ShreeSoft INFORMATICS

Address: Tirumala Plaza, Upendra Nagar, Cidco, Nashik.

Join us on Telegram:
https://t.me/ShreeSoftinformatics

Join us on WhatsApp:
https://chat.whatsapp.com/BrLM0ZoWMbpA8nXi9Mrocl

Contact: **+91 8308341531**

String in Java

What is String?

- A String is sequence of character which is enclosed within double quote.
- In C / C++, String will be implemented by using array of character.
- In java, String is an object of String class that represents a string of characters.
- String class is used to create a string object.
- It is a predefined immutable class that is present in **java.lang package.**
- But in Java, all classes are also considered as a data type. So, we can also consider a string as a data type.
- Immutable means it cannot be changed.
- There are three main classes by which we can create String:
  - ❖ String
  - ❖ StringBuffer
  - ❖ StringBuilder

String class:

We can create String object in 3 ways in java as

1. **By string literal.**
2. **By new keyword.**
3. **By converting character arrays into strings**

The following 3 areas where String values are stored by JVM.

1. String constant pool(SCP)
2. Heap Memory
3. Stack Memory

What is String Constant Pool?

❒ The String Constant pool is also called as String Literal Pool.

❒ The string literal is always created in the string constant pool. In Java, String constant pool is a special area that is used for storing string objects.

❒ Internally, String class uses a string constant pool (SCP). This SCP area is part of the method area (Permanent Generation Method area) until Java 1.6 version.

❒ From Java 1.7 onwards, SCP area is moved in the heap memory because SCP is a fixed size in the method area but in the heap memory, SCP can be expandable.

- Therefore, the string constant pool has been moved to heap area **for memory utilization only**.
- Later, in Java 8, Permanent Generation has been completely removed.
- **So in the latest JVMs, the String pool is a special area in heap memory allocated for storing the String literals.**

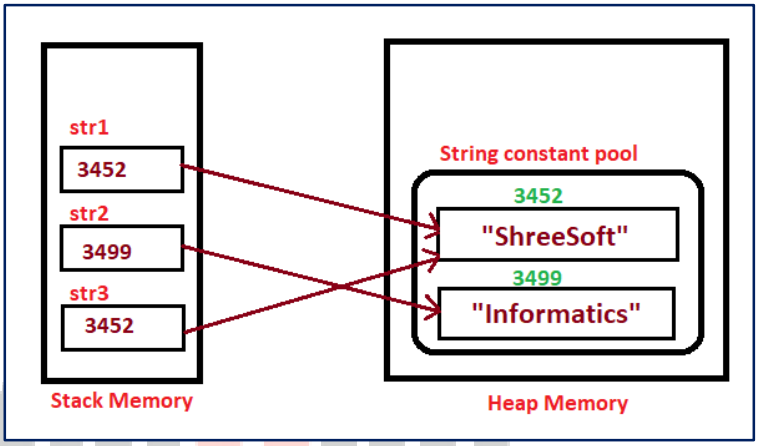
1) Creating String by using Literal:

Whenever we create a string literal in Java, JVM checks string constant pool first. If the string already exists in string constant pool, no new string object will be created in the string pool by JVM.

JVM uses the same string object by pointing a reference to it to save memory. But if string does not exist in the string pool, JVM creates a new string object and placed it in the pool.

Example:

String str1 = “ShreeSoft”;
String str2 = “Informatics”;
String str2 = “ShreeSoft”;



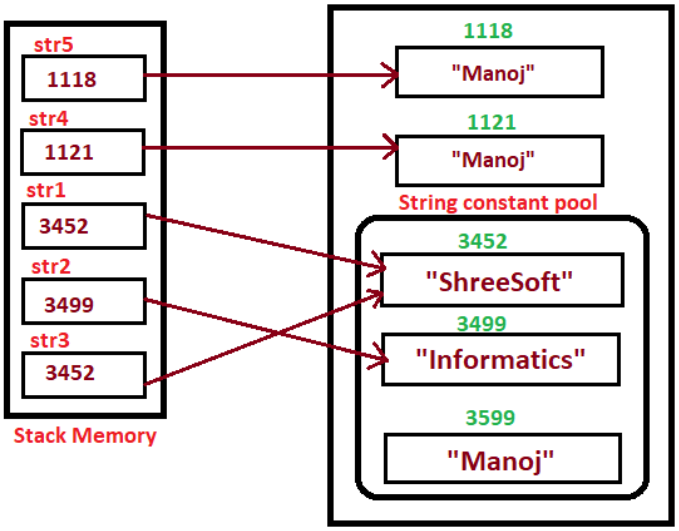

```
public class StringExample1 {
        public static void main(String[] args) {
                String str1 = "ShreeSoft";
                String str2 = "Informatics";
                String str3 = "ShreeSoft";

                //comparing str1 and str2
                if(str1==str2)
                System.out.println("Str1 and Str2 strings are equal");
                else
                System.out.println("Str1 and Str2 strings are not equal");

                //comparing str1 and str3
                if(str1==str3)
                System.out.println("Str1 and Str2 strings are equal");
                else
                System.out.println("Str1 and Str2 strings are not equal");
        }
}
```

