Class topics

1. Introduction of Strings
2. Why are Strings Immutable?
3. Java String Concatenation
4. String Class
5. StringBuffer Class
6. StringBuilder Class
7. Strings vs StringBuffer vs StringBuilder
8. Java String Programs

Introduction of Strings

In Java, a String is the type of object that can store a sequence of characters enclosed by double quotes, and every character is stored in 16 bits, i.e., using UTF 16-bit encoding. A string acts the same as an array of characters. Java provides a robust and flexible API for handling strings, allowing for various operations such as concatenation, comparison, and manipulation.

Example:

```
String name = "Pathshala360";
String num = "1234"
```

Program:

```
// Java Program to demonstrate String
public class StringExample {
    public static void main(String args[]) {
        // creating Java string using a new keyword
        String str = new String("Pathshala360");
        System.out.println(str);
    }
}

Output
Pathshala360
```

Ways of Creating a Java String

There are two ways to create a string in Java:

- String Literal
- Using new Keyword

Syntax:

```
<String_Type> <string_variable> = "<sequence_of_string>";
```

1. String literal (Static Memory)

To make Java more memory efficient (because no new objects are created if it exists already in the string constant pool).

Example:

```
String demoString = "Pathshala360";
```

2. Using new keyword (Heap Memory)

- String s = new String("Welcome");
- In such a case, JVM will create a new string object in normal (non-pool) heap memory and the literal "Welcome" will be placed in the string constant pool. The variable s will refer to the object in the heap (non-pool)

In the given example only one object will be created. Firstly JVM will not find any string object with the value "Welcome" in the string constant pool, so it will create a new object. After that it will find the string with the value "Welcome" in the pool, it will not create a new object but will return the reference to the same instance.

Example:

```
String demoString = new String ("Welcome");
```

Interfaces and Classes in Strings in Java

- **CharBuffer:** This class implements the CharSequence interface. This class is used to allow character buffers to be used in place of CharSequences. An example of such usage is the regular-expression package java.util.regex.
- **String:** It is a sequence of characters. In Java, objects of String are immutable which means a constant and cannot be changed once created.

CharSequence Interface

CharSequence Interface is used for representing the sequence of Characters in Java. Classes that are implemented using the CharSequence interface are mentioned below and these provide much of functionality like substring, last occurence, first occurence, concatenate , toupper, tolower etc.

- String
- StringBuffer
- StringBuilder

1. String

String is an immutable class which means a constant and cannot be changed once created and if wish to change , we need to create a new object and even the functionality it provides like toupper, tolower, etc all return a new object , it's not modify the original object. It is automatically thread safe.

Syntax:

```
// Method 1
String str= "Pathshala360";
// Method 2
String str= new String("Pathshala360")
```

2. StringBuffer

StringBuffer is a peer class of String, it is mutable in nature and it is thread safe class , we can use it when we have a multi threaded environment and shared object of string buffer i.e, used by multiple threads. As it is thread safe so there is extra overhead, so it is mainly used for multithreaded programs.

Syntax:

```
StringBuffer demoString = new StringBuffer("Pathshala360");
```

3. StringBuilder

StringBuilder in Java represents an alternative to String and StringBuffer Class, as it creates a mutable sequence of characters and it is not thread safe. It is used only within the thread , so there is no extra overhead , so it is mainly used for single threaded programs.

Syntax:

```
StringBuilder demoString = new StringBuilder();
demoString.append("Pathshala360");
```

Immutable String in Java

In Java, string objects are immutable. Immutable simply means unmodifiable or unchangeable. Once a string object is created its data or state can't be changed but a new string object is created.

Example:

```
// Java Program to demonstrate Immutable String in Java
import java.io.*;

class Main {
    public static void main(String[] args) {
        String s = "Sachin";
        // concat() method appends the string at the end
        s.concat("Tendulkar");
        // This will print Sachin because strings are immutable objects
        System.out.println(s);
    }
}

Output
Sachin
```

Here, Sachin is not changed but a new object is created with “Sachin Tendulkar”. That is why a string is known as immutable. As we can see in the given figure that two objects are created but the reference variable still refers to "Sachin" and not to "Sachin Tendulkar". But if we explicitly assign it to the reference variable, it will refer to the "Sachin Tendulkar" object.

Example: Java program to assign the reference explicitly in String using String.concat() method.

```
// Java Program to demonstrate Explicitly assigned strings
import java.io.*;

Class Main {
    public static void main(String[] args) {
        String name = "Sachin";
        name = name.concat(" Tendulkar");
        System.out.println(name);
    }
}

Output
Sachin Tendulkar
```

