

Module 2 String Handling

String Handling: The String Constructors, String Length, Special String Operations, Character Extraction, String Comparison, Searching Strings, Modifying a String, Data Conversion Using `valueOf()`, Changing the Case of Characters Within a String, joining strings, Additional String Methods, `StringBuffer`, `StringBuilder`.

1. What are the different types of String Constructors available in Java?

The `String` class supports several constructors.

- a. To create an empty `String`
the default constructor is used.

Ex: `String s = new String();`

will create an instance of `String` with no characters in it.

- b. To create a `String` initialized by an array of characters, use the constructor shown here:

```
String(char chars[ ])
```

ex: `char chars[] = { 'a', 'b', 'c' };`

```
String s = new String(chars);
```

This constructor initializes `s` with the string "abc".

- c. To specify a subrange of a character array as an initializer using the following constructor:

```
String(char chars[ ], int startIndex, int numChars)
```

Here, `startIndex` specifies the index at which the subrange begins, and `numChars` specifies the number of characters to use. Here is an example:

```
char chars[] = { 'a', 'b', 'c', 'd', 'e', 'f' };
```

```
String s = new String(chars, 2, 3);
```

This initializes `s` with the characters `cde`.

- d. To construct a `String` object that contains the same character sequence as another `String` object using this constructor:

```
String(String strObj)
```

Here, `strObj` is a `String` object.

```
class MakeString
{
    public static void main(String args[])
    {
        char c[] = { 'J', 'a', 'v', 'a' };
        String s1 = new String(c);
    }
}
```

```
String s2 = new  
String(s1);  
System.out.println(s1);  
System.out.println(s2);  
}  
}
```

The **output** from this program is as follows:

Java

Java

As you can see, s1 and s2 contain the same string.

- e. To Construct string using byte array:

Even though Java's char type uses 16 bits to represent the basic Unicode character set, the typical format for strings on the Internet uses arrays of 8-bit bytes constructed from the ASCII character set. Because 8-bit ASCII strings are common, the String class provides constructors that initialize a string when given a byte array.

Ex: String(byte asciiChars[])

String(byte asciiChars[], int startIndex, int numChars)

The following program illustrates these constructors:

```
class SubStringCons  
{  
    public static void main(String args[])  
    {  
        byte ascii[] = {65, 66, 67, 68, 69, 70 };  
        String s1 = new String(ascii);  
        System.out.println(s1);  
        String s2 = new String(ascii, 2, 3);  
        System.out.println(s2);  
    }  
}
```

This program generates the following output:

ABCD

EF

CDE

- f. To construct a String from a StringBuffer by using the constructor shown here:

Ex: String(StringBuffer strBufObj)

- g. Constructing string using Unicode character set and is shown

here: String(int codePoints[], int startIndex, int numChars)

codePoints is an array that contains Unicode code points.

- h. Constructing string that supports the new StringBuilder class.Ex : String(StringBuilder strBuildObj)

Note:

String can be constructed by using string literals.String s1="Hello World"

String concatenation can be done using + operator. With other data type also.

String Length

1. The length of a string is the number of characters that it contains. To obtain this value, call the length() method,
2. Syntax:
int length()
3. Example
char chars[] = { 'a', 'b', 'c' }; String s = new String(chars);
System.out.println(s.length());// 3

toString()

1. Every class implements toString() because it is defined by Object.However, the default Implementation of toString() is sufficient.
2. For most important classes that you create, will want to override toString()and provide your own string representations.
String toString()
3. To implement toString(), simply return a String object that contains the human-readable string that appropriately describes an object of your class.
4. By overriding toString() for classes that you create, you allow them to be fully integrated into Java's programming environment. For example, they can be used in print() and println() statements and in concatenation expressions.
5. The following program demonstrates this by overriding toString() for the Box class:

```
class Box
{
double width; double height; double
depth;Box(double w, double h, double
```

```
d)
{
    width = w; height = h; depth = d;
}

public String toString()
{
    return "Dimensions are " + width + " by " + depth + " by " + height + "."; }
}

class toStringDemo
{
    public static void main(String args[])
    {
        Box b = new Box(10, 12, 14);
        String s = "Box b: " + b;
        System.out.println(b);
        System.out.println(s);
    }
}
```

The output of this program is shown here:

Dimensions are 10.0 by 14.0 by 12.0

Box b: Dimensions are 10.0 by 14.0 by 12.0

Character Extraction

The String class provides a number of ways in which characters can be extracted from a String object. String object can not be indexed as if they were a character array, many of the String methods employ an index (or offset) into the string for their operation. Like arrays, the string indexes begin at zero.

A. charAt()

1. description:

To extract a single character from a String, you can refer directly to an individual character via the charAt() method.

2. Syntax

char charAt(int where)

Here, where is the index of the character that you want to obtain. charAt() returns the character at the specified location.

3. example,

```
char ch;
ch = "abc".charAt(1);
// assigns the value "b" to ch.
```

Java Program

```
public class CharAtExample
{
    public static void main(String[] args)
    {
        String sampleString = "Hello, World!";
        // Get the character at index 1
        char characterAtIndex1 = sampleString.charAt(1);

        // Print the character
        System.out.println("The character at index 1 is: " + characterAtIndex1);

        // Get the character at index 7
        char characterAtIndex7 = sampleString.charAt(7);

        // Print the character
        System.out.println("The character at index 7 is: " + characterAtIndex7);

        // Example with user input
        String userInput = "Java Programming";
        int index = 5;
        char characterAtIndex = userInput.charAt(index);
        System.out.println("The character at index " + index + " in \"" +
            userInput + "\" is: " + characterAtIndex);
    }
}
```

Output:

The character at index 1 is: e

The character at index 7 is: W

The character at index 5 in "Java Programming" is: P

B. **getChars()**

1. to extract more than one character at a time, you can use the `getChars()` method.
2. Syntax

**void getChars(int sourceStart, int sourceEnd, char target[],
int targetStart)**

Here, `sourceStart` specifies the index of the beginning of the substring, `sourceEnd` specifies an index that is one past the end of the desired substring. The array that will receive the characters is specified by `target`. The index within `target` at which the substring will be copied is passed in `targetStart`.