2) Creating String by using new keyword:

When we create a String via the new operator, the Java compiler will create a new object and store it in the heap space reserved for the JVM.

Every String created like this will point to a different memory region with its own address.

Example:
String str4 = new String(“Manoj”);
String str5 = new String(“Manoj”);



Program:

```
public class StringExample2 {
	public static void main(String[] args) {
		String str1 = "ShreeSoft";
		String str2 = "Informatics";
		String str3 = "ShreeSoft";
		String str4 = new String("Manoj");
		String str5 = new String("Manoj");
		if(str1==str3)
			System.out.println("Str1 and str3 are equal");
		else
			System.out.println("Str1 and str3 are not equal");

		if(str4==str5)
			System.out.println("Str4 and str5 are equal");
		else
			System.out.println("Str4 and str5 are not equal");
	}
}
```


What is String Interning?

As we know String is immutable in Java, the JVM can optimize the amount of memory allocated for them by storing only one copy of each literal String in the pool. This process is called interning.

When we create a String variable and assign a value to it, the JVM searches the pool for a String of equal value.

If found, the Java compiler will simply return a reference to its memory address, without allocating additional memory.

If not found, it’ll be added to the pool (interned) and its reference will be returned.

How to manually intern the String?

We can manually intern a String in the Java String Constant Pool by calling the **intern()** method on the object we want to intern.

Manually interning the String will store its reference in the pool, and the JVM will return this reference when needed.

Example:

```
public class StringInternExample {
	public static void main(String[] args) {
		String str1 = "ShreeSoft";
		String str2 = new String("ShreeSoft");

		System.out.println("Before String Interning");

		if(str1==str2)
			System.out.println("Str1 and Str2 strings are equal");
		else
			System.out.println("Str1 and Str2 strings are not equal");

		str2 = str2.intern();
		System.out.println("\nAfter String Interning");
		if(str1==str2)
			System.out.println("Str1 and Str2 strings are equal");
		else
			System.out.println("Str1 and Str2 strings are not equal");
	}
}
```


3) Creating String By converting character arrays into strings:

We can also create string using character arrays as

Example 1:

```
	char arr[]= {'S','h','r','e','e','s','o','f','t'};
	String str = new String(arr);
	System.out.println("Str:"+str);
```

Output:

Shreesoft

Example 2:

```
	char arr[]= {'S','h','r','e','e','s','o','f','t'};
	String str = new String(arr, 5,4);
	System.out.println("Str:"+str);
```

Output:

soft

How many total objects will be created in memory for following string objects?

```
String s1 = new String("Manoj");
String s2 = new String("Manoj");

String s3 = "Manoj";
String s4 = "Manoj";
```

1. During the execution of first statement using new operator, JVM will create two objects, one with content in heap area and another as a copy for literal "Manoj" in the SCP area for future purposes as shown in the figure.
2. The reference variable s1 is pointing to the first object in the heap area.

3. When the second statement will be executed, for every new operation, JVM will create again one new object with content "Manoj" in the heap area.

4. But in the SCP area, no new object for literal will be created by JVM because it is already available in the SCP area. The s2 is pointing to the object in the heap area as shown in the figure.