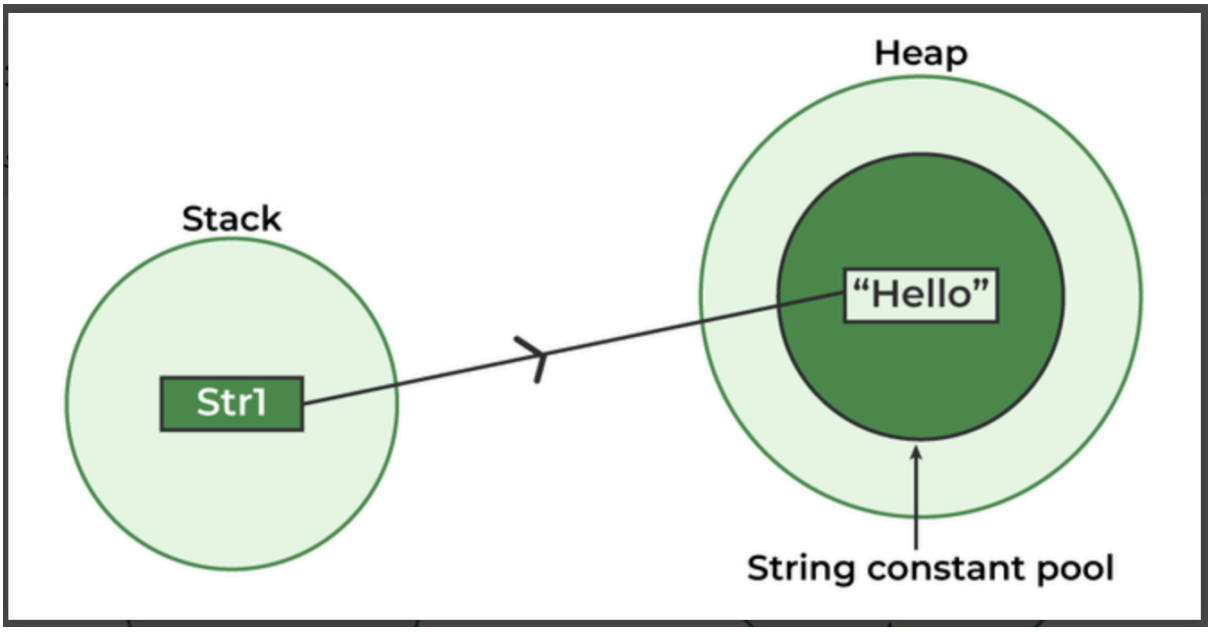
Memory Allotment of Java String

String literal

Whenever a String Object is created as a literal, the object will be created in the String constant pool. This allows JVM to optimize the initialization of String literal. The string constant pool is present in the heap.

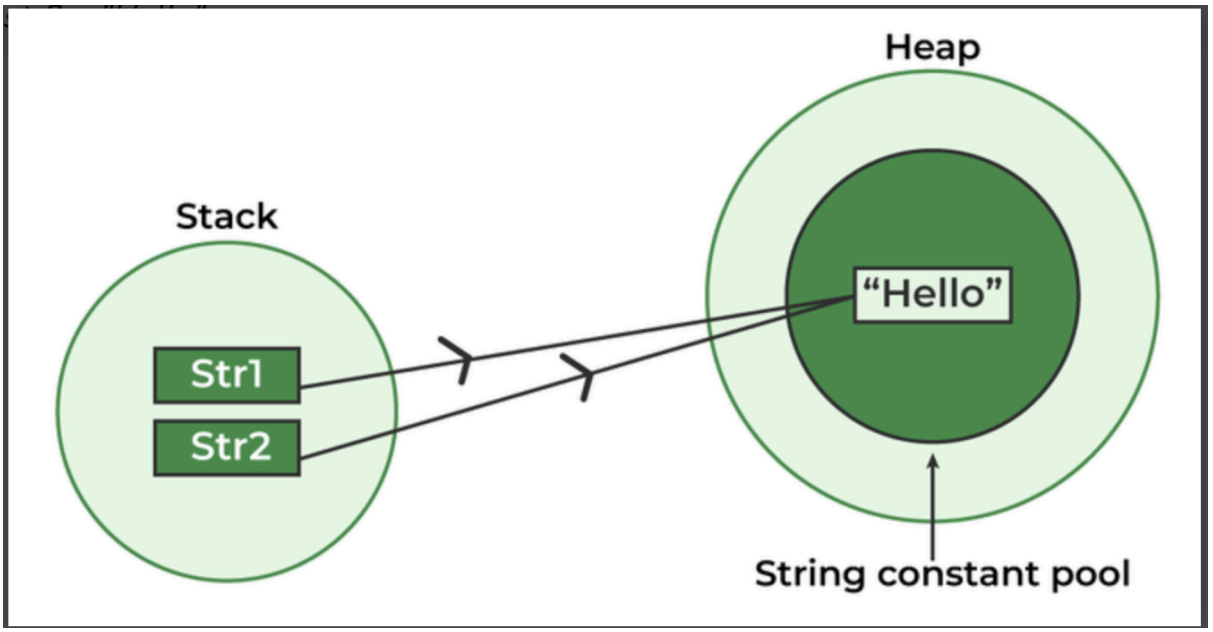
Example 1: Using String literals to assign char sequence value.

```
String str1 = "Hello";
```



Example 2: When we initialize the same char sequence using string literals.

```
String str1 = "Hello";
String str2 = "Hello";
```

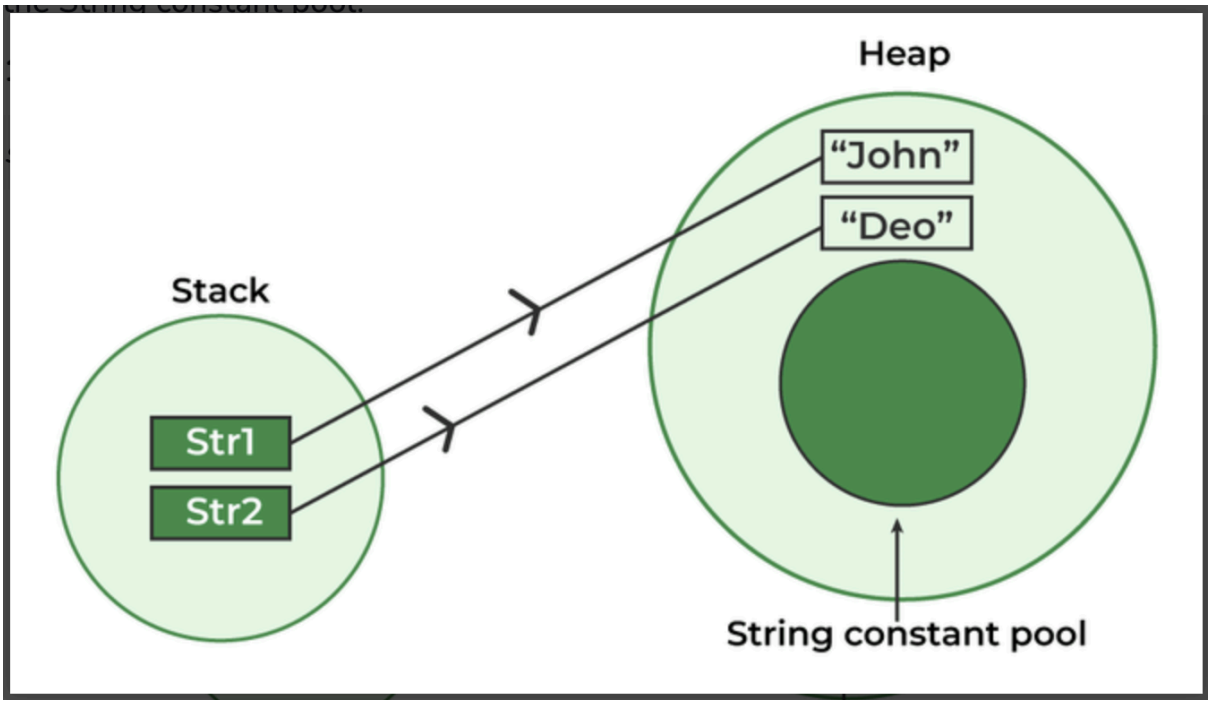


Using new Keyword

The string can also be declared using a new operator i.e. dynamically allocated. In case String is dynamically allocated they are assigned a new memory location in the heap . This string will not be added to the String constant pool.