Java Program

```
public class GetCharsExample {
    public static void main(String[] args) {
        String sampleString = "Hello, World!";

        // Define the destination character array with sufficient length
        char[] destinationArray = new char[5];

        // Copy characters from sampleString to destinationArray
        // starting from index 7 to 12 (exclusive of 12) from sampleString
        // and place them starting at index 0 in destinationArray
        sampleString.getChars(7, 12, destinationArray, 0);

        // Print the destinationArray
        System.out.print("The characters copied to destinationArray are: ");
        for (char c : destinationArray) {
            System.out.print(c);
        }
        System.out.println();
    }
}
```

Output:

The characters copied to destination Array are: World

```
class getCharsDemo
{
    public static void main(String args[])
    {
        String s = "This is a demo of the getChars method.";
        int start = 10;
        int end = 14;
        char buf[] = new char[end - start];
        s.getChars(start, end, buf, 0);
        System.out.println(buf);
    }
}
```

Here is the output of this program:

Demo

C. **getBytes()**

1. This method is called `getBytes( )`, and it uses the default character-to-byte conversions provided by the platform.

Syntax:

```
byte[ ] getBytes( )
```

Other forms of `getBytes( )` are also available.

2. `getBytes( )` is most useful when you are exporting a String value into an environment that does not support 16-bit Unicode characters. For

example,most Internet protocols and text file formats use 8-bit ASCII for all text interchange.

Java Program

```
public class GetBytesExample
{
    public static void main(String[] args)
    {
        String sampleString = "Hello, World!";

        // Get bytes from the string using the platform's default charset
        byte[] byteArray = sampleString.getBytes();

        // Print the bytes
        System.out.println("The bytes of the string are:");
        for (byte b : byteArray)
        {
            System.out.print(b + " ");
        }
        System.out.println();

        // Get bytes from the string using a specified charset (e.g., UTF-8)
        try
        {
            byte[] utf8ByteArray = sampleString.getBytes("UTF-8");
            // Print the bytes
            System.out.println("The bytes of the string in UTF-8 are:");
            for (byte b : utf8ByteArray)
            {
                System.out.print(b + " ");
            }
            System.out.println();
        }
        catch (java.io.UnsupportedEncodingException e)
        {
            System.out.println("The specified charset is not supported.");
        }
    }
}
```

Output:

The bytes of the string are:

72 101 108 108 111 44 32 87 111 114 108 100 33

The bytes of the string in UTF-8 are:

72 101 108 108 111 44 32 87 111 114 108 100 33

D. toCharArray()

If you want to convert all the characters in a String object into a characterarray, the easiest way is to call toCharArray().

It returns an array of characters for the entire

string. It has this general form:

```
char[ ] toCharArray( )
```

Java Program

```
public class ToCharArrayExample
{
    public static void main(String[] args)
    {
        String sampleString = "Hello, World!";

        // Convert the string to a character array
        char[] charArray = sampleString.toCharArray();

        // Print the characters in the array
        System.out.print("The characters in the array are: ");
        for (char c : charArray) {
            System.out.print(c + " ");
        }
        System.out.println();
    }
}
```

Output:

The characters in the array are: H e l l o , W o r l d !

String Comparison

String comparison in Java can be performed using several methods, each serving different purposes and offering varying degrees of control over how the comparison is executed. Here are some common ways to compare strings in Java:

The String class includes several methods that compare strings or substrings within strings.

a. equals ():

To compare two strings for equality, use equals().

It has this general form:

```
boolean equals(Object str)
```

Here, str is the String object being compared with the invoking String object. It returns true if the strings contain the same characters in the same order, and false otherwise. The comparison is case-sensitive.

The equals() method compares the content of two strings for equality. It is case-sensitive.

```
public class EqualsExample
{
    public static void main(String[] args)
    {
        String str1 = "Hello";
        String str2 = "Hello";
        String str3 = "hello";

        System.out.println(str1.equals(str2)); // true
        System.out.println(str1.equals(str3)); // false
    }
}
```

Output: true

false

b. equalsIgnoreCase()

To perform a comparison that ignores case differences, call equalsIgnoreCase(). When it compares two strings, it considers A-Z to be the same as a-z.

It has this general form:

```
boolean equalsIgnoreCase(String str)
```

Here, str is the String object being compared with the invoking String object. It, too, returns true if the strings contain the same characters in the same order, and false otherwise.

Example 1:

```
public class EqualsIgnoreCaseTrueExample
{
    public static void main(String[] args)
    {
        // Define two strings
        String str1 = "HelloWorld";
        String str2 = "helloworld";

        // Compare the two strings using equalsIgnoreCase()
        boolean areEqual = str1.equalsIgnoreCase(str2);

        // Print the result
        if (areEqual)
```

StringHandling

```
        {  
            System.out.println("The strings are equal.");  
        }  
    else  
    {  
        System.out.println("The strings are not equal.");  
    }  
}  
}
```

Output: The strings are equal.

Example2 :

```
public class EqualsIgnoreCaseFalseExample  
{  
    public static void main(String[] args)  
    {  
        // Define two strings  
        String str1 = "HelloWorld";  
        String str2 = "HelloJava";  
        // Compare the two strings using equalsIgnoreCase()  
        boolean areEqual = str1.equalsIgnoreCase(str2);  
        // Print the result  
        if (areEqual)  
        {  
            System.out.println("The strings are equal (ignoring case).");  
        }  
    else  
    {  
        System.out.println("The strings are not equal.");  
    }  
}
```

```
}  
  
}
```

Output: The strings are not equal.

C. `regionMatches()`

1. The `regionMatches()` method compares a specific region inside a string with another specific region in another string. There is an overloaded form that allows you to ignore case in such comparisons.

2. Syntax:

`boolean regionMatches(int startIndex, String str2, int str2StartIndex, int numChars)`

`boolean regionMatches(boolean ignoreCase, int startIndex, String str2, int str2StartIndex, int numChars)`

3. For both versions, `startIndex` specifies the index at which the region begins within the invoking String object.
The String being compared is specified by `str2`. The index at which the comparison will start within `str2` is specified by `str2 StartIndex`. The length of the substring being compared is passed in `numChars`.
4. In the second version, if `ignoreCase` is true, the case of the characters is ignored. Otherwise, case is significant.

The `regionMatches()` method in Java is used to compare a specific region of one string with a region of another string. It allows you to specify the starting index and the length of the region to compare.

Here is a simple program that demonstrates the use of the `regionMatches()` method. I'll provide two versions: one where `regionMatches()` returns true and another where it returns false.