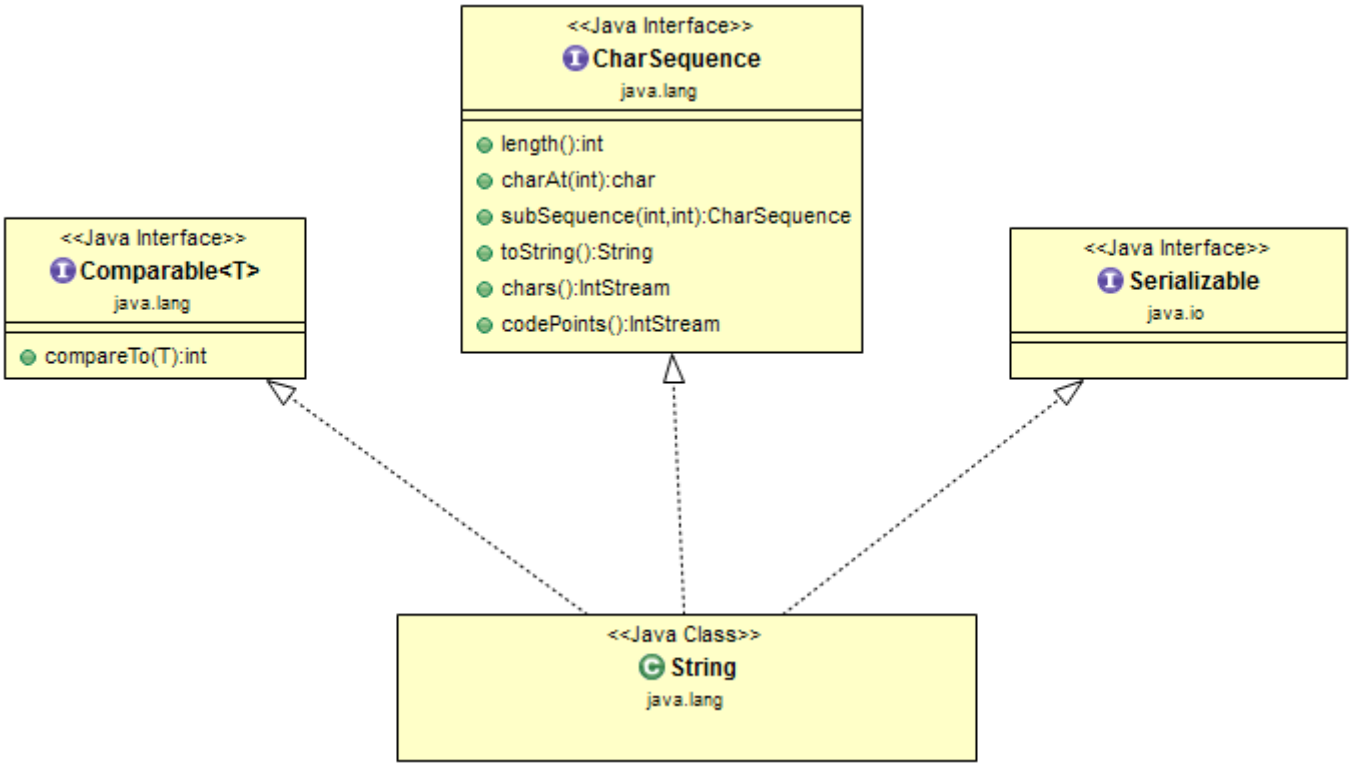
5. During the execution of third and fourth statements, JVM will not create a new object with content "Manoj" in the SCP area because it is already available in string constant pool.

It simply points the reference variables s3 and s4 to the same object in the SCP. They are shown in the above figure.

Thus, three objects are created, two in the heap area and one in string constant pool. This kind of question is always asked in the interview or technical tests in the different companies. So, you must prepare answers for this kind of question.

String Class Hierarchy in Java

String class implemented interfaces: Serializable, CharSequence, Comparable< String >



String Class Constructors:

- **String()** - Initializes a newly created String object so that it represents an empty character sequence.

- **String(byte[] bytes)** - Constructs a new String by decoding the specified array of bytes using the platform's default charset.
- **String(byte[] bytes, Charset charset)** - Constructs a new String by decoding the specified array of bytes using the specified charset.

- **String(byte[] bytes, int offset, int length)** - Constructs a new String by decoding the specified subarray of bytes using the platform's default charset.

- **String(byte[] bytes, int offset, int length, Charset charset)** - Constructs a new String by decoding the specified subarray of bytes using the specified charset.

- **String(byte[] bytes, int offset, int length, String charsetName)** - Constructs a new String by decoding the specified subarray of bytes using the specified charset.

- **String(byte[] bytes, String charsetName)** - Constructs a new String by decoding the specified array of bytes using the specified charset.

- **String(char[] value)** - Allocates a new String so that it represents the sequence of characters currently contained in the character array argument.

- **String(char[] value, int offset, int count)** - Allocates a new String that contains characters from a subarray of the character array argument.