Example 1: Using a new keyword to assign a char sequence to a String object.

```
String str1 = new String("John"); String str2 = new String("Deo");
```



If we want to store this string in the constant pool then we will need to "intern" it.

Example 2: Using .intern() to add a string object in string constant pool.

// this will add the string to the string constant pool.

```
String internedString = demoString.intern();
```

It is preferred to use String literals as it allows JVM to optimize memory allocation.

If we notice if we use new keyword or string literals both store the values in the string but the difference is if we use the string literals or intern() the string object it will store the values in the string constant pool which is present inside the heap.

Example that shows how to declare a String:

```
// Java Program to Declare a String
import java.io.*;
import java.lang.*;

class Main {
    public static void main(String[] args) {
        // Declare String without using new operator
        String name = "Pathshala360";
        // Prints the String.
        System.out.println("String name = " + name);
        // Declare String using new operator
        String newString = new String("Pathshala360");
        // Prints the String.
        System.out.println("String newString = " + newString);
    }
}

Output
String name = Pathshala360
String newString = Pathshala360
```

Note: String Objects are created in Heap area and Literals are stored in a special memory area known as string constant pool.

Why are Java Strings Immutable?

In Java, strings are immutable means their values cannot be changed once they are created. This feature enhances performance, security, and thread safety. In this article, we are going to learn why strings are immutable in Java and how this benefits Java applications.

What Does Immutable Mean?

When we say something is immutable, it means that once it is created, it cannot be changed. In Java, this concept applies to strings, which means that once a string object is created, its content cannot be changed. If we try to change a string, Java does not modify the original one, it creates a new string object instead.

How are Strings Immutable in Java?

Java strings are immutable to make sure memory is used efficiently. Strings in Java that are specified as immutable as the content is shared storage in a single pool to minimize creating a copy of the same value. String class and all wrapper classes in Java that include Boolean, Character, Byte, Short, Integer, Long, Float, and Double are immutable. A user is free to create immutable classes of their own.

When we use methods like concat() or replace(), they don't alter the original string but create a new one with the new content. This helps avoid bugs that might arise from modifying shared data across different parts of the program.

Let's understand this with an example:

```
public class Main {
    public static void main(String[] args) {
        String s1 = "knowledge";
        String s2 = s1;            // s2 points to the same "knowledge"
        s1 = s1.concat(" base");// creates a new String "knowledge base"
        System.out.println(s1);
    }
}

Output
knowledge base
```

Explanation: When we call s1.concat(" base"), it does not modify the original string "knowledge". It only creates a new string "knowledge base" and assigns it to s1. The original string remains unchanged.

Why Are Java Strings Immutable?

- String Pool: Java stores string literals in a pool to save memory. Immutability ensures one reference does not change the value for others pointing to the same string.
- Security: Strings are used for sensitive data like usernames and passwords. Immutability prevents attackers from altering the values.
- Thread Safety: Since string values cannot be changed, they are automatically thread-safe, meaning multiple threads can safely use the same string.
- Efficiency: The JVM reuses strings in the String Pool by improving memory usage and performance.

Example to Demonstrate String Immutability

The below program demonstrate the immutability of Java strings, where we try to modify a string using concat():

```
// Java Program to demonstrate why Java Strings are immutable
import java.io.*;

class Main {
    public static void main(String[] args) {
        String s1 = "java";
        // creates a new String "java rules, but does not change s1
        s1.concat(" rules");
        // s1 still refers to "java"
        System.out.println("s1 refers to " + s1);
    }
}

Output
s1 refers to java
```

Explanation: In the above example, even though we call concat() to append " rules", the original string s1 still refers to "java". The new string "java rules" is created, but it is not assigned to any variable, so it is lost.

Java String Concatenation

The string concat() method concatenates (appends) a string to the end of another string. It returns the combined string. It is used for string concatenation in Java. It returns a NullPointerException if any one of the strings is Null. In this article, we will learn how to concatenate two strings in Java.

Program to Concatenate Two Strings using concat() Method in Java
Below is the simplest method to concatenate the string in Java. We will be using the concat() method of String class to append one string to the end of another and return the new string.

```
// Java Program to Illustrate concat() method of String.
class Main {
    public static void main(String args[]) {
        // String Initialization
        String s = "Pathshala";
        // Use concat() method for string concatenation
        s = s.concat("360");
        System.out.println(s);
    }
}

Output
Pathshala360
```

Syntax of concat() Method

```
public String concat (String s);
```

Parameters:

A string to be concatenated at the end of the other string.

Return Value:

Concatenated(combined) string.

Exception:

NullPointerException: When either of the string is Null.

Java String concat() Examples

There are many ways to use the concat() method in Java:

Sequential Concatenation of Multiple Strings

The below example shows sequential concatenation using String concat() method in Java.

Example:

```
// Java program to Illustrate Working of concat() method in strings
where we are sequentially adding multiple strings as we need
public class Main{
    public static void main(String args[]) {
        String s1 = "Computer-";
        String s2 = "Science-";

        // Combining above strings by passing one string as an argument
        String s3 = s1.concat(s2);
        // Print and display temporary combined string
        System.out.println(s3);
        String s4 = "Portal";
        String s5 = s3.concat(s4);
        System.out.println(s5);
    }
}

Output
Computer-Science-
Computer-Science-Portal
```

Note: As perceived from the code we can do as many times as we want to concatenate strings bypassing older strings with new strings to be contaminated as a parameter and storing the resultant string in String datatype.

Handling NullPointerException in String concat()

The below example shows the NullPointerException in String concat() method.