Program 1: `regionMatches()` Returns true

```
public class RegionMatchesTrueExample  
{  
    public static void main(String[] args)  
    {  
        // Define two strings  
        String str1 = "HelloWorld";  
        String str2 = "World";  
        // Compare regions of the two strings  
        boolean match = str1.regionMatches(5, str2, 0, 5);
```

```
// Print the result
if (match)
{
    System.out.println("The regions match.");
}
else
{
    System.out.println("The regions do not match.");
}
}
```

Output: The region matches

Program 2: regionMatches() Returns false

```
public class RegionMatchesFalseExample
{
    public static void main(String[] args)
    {
        // Define two strings
        String str1 = "HelloWorld";
        String str2 = "Java";
        // Compare regions of the two strings
        boolean match = str1.regionMatches(5, str2, 0, 4);
        // Print the result
        if (match)
        {
            System.out.println("The regions match.");
        }
        else
        {

```

```
        System.out.println("The regions do not match.");
    }
}
}
```

Output: The region do not match

d. **startsWith() and endsWith()**

- The **startsWith()** method determines whether a given String begins with aspecified string.
 - **endsWith()** determines whether the String in question ends with aspecified string.
 - **Syntax**
 - `boolean startsWith(String str);`
 - `boolean endsWith(String str);`
 - Here, str is the String being tested.
 - If the string matches, true is returned.
- Otherwise, false is returned.

The **startsWith()** method in Java is used to check if a string begins with a specified prefix. Here are two simple programs that demonstrate the use of the **startsWith()** method: one where the method returns true and another where it returns false.

Program 1:

```
public class StartsWithTrueExample
{
    public static void main(String[] args)
    {
        // Define the string
        String str = "HelloWorld";

        // Define the prefix
        String prefix = "Hello";

        // Check if the string starts with the specified prefix
        boolean result = str.startsWith(prefix);

        // Print the result
        if (result)
        {
            System.out.println("The string starts with the prefix.");
        }
    }
}
```

```
    }  
    else  
    {  
        System.out.println("The string does not start with the prefix.");  
    }  
}  
}
```

Output: The string starts with the prefix

Program 1: StartsWithTrueExample

Define String: str is "HelloWorld".

Define Prefix: prefix is "Hello".

Check Prefix: str.startsWith(prefix) returns true because str starts with "Hello".

Print Result: The program prints "The string starts with the prefix."

Program2 :

public class StartsWithFalseExample

```
{  
    public static void main(String[] args)  
    {  
        // Define the string  
        String str = "HelloWorld";  
        // Define the prefix  
        String prefix = "World";  
        // Check if the string starts with the specified prefix  
        boolean result = str.startsWith(prefix);  
        // Print the result  
        if (result)  
        {  
            System.out.println("The string starts with the prefix.");  
        }  
    }  
}
```

```
        else
        {
            System.out.println("The string does not start with the prefix.");
        }
    }
}
```

Program 2: StartsWithFalseExample

Output: The string does not start with the prefix

Define String: str is "HelloWorld".

Define Prefix: prefix is "World".

Check Prefix: str.startsWith(prefix) returns false because str does not start with "World".

Print Result: The program prints "The string does not start with the prefix."

endsWith () method:

The endsWith() method in Java is used to check if a string ends with a specified suffix. Here are two simple programs that demonstrate the use of the endsWith() method: one where the method returns true and another where it returns false.

Program 1: endsWith() Returns true

```
public class EndsWithTrueExample
{
    public static void main(String[] args)
    {
        // Define the string
        String str = "HelloWorld";

        // Define the suffix
        String suffix = "World";

        // Check if the string ends with the specified suffix
        boolean result = str.endsWith(suffix);

        // Print the result
        if (result)
        {
```

```
        System.out.println("The string ends with the suffix.");
    }
    else
    {
        System.out.println("The string does not end with the suffix.");
    }
}
}
```

Output: The string ends with the suffix

Program 1: EndsWithTrueExample

Define String: str is "HelloWorld".

Define Suffix: suffix is "World".

Check Suffix: str.endsWith(suffix) returns true because str ends with "World".

Print Result: The program prints "The string ends with the suffix."

Program 2: endsWith() Returns false

```
public class EndsWithFalseExample
{
    public static void main(String[] args)
    {
        // Define the string
        String str = "HelloWorld";

        // Define the suffix
        String suffix = "Hello";

        // Check if the string ends with the specified suffix
        boolean result = str.endsWith(suffix);

        // Print the result
        if (result)
        {
            System.out.println("The string ends with the suffix.");
        }
    }
}
```

```
    }  
    else  
    {  
        System.out.println("The string does not end with the suffix.");  
    }  
}  
}
```

Output: The string does not end with the suffix

Program 2: EndsWithFalseExample

Define String: str is "HelloWorld".

Define Suffix: suffix is "Hello"

Check Suffix: str.endsWith(suffix) returns false because str does not end with "Hello".

Print Result: The program prints "The string does not end with the suffix."

e. equals() Versus ==

It is important to understand that the equals() method and the == operator perform two different operations.

the equals() method compares the characters inside a String object.

The == operator compares two object references to see whether they refer to the same instance.

```
class EqualsNotEqualTo  
{  
    public static void main(String args[])  
    {  
        String s1 = "Hello";  
        String s2 = new String(s1);  
        System.out.println(s1 + " equals " + s2 + " -> " + s1.equals(s2));  
        System.out.println(s1 + " == " + s2 + " -> " + (s1 == s2));  
    }  
}
```

equals() Method

The equals() method is a method provided by the Object class, which is overridden by many classes including String. It compares the content of two objects to determine if they are logically equal. For strings, equals() compares the actual contents of the strings, character by character.

```
String str1 = "Hello";  
String str2 = "Hello";  
String str3 = new String("Hello");
```

```
System.out.println(str1.equals(str2)); // true
System.out.println(str1.equals(str3)); // true
```

== Operator

The == operator in Java checks whether two object references point to the same memory location, i.e., whether they refer to the same object. When applied to strings, it checks whether two string variables refer to the same string object in memory.

```
String str1 = "Hello";
String str2 = "Hello";
String str3 = new String("Hello");
```

```
System.out.println(str1 == str2); // true (because of string literal pooling)
System.out.println(str1 == str3); // false (different memory locations)
```

Note:

equals() compares the contents of two strings.

== compares the references of two string objects.

f. compareTo()

- Sorting applications, you need to know which is less than, equal to, or greater than the next.
- A string is less than another if it comes before the other in dictionary order. A string is greater than another if it comes after the other in dictionary order. The String method compareTo() serves this purpose.
- It has this general form:
 - int compareTo(String str)
 - Here, str is the String being compared with the invoking String. The result of the comparison is returned and is interpreted,
 - Less than zero when invoking string is less than str.
 - Greater than zero when invoking string is greater than str.
 - Zero The two strings are equal.

The compareTo() method in Java is used to compare two strings lexicographically. It returns an integer value based on the comparison result:

If the first string is lexicographically less than the second string, it returns a negative integer.

If the strings are equal, it returns 0.

If the first string is lexicographically greater than the second string, it returns a positive integer.