- **String(int[] codePoints, int offset, int count)** - Allocates a new String that contains characters from a subarray of the Unicode code point array argument.

- **String(String original)** - Initializes a newly created String object so that it represents the same sequence of characters as the argument; in other words, the newly created string is a copy of the argument string.

- **String(StringBuffer buffer)** - Allocates a new string that contains the sequence of characters currently contained in the string buffer argument.

- **String(StringBuilder builder)** - Allocates a new string that contains the sequence of characters currently contained in the string builder argument.

String Class Methods:

1) **char charAt**(int index): Returns the char value at the specified index.

2) **int codePointAt**(int index): Returns the character (Unicode code point) at the specified index.

3) **int codePointBefore**(int index): Returns the character (Unicode code point) before the specified index.

4) **int codePointCount**(int beginIndex, int endIndex): Returns the number of Unicode code points in the specified text range of this String.

5) **int compareTo**(**String** anotherString): Compares two strings lexicographically.

6) **int compareToIgnoreCase**(**String** str): Compares two strings lexicographically, ignoring case differences.

7) **String concat**(**String** str): Concatenates the specified string to the end of this string.

8) **boolean contains**(**CharSequence** s): Returns true if and only if this string contains the specified sequence of char values.

9) **boolean contentEquals**(**CharSequence** cs): Compares this string to the specified CharSequence.

10) **boolean contentEquals**(**StringBuffer** sb): Compares this string to the specified StringBuffer.

11) **static String copyValueOf**(char[] data): Equivalent to **valueOf(char[])**.

12) **static String copyValueOf**(char[] data, int offset, int count): Equivalent to **valueOf(char[], int, int)**.

13) **boolean endsWith**(**String** suffix): Tests if this string ends with the specified suffix.

14) **equals**(**Object** anObject): Compares this string to the specified object.

15) **boolean equalsIgnoreCase**(**String** anotherString): Compares this String to another String, ignoring case considerations.

16) **static String format**(**Locale** l, **String** format, **Object**... args): Returns a formatted string using the specified locale, format string, and arguments.

17) **static String format**(**String** format, **Object**... args): Returns a formatted string using the specified format string and arguments.

18) **byte[] getBytes**(): Encodes this String into a sequence of bytes using the platform's default charset, storing the result into a new byte array.

19) **byte[] getBytes(Charset** charset): Encodes this String into a sequence of bytes using the given **charset**, storing the result into a new byte array.

20) **void getBytes**(int srcBegin, int srcEnd, byte[] dst, int dstBegin)**Deprecated.** This method does not properly convert characters into bytes. As of JDK 1.1, the preferred way to do this is via the **getBytes()** method, which uses the platform's default charset.

21) **byte[] getBytes(String** charsetName): Encodes this String into a sequence of bytes using the named charset, storing the result into a new byte array.

22) **void getChars**(int srcBegin, int srcEnd, char[] dst, int dstBegin): Copies characters from this string into the destination character array.

23) **int hashCode**(): Returns a hash code for this string.

24) **int indexOf**(int ch): Returns the index within this string of the first occurrence of the specified character.

25) **int indexOf**(int ch, int fromIndex): Returns the index within this string of the first occurrence of the specified character, starting the search at the specified index.

26) **int indexOf(String** str): Returns the index within this string of the first occurrence of the specified substring.

27) **int indexOf(String** str, int fromIndex): Returns the index within this string of the first occurrence of the specified substring, starting at the specified index.

28) **String intern**(): Returns a canonical representation for the string object.

29) **boolean isEmpty**(): Returns true if, and only if, **length()** is 0.

30) **static String join(CharSequence** delimiter, **CharSequence**... elements): Returns a new String composed of copies of the CharSequence elements joined together with a copy of the specified delimiter.

31) **static String join(CharSequence** delimiter, **Iterable**<? extends **CharSequence**> elements): Returns a new String composed of copies of the CharSequence elements joined together with a copy of the specified delimiter.

32) **int lastIndexOf**(int ch): Returns the index within this string of the last occurrence of the specified character.

33) **int lastIndexOf**(int ch, int fromIndex): Returns the index within this string of the last occurrence of the specified character, searching backward starting at the specified index.

34) **int lastIndexOf(String** str): Returns the index within this string of the last occurrence of the specified substring.

35) **int lastIndexOf(String** str, int fromIndex): Returns the index within this string of the last occurrence of the specified substring, searching backward starting at the specified index.

