - Example: sb.ensureCapacity(50);
12. **substring()**:
    - Extracts a substring starting from a specified index or between two indices.
    - Example: String substr = sb.substring(5, 10);
13. **indexOf()**:
    - Returns the index of the first occurrence of a specified substring.
    - Example: int index = sb.indexOf("Java");
14. **lastIndexOf()**:
    - Returns the index of the last occurrence of a specified substring.
    - Example: int index = sb.lastIndexOf("World");

How would you differentiate between a String, StringBuffer, and a StringBuilder?

1) Storage area : In string, the String pool serves as the storage area. For StringBuilder and StringBuffer, heap memory is the storage area.

2) Mutability : A String is immutable, whereas both the StringBuilder and StringBuffer are mutable.

3) Efficiency : It is quite slow to work with a String. However, StringBuilder is the fastest in performing operations. The speed of a StringBuffer is more than a String and less than a StringBuilder. (For example appending a character is fastest in StringBuilder and very slow in String because a new memory is required for the new String with appended character.)

4) Thread-safe : In the case of a threaded environment, StringBuilder and StringBuffer are used whereas a String is not used. However, StringBuilder is suitable for an environment with a single thread, and a StringBuffer is suitable for multiple threads.

Which among String or String Buffer should be preferred when there are a lot of updates required to be done in the data?

StringBuffer is **mutable** and **dynamic** in nature, allowing direct modification of its content without creating new objects. On the other hand, String is **immutable**, meaning any update or modification results in the creation of a **new String object**, which leads to:

- **Increased memory usage**
- **Overhead in the String Constant Pool**
- **Reduced performance** during heavy string manipulations

Due to this, when a program involves **multiple updates or modifications** to string data (like inside loops or iterative processing), it's always preferred to use **StringBuffer** (or StringBuilder in single-threaded environments). This avoids the overhead of repeated object creation and significantly **improves performance and memory efficiency**.