Here's a simple example demonstrating the use of the compareTo() method:

Program1:

```
public class CompareToExample
{
    public static void main(String[] args)
    {
        String str1 = "apple";
        String str2 = "banana";
        int result = str1.compareTo(str2);
        if (result < 0)
        {
            System.out.println(str1 + " comes before " + str2 + " lexicographically.");
        }
        else if (result > 0)
        {
            System.out.println(str1 + " comes after " + str2 + " lexicographically.");
        }
        else
        {
            System.out.println(str1 + " and " + str2 + " are equal lexicographically.");
        }
    }
}
```

In this example, "apple" is lexicographically less than "banana", so the compareTo() method returns a negative integer. The program then prints that

Output: "apple" comes before "banana" lexicographically.

"Lexicographically" refers to the ordering of strings based on the alphabetical order of their characters. When comparing strings lexicographically:

The comparison starts from the first character of each string.

If the characters at the corresponding positions are equal, the comparison continues to the next characters.

If the characters differ, the string with the lower Unicode value at that position is considered "smaller" in lexicographical order.

If one string ends before the other and all corresponding characters are equal, the shorter string is considered "smaller".

For example:

"apple" comes before "banana" lexicographically because 'a' comes before 'b'.

"apple" comes after "ant" lexicographically because 'p' comes after 'n', even though 'p' comes after 'n'.

"apple" and "apple" are considered equal lexicographically because they are identical.

Lexicographical comparison is not limited to alphabetic characters; it can be applied to any sequence of characters, including digits and special characters, based on their Unicode values.

5. Searching String

A.indexOf() and lastIndexOf()

1. `indexOf( )` Searches for the first occurrence of a character or substring.
2. `lastIndexOf( )` Searches for the last occurrence of a character or substring.
3. These two methods are overloaded in several different ways
4. return the index at which the character or substring was found, or -1 on failure.
5. To search for the first occurrence of a character, `int indexOf(int ch)`
6. To search for the last occurrence of a character, `int lastIndexOf(int ch)` Here, `ch` is the character being sought
7. To search for the first or last occurrence of a substring, use `int indexOf(String str)` `int lastIndexOf(String str)` Here, `str` specifies the substring.
8. You can specify a starting point for the search using these forms:
`int indexOf(int ch, int startIndex)`
`int lastIndexOf(int ch, int startIndex)`
9. `int indexOf(String str, int startIndex)` `int lastIndexOf(String str, int startIndex)`
Here, `startIndex` specifies the index at which point the search begins.
10. For `indexOf( )`, the search runs from `startIndex` to the end of the string. For `lastIndexOf( )`, the search runs from `startIndex` to zero. The following example shows how to use the various index methods to search inside of Strings:

The `indexOf()` method is a function commonly found in programming languages, particularly in languages like JavaScript. It is used to determine the index of the first occurrence of a specified value within a string or an array. Here's a brief explanation of how it works:

For Strings: In the context of strings, `indexOf()` searches the string for a specified substring and returns the position of the first occurrence of that substring. If the substring is not found, it returns -1.

```
let str = "Hello, world!";
```

```
console.log(str.indexOf("world")); // Output: 7
```

```
console.log(str.indexOf("Earth")); // Output: -1 (not found)
```

or Arrays: In the context of arrays, `indexOf()` searches the array for a specified item and returns the index of the first occurrence of that item. If the item is not found, it returns -1.

JavaScript

```
let arr = [1, 2, 3, 4, 5];
```

```
console.log(arr.indexOf(3)); // Output: 2
```

```
console.log(arr.indexOf(6)); // Output: -1 (not found)
```

Java Program

```
public class IndexOfDemo
```

```
{
```

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

```
{
```

```
    // Example with a String
```

```
    String str = "Hello, world!";
```

```
    int stringIndex = str.indexOf("world");
```

```
    System.out.println("Index of 'world' in string: " + stringIndex); // Output: 7
```

```
    // Example with an Array
```

```
    int[] arr = {1, 2, 3, 4, 5};
```

```
    int arrayIndex = indexOf(arr, 3);
```

```
    System.out.println("Index of 3 in array: " + arrayIndex); // Output: 2
```

```
}

// Helper method to find the index of an element in an array

public static int indexOf(int[] array, int value)

{
    for (int i = 0; i < array.length; i++)
    {
        if (array[i] == value)
        {
            return i;
        }
    }

    return -1; // Value not found in the array
}}
```

Output:

Index of 'world' in string : 7

Index of 3 in array: 2

The `lastIndexOf()` method is similar to the `indexOf()` method but searches for the last occurrence of a specified value instead of the first occurrence. Here's a small Java program demonstrating the use of the `lastIndexOf()` method for both a string and an array.

```
public class LastIndexOfDemo
{
    public static void main(String[] args)
    {
        // Example with a String
        String str = "Hello, world! Hello, everyone!";
        int stringLastIndex = str.lastIndexOf("Hello");

        System.out.println("Last index of 'Hello' in string: " + stringLastIndex);

        // Output: 14
    }
}
```

```
// Example with an Array

int[] arr = {1, 2, 3, 4, 3, 2, 1};

int arrayLastIndex = lastIndexOf(arr, 3);

System.out.println("Last index of 3 in array: " + arrayLastIndex); // Output: 4
}

// Helper method to find the last index of an element in an array
public static int lastIndexOf(int[] array, int value)
{
    for (int i = array.length - 1; i >= 0; i--)
    {
        if (array[i] == value)
        {
            return i;
        }
    }

    return -1; // Value not found in the array
}

}
```

Output:

Last index of 'Hello' in string: 14

Last index of 3 in array: 4

5. Modifying a String

String objects are immutable, whenever you want to modify a String, you must either copy it into a StringBuffer or StringBuilder, or use one of the following String methods, which will construct a new copy of the string with your modifications complete.

The substring() method in Java is used to extract a part of a string. It has two overloaded versions:

substring(int beginIndex): Returns a new string that is a substring of the original string, starting from the specified index to the end of the string.

substring(int beginIndex, int endIndex): Returns a new string that is a substring of the original string, starting from the specified index to the specified end index (exclusive).

Here's a small Java program that the use of both versions of the substring()

method:

Java Program:

```
public class SubstringDemo
{
    public static void main(String[] args)
    {
        // Original String
        String str = "Hello, world!";
        // Using substring(beginIndex)
        String subStr1 = str.substring(7);
        System.out.println("Substring from index 7: " + subStr1); // Output: "world!"
        // Using substring(beginIndex, endIndex)
        String subStr2 = str.substring(0, 5);
        System.out.println("Substring from index 0 to 5: " + subStr2); // Output: "Hello"
        String subStr3 = str.substring(7, 12);
        System.out.println("Substring from index 7 to 12: " + subStr3); // Output: "world"
    }
}
```

Output:

Substring from index 7: world!