36) **int length**(): Returns the length of this string.

37) **boolean matches(String** regex): Tells whether or not this string matches the given **regular expression**.

38) **int offsetByCodePoints**(int index, int codePointOffset): Returns the index within this String that is offset from the given index by codePointOffset code points.

39) **boolean regionMatches**(boolean ignoreCase, int toffset, **String** other, int ooffset, int len): Tests if two string regions are equal.

40) **boolean regionMatches**(int toffset, **String** other, int ooffset, int len): Tests if two string regions are equal.

41) **String replace**(char oldChar, char newChar): Returns a string resulting from replacing all occurrences of oldChar in this string with newChar.

42) **String replace**(**CharSequence** target, **CharSequence** replacement): Replaces each substring of this string that matches the literal target sequence with the specified literal replacement sequence.

43) **String replaceAll**(**String** regex, **String** replacement): Replaces each substring of this string that matches the given **regular expression** with the given replacement.

44) **String replaceFirst**(**String** regex, **String** replacement): Replaces the first substring of this string that matches the given **regular expression** with the given replacement.

45) **String[] split**(**String** regex): Splits this string around matches of the given **regular expression**.

46) **String[]split**(**String** regex, int limit): Splits this string around matches of the given **regular expression**.

47) **boolean startsWith**(**String** prefix): Tests if this string starts with the specified prefix.

48) **boolean startsWith**(**String** prefix, int toffset): Tests if the substring of this string beginning at the specified index starts with the specified prefix.

49) **CharSequence subSequence**(int beginIndex, int endIndex): Returns a character sequence that is a subsequence of this sequence.

50) **String substring**(int beginIndex): Returns a string that is a substring of this string.

51) **String substring**(int beginIndex, int endIndex): Returns a string that is a substring of this string.

52) **char[] toCharArray**(): Converts this string to a new character array.

53) **String toLowerCase**(): Converts all of the characters in this String to lower case using the rules of the default locale.

54) **String toLowerCase**(**Locale** locale): Converts all of the characters in this String to lower case using the rules of the given Locale.

55) **String toString**(): This object (which is already a string!) is itself returned.

56) **String toUpperCase**(): Converts all of the characters in this String to upper case using the rules of the default locale.

57) **String toUpperCase**(**Locale** locale): Converts all of the characters in this String to upper case using the rules of the given Locale.

58) **String trim**(): Returns a string whose value is this string, with any leading and trailing whitespace removed.

59) **static String valueOf**(boolean b): Returns the string representation of the boolean argument.

60) **static String valueOf**(char c): Returns the string representation of the char argument.

61) **static String valueOf**(char[] data): Returns the string representation of the char array argument.

62) **static String valueOf**(char[] data, int offset, int count): Returns the string representation of a specific subarray of the char array argument.

63) **static String valueOf**(double d): Returns the string representation of the double argument.

64) **static String valueOf**(float f): Returns the string representation of the float argument.

65) **static String valueOf**(int i): Returns the string representation of the int argument.

66) **static String valueOf**(long l): Returns the string representation of the long argument.

67) **static String valueOf**(**Object** obj): Returns the string representation of the Object argument.

StringBuffer Class:

- **StringBuffer in Java** is a peer class of String that provides much of the functionality of strings. It provides more flexibility than String.
- As we know that a string is fixed-length, immutable means cannot be modified. That is, once we create a String object, we cannot change its content.
- To overcome this, Java language introduced another class called StringBuffer.

- Java StringBuffer represents a growable nature and mutable. That means that once we create a StringBuffer class object, we can perform any required changes in the object. i.e, its data can be modified.

- That's why **StringBuffer class objects are mutable in Java**. It creates strings of flexible length that can be modified in terms of both length and content. That's why StringBuffer is more flexible than String.