Example:

```
// Java program to Illustrate NullPointerException
public class Main{
    public static void main(String args[]) {
        String s1 = "Computer-";
        String s2 = null;
        String s3 = s1.concat(s2);
        System.out.println(s3);
    }
}
Output:
Exception in thread "main" java.lang.NullPointerException
```

Combining Two Strings with concat() Method

The below example combines two strings using the concat() method of string class.

Example:

```
// Java Program to combine two strings with concat()
class Main{
    public static void main(String args[]) {
        // String Initialization
        String s1 = "Pathshala";
        String s2 = "360";

        // Concatenate the strings s1 and s2 using the concat() method
and store the result back in s1.
        s1 = s1.concat(s2);
        System.out.println(s1);
    }
}

Output
Pathshala360
```

Reversing a String Using concat() Method

We can reverse a string using the concat() method of string class. Below is the example to reverse a string in Java.

Example:

```
// Java Program to reverse a string using concat() method
public class ReverseString {
    public static void main(String[] args) {
        String a = "path";
        String b = "";
        for (int i = a.length() - 1; i >= 0; i--) {
            // Extract the current character at index "i" of the "a"
string
            char ch = a.charAt(i);
            String ch1 = Character.toString(ch);
            b = b.concat(ch1);
        }
        System.out.println("" + a);
        System.out.println("" + b);
    }
}

Output
path
htap
```

String Class in Java

A string is a sequence of characters. In Java, objects of the String class are immutable, which means they cannot be changed once created. In this article, we are going to learn about the String class in Java.

Example of String Class in Java:

```
// Java Program to Create a String
import java.io.*;

class Main {
    public static void main (String[] args) {
        String s = "String in Java";
        System.out.println(s);
    }
}

Output
String in Java
```

Creating a String

There are two ways to create string in Java:

1. **Using String literal**
   String s = “Pathshala360”;
2. **Using new keyword**
   String s = new String (“Pathshala360”);

Example of Creating String in Java

```
import java.io.*;

class Main {
    public static void main (String[] args) {
        // String literal
        String s1="String1";
        System.out.println(s1);
        // Using new Keyword
        String s2= new String("String2");
        System.out.println(s2);
    }
}

Output
String1
String2
```

String Constructors in Java

String Constructors	Description
String(byte[] byte_arr)	Construct a new String by decoding the byte array. It uses the platform's default character set for decoding.
String(byte[] byte_arr, Charset char_set)	Construct a new String by decoding the byte array. It uses the char_set for decoding.
String(byte[] byte_arr, int start_index, int length)	Construct a new string from the bytes array depending on the start_index(Starting location) and length(number of characters from starting location).
String(byte[] byte_arr, int start_index, int length, Charset char_set)	Construct a new string from the bytes array depending on the start_index(Starting location) and length(number of characters from starting location).Uses char_set for decoding.
String(char[] char_arr)	Allocates a new String from the given Character array.
String(char[] char_array, int start_index, int count)	Allocates a String from a given character array but chooses count characters from the start_index.
String(int[] uni_code_points, int offset, int count)	Allocates a String from a uni_code_array but chooses count characters from the start_index.
String(StringBuffer s_buffer)	Allocates a new string from the string in s_buffer.
String(StringBuilder s_builder)	Allocates a new string from the string in s_builder.

Let us check these constructors using an example demonstrating the use of them.

Example String Constructor in Java

```
// Java Program to implement String Constructor
import java.nio.charset.Charset;

class Main {
    // Variables in Methods
    static byte[] byteArr = {65, 112, 112, 108, 101};
    static Charset cs = Charset.defaultCharset();
    static char[] charArr = {'a', 'p', 'p', 'l', 'e'};
    static int[] uniCode = {65, 112, 112, 108, 101};
    static StringBuffer buffer = new StringBuffer("Apple");
    static StringBuilder builder = new StringBuilder("Apple");

    public static void main(String[] args) {
        // Method 1
        String s1 = new String(byteArr);
        System.out.println("Method 1: " + s1);
        System.out.println();
        // Method 2
        String s2 = new String(b_arr, cs);
        System.out.println("Method 2: " + s2);
        System.out.println();
        // This is alternative way for Method 2
        String s3 = new String(byteArr, Charset.forName("US-ASCII"));
        System.out.println("Method 2 Alternative: " + s3);
        System.out.println();
        // Method 3
        String s4 = new String(byteArr, 1, 3);
        System.out.println("Method 3: " + s4);
        System.out.println();
        // Method 4
        String s5 = new String(byteArr, 1, 3, cs);
        System.out.println("Method 4: " + s5);
        System.out.println();
        // This is alternative way for Method 4
        String s6 = new String(byteArr, 1,
4,Charset.forName("US-ASCII"));
        System.out.println("Method 4 Alternative: " + s6);
        System.out.println();
        // Method 5
        String s7 = new String(charArr);
        System.out.println("Method 5: " + s7);
```

```
        System.out.println();
        // Method 6
        String s8 = new String(charArr, 1, 3);
        System.out.println("Method 6: " + s8);
        System.out.println();
        // Method 7
        String s9 = new String(uniCode, 1, 3);
        System.out.println("Method 7: " + s9);
        System.out.println();
        // Method 8
        String s10 = new String(buffer);
        System.out.println("Method 8: " + s10);
        System.out.println();
        // Method 9
        String s11 = builder.toString();
        System.out.println("Method 9: " + s11);
        System.out.println();
    }
}
Output
Method 1: Apple
Method 2: Apple
Method 2 Alternative: Apple
Method 3: ppl
Method 4: ppl
Method 4 Alternative: pple
Method 5: Apple
Method 6: ppl
Method 7: ppl
Method 8: Apple
Method 9: Apple...
```

String Methods in Java

String Methods	Description
int length()	Returns the number of characters in the String.
Char charAt(int i)	Returns the character at ith index.
String substring (int i)	Return the substring from the ith index character to end.
String substring (int i, int j)	Returns the substring from i to j-1 index.
String concat(String str)	Concatenates specified string to the end of this string.
int indexOf (String s)	Returns the index within the string of the first occurrence of the specified string. If String s is not present in the input string then -1 is returned as the default value.
int indexOf (String s, int i)	Returns the index within the string of the first occurrence of the specified string, starting at the specified index.
Int lastIndexOf(String s)	Returns the index within the string of the last occurrence of the specified string. If String s is not present in the input string then -1 is returned as the default value.
boolean equals(Object otherObj)	Compare this string to the specified object.
boolean equalsIgnoreCase (String anotherString)	Compares string to another string, ignoring case considerations.
int compareTo(String anotherString)	Compare two strings lexicographically.
int compareToIgnoreCase(String anotherString)	Compare two strings lexicographically, ignoring case considerations.
String toLowerCase()	Converts all the characters in the String to lowercase.