Substring from index 0 to 5: Hello

Substring from index 7 to 12: world

The concat() method in Java is used to concatenate, or join together, two strings. This method appends the specified string to the end of the current string.

Here's a small Java program that demonstrates the use of the concat() method:

```
public class ConcatDemo
{
    public static void main(String[] args)
    {
        // Original Strings
```

```
String str1 = "Hello, ";
String str2 = "world!";

// Using concat() method

String concatenatedString = str1.concat(str2);

// Printing the result

System.out.println("Concatenated String: " + concatenatedString); // Output: "Hello,
world!"

// Additional example

String str3 = " Have a great day!";

String finalString = concatenatedString.concat(str3);

// Printing the final result

System.out.println("Final Concatenated String: " + finalString); // Output: "Hello,
world! Have a great day!"

}

}
```

Output: Concatenated String: Hello, world!

Final Concatenated String: Hello, world! Have a great day!

replace()

The replace() method has two forms.

The first replaces all occurrences of one character in the invoking string with another character.

Syntax:

String replace(char original, char replacement)

Here, original specifies the character to be replaced by the characters specified by replacement. The resulting string is returned.

Example

String s = "Hello".replace('l', 'w'); puts the string
"Hewwo" into s.

The second form of replace() replaces one character sequence with another. It has this general form:

String replace(CharSequence original, CharSequence replacement)

```
public class SimpleReplaceExample
```

```
{
```

```
public static void main(String[] args)
{
    // Define the original string
    String originalString = "Hello, World!";

    // Define the substring to be replaced
    String toReplace = "World";

    // Define the replacement substring
    String replacement = "Universe";

    // Use the replace() method to create a new string with the replacements
    String modifiedString = originalString.replace(toReplace, replacement);

    // Display the original and modified strings
    System.out.println("Original string: " + originalString);
    System.out.println("Modified string: " + modifiedString);
}
}
```

Output:

Original string: Hello, World!

Modified string: Hello, Universe!

trim()

The trim() method returns a copy of the invoking string from which any leading and trailing whitespace has been removed.

Syntax:

String trim() Example:

String s = " Hello World".trim();

This puts the string "Hello World" into s.

The trim() method is quite useful when you process user commands.

The trim() method in Java is used to remove leading and trailing whitespace from a string. It is a simple and commonly used method for cleaning up strings before processing them further.

Here's a simple Java program that

the use of the trim() method:

Java Program:

```
public class TrimExample
{
    public static void main(String[] args)
    {
        // Define a string with leading and trailing spaces
        String originalString = " Hello, World! ";

        // Display the original string (including spaces)
        System.out.println("Original string: \"" + originalString + "\"");

        // Use the trim() method to remove leading and trailing spaces
        String trimmedString = originalString.trim();

        // Display the trimmed string
        System.out.println("Trimmed string: \"" + trimmedString + "\"");
    }
}
```

Output:

Original string: " Hello, World! "

Trimmed string: "Hello, World!"

Data Conversion

The `valueOf()` method converts data from its internal format into a human-readable form.

1. It is a static method that is overloaded within `String` for all of Java's built-in types so that each type can be converted properly into a string.

2. `valueOf()` is also overloaded for type `Object`, so an object of any class type you create can also be used as an argument

Syntax:

```
static String valueOf(double num)
static String valueOf(long num)
static String valueOf(Object ob)
static String valueOf(char chars[ ])
```

`valueOf()` is called when a string representation of some other type of data is needed. example, during concatenation operations.

Any object that you pass to `valueOf()` will return the result of a call to the object's `toString()` method.

There is a special version of `valueOf()` that allows you to specify a subset of a char array.

Syntax:

```
static String valueOf(char chars[ ], int startIndex, int numChars)
```

Here, chars is the array that holds the characters, startIndex is the index into the array of characters at which the desired substring begins, and numChars specifies the length of the substring.

```
public class SimpleValueOfExample
{
    public static void main(String[] args)
    {
        // Integer to String
        int intValue = 100;
        String intString = String.valueOf(intValue);
        System.out.println("Integer to String: " + intString);

        // Double to String
        double doubleValue = 50.25;
        String doubleString = String.valueOf(doubleValue);
        System.out.println("Double to String: " + doubleString);

        // Boolean to String
        boolean boolValue = false;
        String boolString = String.valueOf(boolValue);
        System.out.println("Boolean to String: " + boolString);

        // Char to String
        char charValue = 'Z';
        String charString = String.valueOf(charValue);
        System.out.println("Char to String: " + charString);
    }
}
```

Output:

```
Integer to String: 100
```

Double to String: 50.2

Boolean to String: false

Char to String: Z

Changing Case of Characters

toLowerCase()

1. converts all the characters in a string from uppercase to lowercase.
2. This method return a String object that contains the lowercaseequivalent of the invoking String.
3. Non alphabetical characters, such as digits, are unaffected.

Syntax

String toLowerCase()

Java Program

```
public class ToLowerCaseExample
{
    public static void main(String[] args)
    {
        // Define the original string
        String originalString = "Hello, World!";

        // Use the toLowerCase() method to convert the string to lowercase
        String lowerCaseString = originalString.toLowerCase();

        // Display the original and the lowercase strings
        System.out.println("Original string: " + originalString);
        System.out.println("Lowercase string: " + lowerCaseString);
    }
}
```

Output:

Original string: Hello, World!

Lowercase string: hello, world!

A. toUpperCase()

1. converts all the characters in a string from lowercase touppercase.
2. This method return a String object that contains the uppercaseequivalent of the invoking String.

3. Non alphabetical characters, such as digits, are unaffected.

Syntax

String toUpperCase()

In Java, the toUpperCase() method is used to convert all characters in a string to uppercase.

```
public class UpperCaseExample
{
    public static void main(String[] args)
    {
        // Create a string
        String str = "hello world";

        // Convert all characters to uppercase
        String upperCaseStr = str.toUpperCase();

        // Print the original string and the uppercase version
        System.out.println("Original String: " + str);
        System.out.println("Uppercase String: " + upperCaseStr);
    }
}
```

Output:

Original String: hello world

Uppercase String: HELLO WORLD

StringBuffer

StringBuffer is a peer class of String that provides much of the functionality of strings. As you know, String represents fixed-length, immutable character sequences.

StringBuffer in Java is a class that represents a mutable sequence of characters. It's similar to the String class, but unlike strings, objects of type StringBuffer can be modified after they are created.