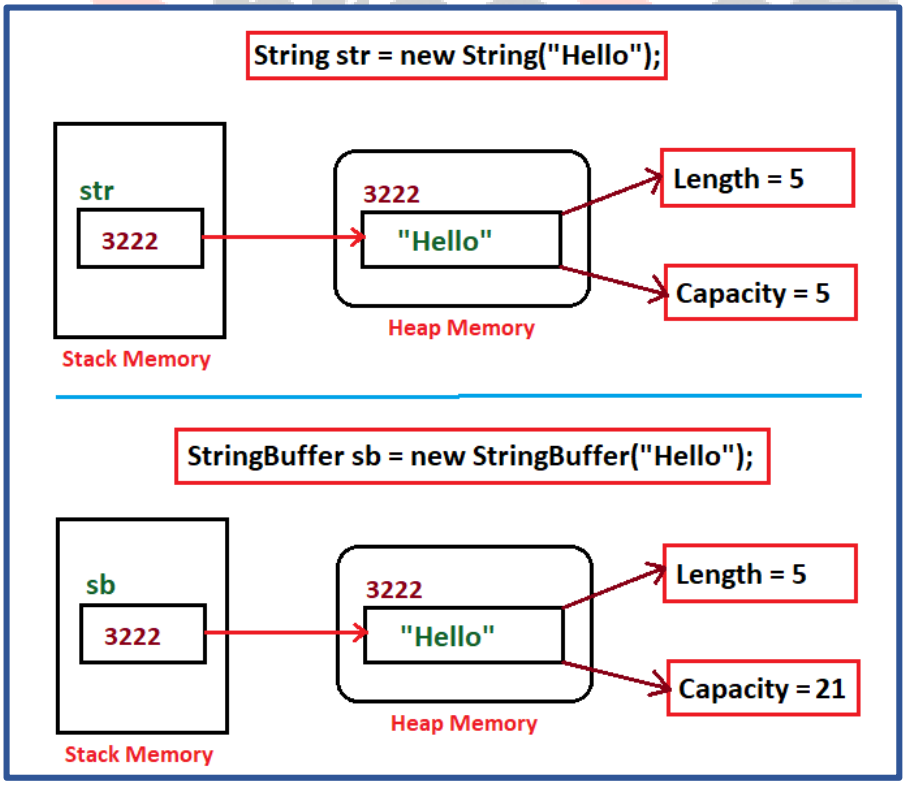
- The methods of string buffer class can directly manipulate data inside the object. We can easily insert characters and substrings in the middle of the string or add another string to the end. StringBuffer can automatically expand to create room for such additions.

- StringBuffer class method is **synchronized.** Therefore, Java StringBuffer class is **thread-safe** i.e. multiple threads cannot access it simultaneously. So it is safe and will result in an order but it is slower than string.

- **String constant pool is applicable only for string but not for StringBuffer. This is because string is the most common object used in Java whereas StringBuffer is rarely used in the application.**

- StringBuffer class is present in java.lang package similar to the string class.

Difference between Length and Capacity in Java StringBuffer



StringBuffer Class Constructor:

1. **StringBuffer():** This constructor creates an empty string buffer object with an initial capacity of 16 characters.

2. **StringBuffer(int size):** This constructor creates an empty string buffer object and accepts an integer value that specifies the size or length of the string object.

3. **StringBuffer(String str):** This constructor creates a string buffer object with the specified string whose size is equal to the string specified in this constructor.

4. **StringBuffer(CharSequence seq):** This form of constructor creates a string buffer that contains the same characters as the specified CharSequence.

StringBuffer Class Methods:

1) **StringBuffer append**(boolean b): Appends the string representation of the boolean argument to the sequence.

2) **StringBuffer append**(char c): Appends the string representation of the char argument to this sequence.

3) **StringBuffer append**(char[] str): Appends the string representation of the char array argument to this sequence.

4) **StringBuffer append**(char[] str, int offset, int len): Appends the string representation of a subarray of the char array argument to this sequence.

5) **StringBuffer append**(**CharSequence** s): Appends the specified CharSequence to this sequence.

6) **StringBuffer append**(**CharSequence** s, int start, int end): Appends a subsequence of the specified CharSequence to this sequence.

7) **StringBuffer append**(double d): Appends the string representation of the double argument to this sequence.

8) **StringBuffer append**(float f): Appends the string representation of the float argument to this sequence.

9) **StringBuffer append**(int i): Appends the string representation of the int argument to this sequence.

10) **StringBuffer append**(long lng): Appends the string representation of the long argument to this sequence.

11) **StringBuffer append**(**Object** obj): Appends the string representation of the Object argument.

12) **StringBuffer append**(**String** str): Appends the specified string to this character sequence.

13) **StringBuffer append(StringBuffer** sb): Appends the specified StringBuffer to this sequence.

14) **StringBuffer appendCodePoint**(int codePoint): Appends the string representation of the codePoint argument to this sequence.

15) **int capacity**(): Returns the current capacity.

16) **char charAt**(int index): Returns the char value in this sequence at the specified index.

17) **int codePointAt**(int index): Returns the character (Unicode code point) at the specified index.