String toUpperCase()	Converts all the characters in the String to uppercase
String trim()	Returns the copy of the String, by removing whitespaces at both ends. It does not affect whitespaces in the middle.
String replace (char oldChar, char newChar)	Returns new string by replacing all occurrences of oldChar with newChar.
boolean contains(CharSequence sequence)	Returns true if the string contains the given string.
Char[] toCharArray():	Converts this String to a new character array.
boolean startsWith(String prefix)	Return true if the string starts with this prefix.

Example of String Constructor and String Methods

Below is the implementation of string constructors and methods in Java.

```
// Java program to illustrate different constructors and methods in
String class
import java.io.*;
import java.util.*;

class Main{
    public static void main (String[] args) {
        String s= "Pathshala360";
        // or String s= new String ("Pathshala360");

        // Returns the number of characters in the String
        System.out.println("String length = " + s.length());

        // Returns the character at ith index
        System.out.println("Character at 3rd position = "
                           + s.charAt(3));

        // Return the substring from the ith  index character
        // to end of string
        System.out.println("Substring " + s.substring(3));

        // Returns the substring from i to j-1 index
        System.out.println("Substring  = " + s.substring(2,5));

        // Concatenates string2 to the end of string1
        String s1 = "Pathshala";
        String s2 = "360";
        System.out.println("Concatenated string  = " +
                           s1.concat(s2));

        // Returns the index within the string
        // of the first occurrence of the specified string
        String s4 = "Learn Share Learn";
        System.out.println("Index of Share " +
                           s4.indexOf("Share"));

        // Returns the index within the string of the
        // first occurrence of the specified string,
        // starting at the specified index
        System.out.println("Index of a  = " +
                           s4.indexOf('a',3));

        // Checking equality of Strings
        Boolean out = "Apple".equals("apple");
```

```
        System.out.println("Checking Equality  " + out);
        out = "Apple".equals("Apple");
        System.out.println("Checking Equality  " + out);

        out = "Apple".equalsIgnoreCase("aPPle ");
        System.out.println("Checking Equality " + out);

        // If ASCII difference is zero then
        // the two strings are similar
        int out1 = s1.compareTo(s2);
        System.out.println("the difference between ASCII value
is="+out1);

        // Converting cases
        String w1 = "Apple";
        System.out.println("Changing to lower Case " +
                           w1.toLowerCase());

        // Converting cases
        String w2 = "Apple";
        System.out.println("Changing to UPPER Case " +w2.toUpperCase());

        // Trimming the word
        String w4 = " Pathshala 360 ";
        System.out.println("Trim the word " + w4.trim());
        // Replacing characters
        String str1 = "Axxle";
        System.out.println("Original String " + str1);
        String str2 = "Axxle".replace('f' ,'g') ;
        System.out.println("Replaced x with p -> " + str2);
    }
}
String length = 12
Character at 3rd position = h
Substring hshala360
Substring  = ths
Concatenated string  = Pathshala360
Index of Share 6
Index of a  = 8
Checking Equality  false
Checking Equality  true
Changing to LOWER Case apple
Changing to UPPER Case APPLE
Trim the word Pathshala 360
Original String Axxle
Replaced x with p -> Apple
```

StringBuffer Class in Java

The StringBuffer class in Java represents a sequence of characters that can be modified, which means we can change the content of the StringBuffer without creating a new object every time. It represents a mutable sequence of characters.

Features of StringBuffer Class

The key features of StringBuffer class are listed below:

- Unlike String, we can modify the content of the StringBuffer without creating a new object.
- StringBuffer has an initial capacity, and it can also be adjusted later with the help of the ensureCapacity() method.
- With the help of the append() method, we can add characters, strings, or objects at the end of the StringBuffer.
- With the help of the insert() method, we can insert characters, strings, or objects at a specified position in the StringBuffer.
- With the help of the delete() method, we can remove characters from the StringBuffer.
- With the help of reverse() method, we can reverse the order of characters in the StringBuffer.

Example: Here is an example of using StringBuffer to concatenate strings:

```
//Demonstrating String Buffer
public class Test {
    public static void main(String[] args){
        // Creating StringBuffer
        StringBuffer s = new StringBuffer();

        // Adding elements in StringBuffer
        s.append("Hello");
        s.append(" ");
        s.append("world");

        // String with the StringBuffer value
        String str = s.toString();
        System.out.println(str);
    }
}

Output
Hello world
```

Advantages of using StringBuffer in Java

- The advantages of StringBuffer class are listed below:
- Mutable: StringBuffer are mutable. It means that we can change the content after the object has been created, on the other hand String is immutable once it is created it can not be modified.
- Efficient: Since StringBuffer objects are mutable, it is suitable in scenarios where we need to modify the string multiple times. If we do the same thing with string, everytime a new object is created and the old one is deleted, which is very bad in terms of performance and memory.

Note: Both String and StringBuffer objects are thread safe, but in different ways. StringBuffer is synchronized; it means it is thread-safe but keep in mind that this synchronization can cause performance issues if accessed by multiple threads at the same time. On the other hand immutable objects like String are thread-safe because their state can not be modified once they are created.

Constructors of StringBuffer Class

Constructor	Description	Syntax
StringBuffer()	It reserves room for 16 characters without reallocation	StringBuffer s = new StringBuffer();
StringBuffer(int size)	It accepts an integer argument that explicitly sets the size of the buffer.	StringBuffer s = new StringBuffer(20);
StringBuffer(String str)	It accepts a string argument that sets the initial contents of the StringBuffer object and reserves room for 16 more characters without reallocation.	StringBuffer s = new StringBuffer("Pathshala360");

Methods of Java StringBuffer Class

Methods	Action Performed
append()	Used to add text at the end of the existing text.
length()	The length of a StringBuffer can be found by the length( ) method.
capacity()	The total allocated capacity can be found by the capacity() method.
charAt()	This method returns the char value in this sequence at the specified index.
delete()	Deletes a sequence of characters from the invoking object.
deleteCharAt()	Deletes the character at the index specified by the loc.
ensureCapacity()	Ensures capacity is at least equal to the given minimum.
insert()	Inserts text at the specified index position.
length()	Returns the length of the string.
reverse()	Reverse the characters within a StringBuffer object.
replace()	Replace one set of characters with another set inside a StringBuffer object.