This makes StringBuffer suitable for situations where you need to modify strings frequently without creating new objects each time.

```
public class StringBufferExample
{
    public static void main(String[] args)
    {
        // Create a StringBuffer object
        StringBuffer stringBuffer = new StringBuffer("Hello");

        // Append text to the StringBuffer
        stringBuffer.append(" World");

        // Insert text at a specific position
        stringBuffer.insert(5, ", ");

        // Replace text
        stringBuffer.replace(6, 11, "there");

        // Delete characters
        stringBuffer.delete(11, 12);

        // Print the modified string
        System.out.println(stringBuffer.toString()); // Output: Hello, there World
    }
}
```

In this example:

We create a StringBuffer object initialized with the string "Hello".

We then append " World" to it.

We insert ", " at index 5.

We replace characters from index 6 to 10 with "there".

We delete the character at index 11.

Finally, we print the modified string.

StringBuffer provides methods for appending, inserting, replacing, and deleting characters in the sequence, making it a versatile class for string manipulation. Additionally, StringBuffer is synchronized, which means it's safe for use in

multithreaded environments. If you don't need thread safety, you can use the `StringBuilder` class, which provides similar functionality but is not synchronized, resulting in better performance in single-threaded scenarios

Output: Hello, there World

For example, in the previous `StringBuffer` example, we demonstrated how various methods (`append()`, `insert()`, `replace()`, `delete()`) can modify the contents of the `StringBuffer` object directly, altering its state without creating new objects.

Here's an example of how you can use `StringBuffer` in Java:

`StringBuffer` represents growable and writeable character sequences.

`StringBuffer` may have characters and substrings inserted in the middle or appended to the end.

`StringBuffer` will automatically grow to make room for such additions and often has more characters pre allocated than are actually needed, to allow room for growth.

StringBuffer Constructors

`StringBuffer` defines these four constructors:

`StringBuffer( )`

`StringBuffer(int size)`

`StringBuffer(String str)`

`StringBuffer(CharSequence chars)`

- The default constructor (the one with no parameters) reserves room for 16 characters without reallocation.
- The second version accepts an integer argument that explicitly sets the size of the buffer.
- The third version accepts a `String` argument that sets the initial contents of the `StringBuffer` object and reserves room for 16 more characters without reallocation.
- `StringBuffer` allocates room for 16 additional characters when no specific buffer length is requested, because reallocation is a costly process in terms of time.

length() and capacity()

- The current length of a `StringBuffer` can be found via the `length( )` method, while the total allocated capacity can be found through the `capacity( )` method.

Syntax

`int length( )`

`int capacity( )`

Java Program on `length( )`

```
public class LengthExample
{
    public static void main(String[] args)
    {
        // Create a string
        String str = "Hello, World!";
        // Get the length of the string
        int length = str.length();
        // Print the length of the string
        System.out.println("Length of the string: " + length);
        // Output: Length of the string: 13
    }
}
```

Output:

Length of the string: 13

Java Program on capacity()

```
public class CapacityExample
{
    public static void main(String[] args)
    {
        // Create a StringBuffer with an initial capacity of 16
        StringBuffer stringBuffer = new StringBuffer();
        // Append some text to increase the capacity
        stringBuffer.append("Hello World");
        // Get the current capacity
        int capacity = stringBuffer.capacity();
        // Print the current capacity
        System.out.println("Current Capacity: " + capacity);
        // Output: Current Capacity: 16
    }
}
```

```
}
```

We create a StringBuffer object without specifying an initial capacity. By default, it initializes with a capacity of 16.

We append the string "Hello World" to the StringBuffer, which increases its capacity if needed to accommodate the new data.

We use the capacity() method of the StringBuffer class to get the current capacity.

The current capacity is stored in the variable capacity.

We then print the current capacity using System.out.println().

Output: Current Capacity: 16

ensureCapacity()

- If you want to pre allocate room for a certain number of characters after a StringBuffer has been constructed, you can use ensureCapacity() to set the size of the buffer.
- This is useful if you know in advance that you will be appending a large number of small strings to a StringBuffer.

Syntax

```
void ensureCapacity(int capacity)
```

Here, capacity specifies the size of the buffer.

The ensureCapacity() method in Java is used to ensure that the StringBuffer or StringBuilder has a minimum capacity specified by the parameter. If the current capacity of the buffer is less than the specified minimum capacity, the buffer's capacity is increased to the specified minimum capacity. Otherwise, the capacity remains unchanged.

Java Program

```
public class EnsureCapacityExample
{
    public static void main(String[] args)
    {
        // Create a StringBuffer with an initial capacity of 10
        StringBuffer stringBuffer = new StringBuffer(10);

        // Print the initial capacity
        System.out.println("Initial Capacity: " + stringBuffer.capacity()); // Output: Initial
        Capacity: 10

        // Ensure that the capacity is at least 30
        stringBuffer.ensureCapacity(30);
```

```
// Print the current capacity after ensuring capacity
System.out.println("Current Capacity after ensuring: " + stringBuffer.capacity());
// Output: Current Capacity after ensuring: 30
}
```

Explanation

We create a StringBuffer object with an initial capacity of 10.

We print the initial capacity of the StringBuffer.

We use the ensureCapacity() method to ensure that the capacity is at least 20.

We print the current capacity after ensuring the capacity.

Output:

Initial Capacity: 10

Current Capacity after ensuring: 20

setLength()

To set the length of the buffer with in a StringBufferobject,

Syntax:

```
void setLength(int len)
```

Here, len specifies the length of the buffer. This value must be nonnegative.

When you increase the size of the buffer, null characters are added to the end of the existingbuffer.

If you call setLength() with a value less than the current value returned by length(), then thecharacters stored beyond the new length will be lost.

In Java, the setLength() method is used to set the length of a StringBuffer or StringBuilder object to a specified value. If the new length is greater than the current length, the additional characters are filled with null characters ('\0'). If the new length is smaller than the current length, the content of the buffer is truncated to fit the new length.

Java Program

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

```
// Create a StringBuffer with some initial content

StringBuffer stringBuffer = new StringBuffer("Hello, World!");

// Print the initial content and length

System.out.println("Initial Content: " + stringBuffer);

System.out.println("Initial Length: " + stringBuffer.length());

// Set the length to 5

stringBuffer.setLength(5);

// Print the content and length after setting the length

System.out.println("Content after setting length: " + stringBuffer);

System.out.println("Length after setting length: " + stringBuffer.length());

}

}
```

Explanation:

We create a StringBuffer object with the initial content "Hello, World!".

We print the initial content and length of the StringBuffer.

We use the setLength() method to set the length of the StringBuffer to 5.

We print the content and length of the StringBuffer after setting the length.

Output:

Initial Content: Hello, World!