18) **int codePointBefore**(int index): Returns the character (Unicode code point) before the specified index.

19) **int codePointCount**(int beginIndex, int endIndex): Returns the number of Unicode code points in the specified text range of this sequence.

20) **StringBuffer delete**(int start, int end): Removes the characters in a substring of this sequence.

21) **StringBuffer deleteCharAt**(int index): Removes the char at the specified position in this sequence.

22) **void ensureCapacity**(int minimumCapacity): Ensures that the capacity is at least equal to the specified minimum.

23) **void getChars**(int srcBegin, int srcEnd, char[] dst, int dstBegin): Characters are copied from this sequence into the destination character array dst.

24) **int indexOf**(**String** str): Returns the index within this string of the first occurrence of the specified substring.

25) **int indexOf**(**String** str, int fromIndex): Returns the index within this string of the first occurrence of the specified substring, starting at the specified index.

26) **StringBuffer insert**(int offset, boolean b): Inserts the string representation of the boolean argument into this sequence.

27) **StringBuffer insert**(int offset, char c): Inserts the string representation of the char argument into this sequence.

28) **StringBuffer insert**(int offset, char[] str): Inserts the string representation of the char array argument into this sequence.

29) **StringBuffer insert**(int index, char[] str, int offset, int len): Inserts the string representation of a subarray of the str array argument into this sequence.

30) **StringBuffer insert**(int dstOffset, **CharSequence** s): Inserts the specified CharSequence into this sequence.

31) **StringBuffer insert**(int dstOffset, **CharSequence** s, int start, int end): Inserts a subsequence of the specified CharSequence into this sequence.

32) **StringBuffer insert**(int offset, double d): Inserts the string representation of the double argument into this sequence.

33) **StringBuffer insert**(int offset, float f): Inserts the string representation of the float argument into this sequence.

34) **StringBuffer insert**(int offset, int i): Inserts the string representation of the second int argument into this sequence.

35) **StringBuffer insert**(int offset, long l): Inserts the string representation of the long argument into this sequence.

36) **StringBuffer insert**(int offset, **Object** obj): Inserts the string representation of the Object argument into this character sequence.

37) **StringBuffer insert**(int offset, **String** str): Inserts the string into this character sequence.

38) **int lastIndexOf**(**String** str): Returns the index within this string of the rightmost occurrence of the specified substring.

39) **int lastIndexOf**(**String** str, int fromIndex): Returns the index within this string of the last occurrence of the specified substring.

40) **int length**(): Returns the length (character count).

41) **int offsetByCodePoints**(int index, int codePointOffset): Returns the index within this sequence that is offset from the given index by codePointOffset code points.

42) **StringBuffer replace**(int start, int end, **String** str): Replaces the characters in a substring of this sequence with characters in the specified String.

43) **StringBuffer reverse**(): Causes this character sequence to be replaced by the reverse of the sequence.

44) **void setCharAt**(int index, char ch): The character at the specified index is set to ch.

45) **void setLength**(int newLength): Sets the length of the character sequence.

46) **CharSequence subSequence**(int start, int end): Returns a new character sequence that is a subsequence of this sequence.

47) **String substring**(int start): Returns a new String that contains a subsequence of characters currently contained in this character sequence.

48) **String substring**(int start, int end): Returns a new String that contains a subsequence of characters currently contained in this sequence.

49) **String toString**(): Returns a string representing the data in this sequence.

50) **void trimToSize**(): Attempts to reduce storage used for the character sequence.

StringBuilder Class:

- Java **StringBuilder** class is used to create mutable (modifiable) string. The Java **StringBuilder** class is same as **StringBuffer** class except that it is non-synchronized.

- As we know instances of **StringBuilder** are not safe for use by multiple threads. If such synchronization is required then it is recommended that **StringBuffer** is used.

- StringBuilder class was introduced in Java 5. It is present in java.lang.StringBuilder package and automatically imported into every java program.

StringBuilder Class Constructor:

1. **StringBuffer():** This constructor creates an empty string buffer object with an initial capacity of 16 characters.

2. **StringBuffer(int size):** This constructor creates an empty string buffer object and accepts an integer value that specifies the size or length of the string object.

3. **StringBuffer(String str):** This constructor creates a string buffer object with the specified string whose size is equal to the string specified in this constructor.

4. **StringBuffer(CharSequence seq):** This form of constructor creates a string buffer that contains the same characters as the specified CharSequence.