Examples of Java StringBuffer Method

1. append() Method

The append() method concatenates the given argument with this string.

Example:

```
import java.io.*;

class Main {
    public static void main(String args[]) {
        StringBuffer sb = new StringBuffer("Hello ");
        sb.append("Java");
        System.out.println(sb);
    }
}

Output
Hello Java
```

2. insert() Method

The insert() method inserts the given string with this string at the given position.

Example:

```
import java.io.*;

class Main {
    public static void main(String args[]) {
        StringBuffer sb = new StringBuffer("Hello ");
        sb.insert(1, "Java");
        // Now original string is changed
        System.out.println(sb);
    }
}

Output
HJavaello
```

3. replace() Method

The replace() method replaces the given string from the specified beginIndex and endIndex - 1.

Example:

```
import java.io.*;

class Main {
    public static void main(String args[]) {
        StringBuffer sb = new StringBuffer("Hello");
        sb.replace(1, 3, "Java");
        System.out.println(sb);
    }
}

Output
HJavalo
```

4. delete() Method

The delete() method is used to delete the string from the specified beginIndex to endIndex-1.

Example:

```
import java.io.*;

class Main {
    public static void main(String args[]) {
        StringBuffer sb = new StringBuffer("Hello");
        sb.delete(1, 3);
        System.out.println(sb);
    }
}

Output
Hlo
```

5. reverse() Method

The reverse() method of the StringBuffer class reverses the current string.

Example:

```
import java.io.* ;

class Main {
    public static void main(String args[]) {
        StringBuffer sb = new StringBuffer("Hello");
        sb.reverse();
        System.out.println(sb);
    }
}

Output
olleH
```

6. capacity() Method

The capacity() method of the StringBuffer class returns the current capacity of the buffer. The default capacity of the buffer is 16. If the number of characters increases from its current capacity, it increases the capacity by (oldcapacity*2) + 2.

For example, if the current capacity is 16, it will be (16*2)+2=34.

Example:

```
import java.io.*;

class Main {
    public static void main(String args[]) {
        StringBuffer sb = new StringBuffer();
        // default 16
        System.out.println(sb.capacity());
        sb.append("Hello");
        // now 16
        System.out.println(sb.capacity());
        sb.append("java is my favourite language");
        // (oldcapacity * 2) + 2
        System.out.println(sb.capacity());
    }
}
Output
16
16
34
```

Some Interesting Facts about the StringBuffer Class

- java.lang.StringBuffer extends (or inherits from) Object class.
- All Implemented Interfaces of StringBuffer classes are Serializable, Appendable, CharSequence.
- public final class StringBuffer extends Object implements Serializable, CharSequence, Appendable.
- String buffers are safe for use by multiple threads. The methods can be synchronized wherever necessary so that all the operations on any particular instance behave as if they occur in some serial order.
- Whenever an operation occurs involving a source sequence (such as appending or inserting from a source sequence) this class synchronizes only on the string buffer performing the operation, not on the source.
- It inherits some of the methods from the Object class such as clone(), equals(), finalize(), getClass(), hashCode(), notifies(), notifyAll().

Other Methods in Java StringBuffer

These auxiliary methods are as follows:

Methods	Description	Syntax
ensureCapacity()	It is used to increase the capacity of a StringBuffer object. The new capacity will be set to either the value we specify or twice the current capacity plus two (i.e. capacity+2), whichever is larger. Here, capacity specifies the size of the buffer.	void ensureCapacity(int capacity)
appendCodePoint(int codePoint)	This method appends the string representation of the codePoint argument to this sequence.	public StringBuffer appendCodePoint(int codePoint)
charAt(int index)	This method returns the char value in this sequence at the specified index.	public char charAt(int index)

IntStream chars()	This method returns a stream of int zero-extending the char values from this sequence.	public IntStream chars()
int codePointAt(int index)	This method returns the character (Unicode code point) at the specified index.	public int codePointAt(int index)
int codePointBefore(int index)	This method returns the character (Unicode code point) before the specified index.	public int codePointBefore(int index)
int codePointCount(int beginIndex, int endIndex)	This method returns the number of Unicode code points in the specified text range of this sequence.	public int codePointCount(int beginIndex, int endIndex)
IntStream codePoints()	This method returns a stream of code point values from this sequence.	public IntStream codePoints()
void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)	In this method, the characters are copied from this sequence into the destination character array dst.	public void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)
int indexOf(String str)	This method returns the index within this string of the first occurrence of the specified substring.	public int indexOf(String str) public int indexOf(String str, int fromIndex)
int lastIndexOf(String str)	This method returns the index within this string of the last occurrence of the specified substring.	public int lastIndexOf(String str) public int lastIndexOf(String str, int fromIndex)

int offsetByCodePoints(int index, int codePointOffset)	This method returns the index within this sequence that is offset from the given index by codePointOffset code points.	public int offsetByCodePoints(int index, int codePointOffset)
void setCharAt(int index, char ch)	In this method, the character at the specified index is set to ch.	public void setCharAt(int index, char ch)
void setLength(int newLength)	This method sets the length of the character sequence.	public void setLength(int newLength)
CharSequence subSequence(int start, int end)	This method returns a new character sequence that is a subsequence of this sequence.	public CharSequence subSequence(int start, int end)
String substring(int start)	This method returns a new String that contains a subsequence of characters currently contained in this character sequence.	public String substring(int start) public String substring(int start,int end)
String toString()	This method returns a string representing the data in this sequence.	public String toString()
void trimToSize()	This method attempts to reduce storage used for the character sequence.	public void trimToSize()

Note: Above we only have discussed the most widely used methods and do keep a tight bound around them as they are widely used in programming.