Initial Length: 13

Content after setting length: Hello

Length after setting length: 5

charAt() and setCharAt()

- a. The value of a single character can be obtained from a StringBuffer via the charAt() method. You can set the value of a character within a StringBuffer

using setCharAt().

b. Syntax

char charAt(int where)

void setCharAt(int where, char ch)

c. For charAt(), where specifies the index of the character being obtained.

d. For setCharAt(), where specifies the index of the character being set, and ch specifies the new value of that character.

In Java, the charAt() method is used to retrieve the character at a specified index in a string. It belongs to the String class. The indexing starts from 0, meaning the first character is at index 0, the second character is at index 1, and so on. Here's how you can use it:

```
public class CharAtExample
{
    public static void main(String[] args)
    {
        // Create a string
        String str = "Hello, World!";

        // Get the character at index 7
        char ch = str.charAt(7);

        // Print the character
        System.out.println("Character at index 7: " + ch);
        // Output: Character at index 7: W
    }
}
```

Output: Character at index 7: W

setChar() method

The setCharAt() method, on the other hand, is a method of the StringBuffer class (or StringBuilder in case of non-synchronized operations), which is used to modify the character at a specified index in a string. It allows you to replace the character at the given index with a new character. Here's how you can use it:

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

```
// Create a StringBuffer
StringBuffer stringBuffer = new StringBuffer("Hello, World!");
// Set the character at index 7 to 'G'
stringBuffer.setCharAt(7, 'G');
// Print the modified string
System.out.println("Modified string: " + stringBuffer); // Output: Modified string:
Hello, Gold!
}
}
```

Output: Modified string: Hello, Gold!

getChars() method

- a. To copy a substring of a StringBuffer into an array, use the getChars()

method.Syntax

Syntax

```
void getChars(int sourceStart, int sourceEnd, char target[ ], int targetStart)
```

Here, sourceStart specifies the index of the beginning of the substring, and sourceEnd specifies an index that is one past the end of the desired substring.

- b. This means that the substring contains the characters from sourceStart through sourceEnd-1.

- c. The array that will receive the characters is specified by target.

The index within target which the substring will be copied is passed in targetStart.

- d. Care must be taken to assure that the target array is large enough to hold the number of characters in the specified substring.

```
public class Main
```

```
{
```

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

```
    {
```

```
        StringBuffer exampleBuffer = new StringBuffer("Hello, World!");
```

```
        int sourceStart = 0;
```

```
        int sourceEnd = 5;
```

```
        char[] target = new char[5];
```

```
int targetStart = 0;

exampleBuffer.getChars(sourceStart, sourceEnd, target, targetStart);

// Convert the target char array to a string and print it

System.out.println(new String(target)); // Output: Hello

}

}
```

Output: Hello

StringBuffer getChars Method:

The method `getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)` copies characters from the StringBuffer into the destination character array `dst`.

`srcBegin`: The starting index in the source StringBuffer.

`srcEnd`: The ending index in the source StringBuffer (exclusive).

`dst`: The destination character array where characters are copied.

`dstBegin`: The starting index in the destination array where copying begins.

Defines a StringBuffer **main Method**:

named `exampleBuffer` with the content "Hello, World!".

Specifies `sourceStart` as 0 and `sourceEnd` as 5 to copy the first five characters ("Hello").

Initializes a target character array `target` with a length of 5 to hold the copied characters.

Specifies `targetStart` as 0 to start copying at the beginning of the target array.

Calls the `getChars` method to copy the specified characters from `exampleBuffer` to `target`.

Converts the target character array to a string and prints it.

append()

The `append()` method concatenates the string representation of any other type of data to the end of the invoking StringBuffer object. It has several overloaded versions. Here are a few of its forms:

StringBuffer append(String str)

StringBuffer append(int num)

**StringBuffer append(Object
obj)**

1. The result is appended to the current StringBuffer object.
2. The buffer itself is returned by each version of append().
3. This allows subsequent calls to be chained together, as shown in the following example:

```
class appendDemo
{
    public static void main(String args[])
    {
        String s;
        int a = 42;
        StringBuffer sb = new StringBuffer(40);
        s = sb.append("a =
").append(a).append("!").toString();
        System.out.println(s);
    }
}
```

Output

a = 42!

insert()

1. The insert() method inserts one string in to another.
2. It is overloaded to accept values of all the simple types, plus Strings, Objects, and CharSequences.
3. Like append(), it calls String.valueOf() to obtain the string representation of the value it is called with.
4. This string is then inserted into the invoking StringBuffer object.
5. These are a few of its forms:

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

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

```
StringBuffer insert(int index, Object obj)
```

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

6. The following sample program inserts “like” between “I” and “Java”:

```
{  
    public static void main(String args[])  
    {  
        StringBuffer sb = new StringBuffer("I Java!");  
        sb.insert(2, "like ");  
        System.out.println(sb);  
    }  
}
```

Output:

I like Java!

reverse()

You can reverse the characters within a StringBuffer object using reverse(), shown here:

This method returns the reversed object on which it was called. The following program demonstrates reverse()

```
class ReverseDemo  
{  
    public static void main(String args  
[]) {  
        StringBuffer s = new  
StringBuffer("abcdef");  
        System.out.println(s); s  
.reverse();  
        System.out.println(s  
); }  
}
```

Output

:

abcdef

fedcba

delete() and deleteCharAt()

You can delete characters within a StringBuffer by using the methods delete() and deleteCharAt().

Syntax:

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

The delete() method deletes a sequence of characters from the invoking object.

Here, startIndex specifies the index of the first character to remove, and endIndex specifies an index one past the last character to remove.

Thus, the substring deleted runs from startIndex 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.

```
// Demonstrate delete() and deleteCharAt()  
  
class deleteDemo  
{  
    public static void main(String args[])  
    {  
  
        StringBuffer sb = new StringBuffer("This is a test.");  
  
        sb.delete(4, 7);  
  
        System.out.println("After delete: " + sb);  
  
        sb.deleteCharAt(0);  
  
        System.out.println("After deleteCharAt: " + sb);  
    }  
}
```

Output:

After delete: This a test.

After deleteCharAt: his a
test.

replace()

- a. You can replace one set of characters with another set inside a StringBuffer object by calling `replace()`.
- b. Syntax

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

The substring being replaced is specified by the indexes `startIndex` and `endIndex`.

- c. Thus, the substring at `startIndex` through `endIndex-1` is replaced. The replacement string is passed in `str`.