StringBuffer Class Methods:

In addition to the inherited methods from the Object class and CharSequence, methods provided by StringBuilder class are almost similar to methods of StringBuffer class.

Difference Between StringBuffer, StringBuilder:

	String	StringBuffer	StringBuilder
String Type	A String class always create an immutable class.	It creates mutable string.	It creates another way of mutable string
Thread Safe	It does not ensure thread safety as it's not synchronized.	It is synchronized which assures thread safe environment.	It is not thread safe.
Performance	It serves slower performance as its creates a separate object on each manipulations.	It is synchronized that is the reason it provides slower performance than StringBuilder.	It is not thread safe and that is the reason it provides faster performance than StringBuffer.
Memory Wastage	Each time when manipulations are made to a string, a new object with applied changes created, despite the instance which always reference to original string.	Operations are always performed on original string object.	It is same as StringBuffer. i.e. operations performed on original object.
Memory Allocation	It gets memory on String Constant Pool, even though it generated from String constructor as it first gets space on Heap memory but afterwards ends up with string constant pool.	It used Heap Memory to store a string.	It used Heap Memory to store a string.
Introduced in	Java 1.0	Java 1.2	Java 1.5

Note:

1) **String concatenation operator (+) internally uses StringBuffer or StringBuilder class.**
2) **For String manipulations in a non-multi threaded environment, we should use StringBuilder else use StringBuffer class.**

We can understand performance wise difference between String, StringBuffer and StringBuilder.

Program1: Performance of String class example

```
class Example implements Runnable {
	String str = new String("Shreesoft");
	@Override
	public void run() {
		for (int i = 0; i < 10000000; i++) {
			str.append("Informatics");
		}
	}
}
public class StringPerformanceExample {
	public static void main(String[] args) {
		Example obj = new Example();
		Thread thread1 = new Thread(obj);

		long start = System.currentTimeMillis();

		thread1.start();

		try {
		thread1.join();
		} catch (InterruptedException e) {
		e.printStackTrace();
		}

		long end = System.currentTimeMillis();
		System.out.printf((end - start) + "ms");
	}
}
```

Program1: Performance of StringBuffer class example

```
class Example implements Runnable {
	StringBuffer str = new StringBuffer("Shreesoft");
	@Override
	public void run() {
		for (int i = 0; i < 10000000; i++) {
			str.append("Informatics");
		}
	}
}

public class StringBufferPerformanceExample {
	public static void main(String[] args) {
		Example obj = new Example();
		Thread thread1 = new Thread(obj);
```

```
		long start = System.currentTimeMillis();

		thread1.start();

		try {
		thread1.join();
		} catch (InterruptedException e) {
		e.printStackTrace();
		}

		long end = System.currentTimeMillis();
		System.out.printf((end - start) + "ms");
	}
}
```

Program1: Performance of StringBuilder class example

```
class Example implements Runnable {
	StringBuilder str = new StringBuilder("Shreesoft");
	@Override
	public void run() {
		for (int i = 0; i < 10000000; i++) {
			str.append("Informatics");
		}
	}
}
public class StringBuilderPerformanceExample {
	public static void main(String[] args) {
		Example obj = new Example();
		Thread thread1 = new Thread(obj);
		long start = System.currentTimeMillis();

		thread1.start();

		try {
		thread1.join();
		} catch (InterruptedException e) {
		e.printStackTrace();
		}

		long end = System.currentTimeMillis();
		System.out.printf((end - start) + "ms");
	}
}
```

Programs:

1) Write a program to find length of given string.
2) Write a program to print reverse the given string.
3) Write a program to print reverse the given string without using reverse methods.
4) Write a program to check the given string is palindrome or not.
5) Write a program to print / convert given string into uppercase and lowercase letters.
6) Write a program to count number of alphabet, digit and special character.
7) Write a program to count vowel, UpperCase and Lowercase letter.
8) Write a program to reverse each word of given string.
9) Write a program to read two string and concatenate or join it.
10) Write a program to read two string and compare it with case.
11) Write a program to read two string and compare it without case.
12) Write a program to read two string and compare it with case.
13) Write a program to print duplicate characters from the string?
14) Write a program to insert one string into another string.
15) Write a program to delete the given character from the given location.
16) Write a program to delete the substring from given string.
17) Write a program to replace the one substring by another substring in the given string.
18) Write a program to check substring present in given string or not.
19) Write a program to check the given string is anagram or not.
20) Program to make first alphabet capital of each word in given string.