Examples of the above Methods

Example 1: length() and capacity() Methods

```
import java.io.*;

class Main {
    public static void main(String[] args) {
        StringBuffer s = new StringBuffer("Apple");
        int p = s.length();
        int q = s.capacity();

        System.out.println("Length of string Apple=" + p);
        System.out.println("Capacity of string Apple=" + q);
    }
}

Output
Length of string Apple=5
Capacity of string Apple=21
```

Example 2: append()

It is used to add text at the end of the existing text. Here are a few of its forms:

```
StringBuffer append(String str)
StringBuffer append(int num)
```

Illustration:

```
import java.io.*;

class Main {
    public static void main(String[] args){
        StringBuffer s = new StringBuffer("Path");

        s.append("shala");
        System.out.println(s);
        s.append(360);
        System.out.println(s);
    }
}

Output
Pathshala
Pathshala360
```

Example 3: insert()
It is used to insert text at the specified index position. Syntax of method is mentioned below:

```
StringBuffer insert(int index, String str)
StringBuffer insert(int index, char ch)
```

Here, the index specifies the index at which point the string will be inserted into the invoking StringBuffer object.

Illustration:

```
import java.io.*;

class Main {
    public static void main(String[] args) {
        // Creating an object of StringBuffer class
        StringBuffer s = new StringBuffer("Pathshala");

        s.insert(9, "360");
        System.out.println(s);
        s.insert(0, 5);
        System.out.println(s);
        s.insert(3, true);
        System.out.println(s);
        s.insert(5, 41.35d);
        System.out.println(s);
        s.insert(8, 41.35f);
        System.out.println(s);

        Char charArr[] = { 'a', 'p', 'p', 'l', 'e' };
        s.insert(2, charArr);
        System.out.println(s);
    }
}

Output
Pathshala360
5Pathshala360
5Patruethshala360
5Patr41.35uethshala360
5Patr41.41.3535uethshala360
5Pappleatr41.41.3535uethshala360
```

Example 4: reverse( )

It can reverse the characters within a StringBuffer object using reverse( ). This method returns the reversed object on which it was called.

```
import java.io.*;
class Reverse {
    public static void main(String[] args){
        StringBuffer s = new StringBuffer("pathshala360");
        s.reverse();
        System.out.println(s);
    }
}
Output
063alahshtap
```

Example 5: delete( ) and deleteCharAt()

It can delete characters within a StringBuffer by using the methods delete( ) and deleteCharAt( ). The delete( ) method deletes a sequence of characters from the invoking object. Here, the start Index specifies the index of the first character to remove, and the end Index specifies an index one past the last character to remove. Thus, the substring deleted runs from start Index to endIndex−1. The resulting StringBuffer object is returned. The deleteCharAt( ) method deletes the character at the index specified by loc. It returns the resulting StringBuffer object.

Syntax:

```
StringBuffer delete(int startIndex, int endIndex)
StringBuffer deleteCharAt(int loc)
```

Illustration:

```
import java.io.*;
class Main {
    public static void main(String[] args) {
        StringBuffer s = new StringBuffer("Pathshala360");
        s.delete(0, 5);
        System.out.println(s);
        s.deleteCharAt(7);
        System.out.println(s);
    }
}

Output
hala3600
hala360
```

Example 6: replace()
It can replace one set of characters with another set inside a StringBuffer object by calling replace( ). The substring being replaced is specified by the indexes start Index and endIndex. Thus, the substring at start Index through endIndex–1 is replaced. The replacement string is passed in str. The resulting StringBuffer object is returned.

Syntax:

```
StringBuffer replace(int startIndex, int endIndex, String str)
```

Illustration:

```
import java.io.*;
class Main {
    public static void main(String[] args) {
        StringBuffer s = new StringBuffer("Pathshala360");
        s.replace(5, 8, "are");
        System.out.println(s);
    }
}
Output
Pathsarea360
```

Java StringBuilder Class

In Java, the StringBuilder class is a part of the java.lang package that provides a mutable sequence of characters. Unlike String (which is immutable), StringBuilder allows in-place modifications, making it memory-efficient and faster for frequent string operations.

Declaration:

```
StringBuilder sb = new StringBuilder("Initial String");
```

Key points of the StringBuilder class:

The key features of the StringBuilder class are listed below:

- StringBuilder in Java represents a mutable sequence of characters.
- String Class in Java creates an immutable sequence of characters, whereas StringBuilder creates a mutable sequence of characters, offering an alternative.
- The functionality of StringBuilder is similar to the StringBuffer class, as both provide mutable sequences of characters.
- StringBuilder does not guarantee synchronization, while StringBuffer does. It is a high-performance and low-overhead non thread non-thread-safe alternative to StringBuffer, suitable for single-threaded applications, while StringBuffer is used for synchronization in multithreaded applications.
- StringBuilder is faster than StringBuffer in most implementations.

Example: Demonstration of a StringBuilder class in Java.

```
public class Main {
    public static void main(String[] args) {
        StringBuilder sb = new StringBuilder("Pathshala");
        System.out.println("Initial StringBuilder: " + sb);

        sb.append("360!");
        System.out.println("After append: " + sb);
    }
}

Output
Initial StringBuilder: Pathshala
After append: Pathshala360!
```

Explanation:

- In the above code, we first create a StringBuilder with the object sb and put the initial string as "Pathshala".
- Then we use the append() method from the StringBuilder class to add the "360!" string in the end of the current StringBuilder object.

Syntax of Java StringBuilder Class

The syntax of the StringBuilder class is listed below:

```
public final class StringBuilder extends Object implements Serializable,
CharSequence
```

StringBuilder Class Hierarchy

```
java.lang.Object
 ↳ java.lang
    ↳ Class StringBuilder
```

Why Use StringBuilder in Java?

Using the StringBuilder class in Java has many benefits, which are listed below:

- **Mutable:** Unlike String, which creates a new object every time it's modified, StringBuilder allows you to change the string without creating new objects. This makes it more efficient for repeated modifications.
- **Faster than String:** StringBuilder doesn't create new objects for every change, it avoids unnecessary memory allocations, which makes it faster than String when performing many string manipulations.
- **More Efficient than StringBuffer:** In single-threaded environments, StringBuilder is more efficient than StringBuffer because it doesn't have the overhead of thread safety, which makes StringBuilder a better choice when we don't need synchronization.