The resulting StringBuffer object is

returned.

```
class replaceDemo
{
    public static void main(String args[])
    {
        StringBuffer sb = new StringBuffer("This is a
        test."); sb.replace(5, 7, "was");
        System.out.println("After replace: " + sb);
    }
}
```

Here is the output:

After replace: This was a test.

substring()

1. It has the following two forms:
Syntax
`String substring(int startIndex)`
`String substring(int startIndex, int endIndex)`
2. The first form returns the substring that starts at `startIndex` and runs to the end of the invoking StringBuffer object.
3. The second form returns the substring that starts at `startIndex` and runs through `endIndex-1`.

These methods work just like those defined for String that were described earlier.

Difference between StringBuffer and StringBuilder.

1. J2SE 5 adds a new string class to Java's already powerful string handling capabilities. This new class is called `StringBuilder`.
2. It is identical to `StringBuffer` except for one important difference: it is

notsynchronized, which means that it is not thread-safe.

3. The advantage of `StringBuilder` is faster performance. However, in cases in which you are using multithreading, you must use `StringBuffer` rather than `StringBuilder`.

Additional Methods in String which was included in Java 5

1. **`int codePointAt(int i)`**

Returns the Unicode code point at the location specified by `i`.

2. **`int codePointBefore(int i)`**

Returns the Unicode code point at the location that precedes that specified by `i`.

3. **`int codePointCount(int start, int end)`**

Returns the number of code points in the portion of the invoking `String` that are between `start` and `end-1`.

4. **`boolean contains(CharSequence str)`**

Returns `true` if the invoking object contains the string specified by `str`. Returns `false`, otherwise.

5. **`boolean contentEquals(CharSequence str)`**

Returns `true` if the invoking string contains the same string as `str`. Otherwise, returns `false`.

6. **`boolean contentEquals(StringBuffer str)`**

Returns `true` if the invoking string contains the same string as `str`. Otherwise, returns `false`.

7. **`static String format(String fmtstr, Object ... args)`**

Returns a string formatted as specified by `fmtstr`.

8. **`static String format(Locale loc, String fmtstr, Object ... args)`**

Returns a string formatted as specified by `fmtstr`.

9. **`boolean matches(string regExp)`**

Returns `true` if the invoking string matches the regular expression passed in `regExp`. Otherwise, returns `false`.

10. **`int offsetByCodePoints(int start, int num)`**

Returns the index with the invoking string that is `num` code points beyond the starting index specified by `start`.

11. **`String replaceFirst(String regExp, String newStr)`**

Returns a string in which the first substring that matches the regular expressions specified by `regExp` is replaced by `newStr`.

12. **`String replaceAll(String regExp, String newStr)`**

Returns a string in which all substrings that match the regular expression specified by `regExp` are replaced by `newStr`.

13. **`String[] split(String regExp)`**

Decomposes the invoking string into parts and returns an array that contains the result. Each part is delimited by the regular expression passed in `regExp`.

14. **`String[] split(String regExp, int max)`**

Decomposes the invoking string into parts and returns an array that contains the result. Each part is delimited by the regular expression passed in `regExp`. The number of pieces is specified by `max`. If `max` is negative, then the invoking string is

fully decomposed. Otherwise, if max contains a nonzero value, the last entry in the returned array contains the remainder of the invoking string. If max is zero, the invoking string is fully decomposed.

15. **CharSequence subSequence(int startIndex , int stopIndex)**

Returns a substring of the invoking string, beginning at startIndex and stopping at stopIndex . This method is required by the CharSequence interface, which is now implemented by String.

Additional Methods in StringBuffer which was included in Java 5

StringBuffer appendCodePoint(int ch)

Appends a Unicode code point to the end of the invoking object. A reference to the object is returned.

int codePointAt(int i)

Returns the Unicode code point at the location specified by i.

int codePointBefore(int i)

Returns the Unicode code point at the location that precedes that specified by i.

int codePointCount(int start , int end)

Returns the number of code points in the portion of the invoking String that are between start and end – 1.

int indexOf(String str)

Searches the invoking StringBuffer for the first occurrence of str. Returns the index of the match, or –1 if no match is found.

int indexOf(String str , int startIndex)

Searches the invoking StringBuffer for the first occurrence of str, beginning at startIndex. Returns the index of the match, or –1 if no match is found.

int lastIndexOf(String str)

Searches the invoking StringBuffer for the last occurrence of str. Returns the index of the match, or –1 if no match is found.

int lastIndexOf(String str , int startIndex)

Searches the invoking StringBuffer for the last occurrence of str, beginning at startIndex. Returns the index of the match, or –1 if no match is found.

The append() method concatenates the string representation of any other type of data to the end of the invoking StringBuffer object. It has several overloaded versions. Here are a few of its forms:

StringBuffer append(String str)

StringBuffer append(int num)

StringBuffer append(Object obj)

4. The result is appended to the current StringBuffer object.
5. The buffer itself is returned by each version of append().

This allows subsequent calls to be chained together

Difference between string builder and string buffer in java

In Java, both `StringBuilder` and `StringBuffer` are classes used to create mutable sequences of characters. They are designed to provide a more efficient way to concatenate strings compared to using the `String` class, which is immutable. Here are the key differences between `StringBuilder` and `StringBuffer`:

Thread-Safety:

StringBuffer: It is thread-safe. All its methods are synchronized, which means it can be safely used by multiple threads at the same time without causing any issues.

StringBuilder: It is not thread-safe. Its methods are not synchronized, which means it should not be used by multiple threads simultaneously. However, this makes it faster in a single-threaded environment because there is no overhead of synchronization.

Performance:

StringBuffer: Due to synchronization, `StringBuffer` is generally slower than `StringBuilder` in a single-threaded environment.

StringBuilder: It is faster than `StringBuffer` because it does not have the overhead of synchronization.

Usage:

StringBuffer: Use `StringBuffer` when you need a mutable sequence of characters that will be accessed by multiple threads concurrently.

StringBuilder: Use `StringBuilder` when you need a mutable sequence of characters that will only be accessed by a single thread, or when thread-safety is not a concern.

Example:

```
StringBuffer stringBuffer = new StringBuffer();  
stringBuffer.append("Hello");  
stringBuffer.append(" World");  
System.out.println(stringBuffer.toString()); // Outputs: Hello World
```

Example:

```
StringBuilder stringBuilder = new StringBuilder();  
stringBuilder.append("Hello");  
stringBuilder.append(" World");  
System.out.println(stringBuilder.toString()); // Outputs: Hello World
```

Thread Safety: `StringBuffer` is thread-safe due to synchronization, whereas `StringBuilder` is not thread-safe.

Performance: StringBuilder is faster than StringBuffer because it lacks synchronization overhead.

Use Case: Use StringBuffer in multi-threaded contexts where thread safety is needed. Use StringBuilder in single-threaded contexts or where thread safety is not an issue and performance is critical.