StringBuilder vs String vs StringBuffer

The table below demonstrates the difference between String, StringBuilder and StringBuffer:

Features	String	StringBuilder	StringBuffer
Mutability	String are immutable(creates new objects on modification)	StringBuilder are mutable(modifies in place)	StringBuffer are mutable (modifies in place)
Thread-Safe	It is thread-safe	It is not thread-safe	It is thread-safe
Performance	It is slow because it creates an object each time	It is faster (no object creation)	It is slower due to synchronization overhead
use Case	Fixed, unchanging strings	Single-threaded string manipulation	Multi-threaded string manipulation

StringBuilder Constructors

The StringBuilder class provides several constructors, which are listed below:

Constructor	Description	Example
StringBuilder()	Creates an empty StringBuilder with a default initial capacity of 16.	StringBuilder strbldr = new StringBuilder();
StringBuilder(int capacity)	Creates a StringBuilder with the specified initial capacity.	StringBuilder strbldr = new StringBuilder(50);
StringBuilder(String str)	Creates a StringBuilder initialized with the contents of the given string.	StringBuilder strbldr = new StringBuilder("Test");
StringBuilder(CharSequence cs)	Creates a StringBuilder initialized with the contents of the given CharSequence object.	CharSequence strbldr = "Test"; StringBuilder sb = new StringBuilder(cs);

Example: Creating a string using the StringBuilder Constructor StringBuilder(String str).

```
public class Main {
    public static void main(String[] args) {
        StringBuilder sb = new StringBuilder("Pathshala360");

        // Converting StringBuilder to String
        String str = sb.toString();

        // Printing the String
        System.out.println(str);
    }
}

Output
Pathshala360
```

Methods of StringBuilder class

The StringBuilder class provides several methods for creating and manipulating strings, which are listed below:

Method	Description	Example
append(String str)	Appends the specified string to the end of the StringBuilder.	sb.append("Test");
insert(int offset, String)	Inserts the specified string at the given position in the StringBuilder.	sb.insert(4, "Test");
replace(int start, int end, String)	Replaces characters in a substring with the specified string.	sb.replace(6, 11, "Test");
delete(int start, int end)	Removes characters in the specified range.	sb.delete(5, 11);
reverse()	Reverses the sequence of characters in the StringBuilder.	sb.reverse();
capacity()	Returns the current capacity of the StringBuilder.	int cap = sb.capacity();
length()	Returns the number of characters in the StringBuilder.	int len = sb.length();
charAt(int index)	Returns the character at the specified index.	char ch = sb.charAt(4);
setCharAt(int index, char)	Replaces the character at the specified position with a new character.	sb.setCharAt(0, 'G');
substring(int start, int end)	Returns a new String that contains characters from the specified range.	String sub = sb.substring(0, 5);
ensureCapacity(int minimum)	Ensures the capacity of the StringBuilder is at least equal to the specified minimum.	sb.ensureCapacity(50);

deleteCharAt(int index)	Removes the character at the specified position.	sb.deleteCharAt(3);
indexOf(String str)	Returns the index of the first occurrence of the specified string.	int idx = sb.indexOf("Test");
lastIndexOf(String str)	Returns the index of the last occurrence of the specified string.	int idx = sb.lastIndexOf("Test");
toString()	Converts the StringBuilder object to a String.	String result = sb.toString();

Example: Performing different String manipulation operations using StringBuilder methods such as appending, inserting, replacing, deleting, reversing, and accessing characters.

```
public class Main {
    public static void main(String[] args) {
        StringBuilder sb = new StringBuilder("Learning");
        System.out.println("Initial StringBuilder: " + sb);

        // 1. Append a string to the StringBuilder
        sb.append(" is awesome!");
        System.out.println("After append: " + sb);

        // 2. Insert a substring at a specific position
        sb.insert(9, " Java");
        System.out.println("After insert: " + sb);

        // 3. Replace a substring with another string
        sb.replace(0, 5, "Welcome to");
        System.out.println("After replace: " + sb);

        // 4. Delete a substring from the StringBuilder
        sb.delete(8, 14);
        System.out.println("After delete: " + sb);

        // 5. Reverse the content of the StringBuilder
        sb.reverse();
        System.out.println("After reverse: " + sb);

        // 6. Get the current capacity of the StringBuilder
```

```
        int capacity = sb.capacity();
        System.out.println("Current capacity: " + capacity);

        // 7. Get the length of the StringBuilder
        int length = sb.length();
        System.out.println("Current length: " + length);

        // 8. Access a character at a specific index
        char charAt5 = sb.charAt(5);
        System.out.println("Character at index 5: " + charAt5);

        // 9. Set a character at a specific index
        sb.setCharAt(5, 'X');
        System.out.println("After setCharAt: " + sb);

        // 10. Get a substring from the StringBuilder
        String substring = sb.substring(5, 10);
        System.out.println("Substring (5 to 10): " + substring);

        // 11. Find the index of a specific substring
        sb.reverse(); // Reversing back to original order for search
        int indexOfJava = sb.indexOf("Java");
        System.out.println("Index of Java: " + indexOfJava);

        // 12. Delete a character at a specific index
        sb.deleteCharAt(5);
        System.out.println("After deleteCharAt: " + sb);

        // 13. Convert the StringBuilder to a String
        String result = sb.toString();
        System.out.println("Final String: " + result);
    }
}
```

Explanation: In the above program, we use different methods of the StringBuilder class to perform different string manipulation operations such as append(), insert(), reverse(), and delete().

Advantages

- The advantages of the StringBuilder class are listed below:
- It is more efficient than String when performing multiple string manipulations (like concatenation) since it modifies the string in place.
- It avoids creating new objects on every modification, reducing memory overhead.
- Unlike String, StringBuilder allows the modification of strings without creating new instances.
- It dynamically adjusts its capacity as needed, minimizing the need for resizing.
- Great for scenarios where strings are modified repeatedly inside loops.

Disadvantages

The disadvantages of the StringBuilder class are listed below:

- It is not synchronized, making it unsuitable for use in multi-threaded environments.
- If not used properly, StringBuilder may allocate excess memory, especially if the initial capacity is set too large.
- For multi-threaded scenarios, you must handle synchronization manually, unlike StringBuffer.