Strings

Character Arrays

Character arrays in Java are used to store sequences of characters. They are declared similarly to other arrays in Java but specifically hold characters.

Declaration and initialization of character array:

```
char[] charArray = {'a', 'b', 'c', 'd'}; // Initializing a character array
```

Here, we declare a reference variable charArray of type char[] (a character array). Then, we initialize charArray by creating a new character array and assigning it the values 'a', 'b', 'c', 'd'.

Strings in Java:

Strings, which are widely used in Java programming, are a sequence of characters. In the Java programming language, strings are objects(we study about objects in detail in the OOPS lecture). The Java platform provides the String class to create and manipulate strings. The most direct and easiest way to create a string is to write: String str = "Hello world"; In the above example, "Hello world!" is a string literal—a series of characters in code that is enclosed in double quotes. Whenever it encounters a string literal in code, the compiler creates a String object with its value—in this case, Hello world!. Note: Strings in Java are immutable thus, we cannot modify their value. If we want to create a mutable string, we can use StringBuffer and StringBuilder classes.

A String can be constructed by either:

Directly assigning a string literal to a String reference - just like a primitive, or via the "new" operator and constructor, similar to any other classes(like arrays and scanner). However, this is not commonly used and is not recommended.

For example,

```
String str1 = "Java is Amazing!"; // Implicit construction via string
literal String str2 = new String("Java is Cool!"); // Explicit
construction via new
```

In the first statement, str1 is declared as a String reference and initialized with a string literal "Java is Amazing." In the second statement, str2 is declared as a String reference and initialized via the new operator to contain "Java is Cool." String literals are stored in a common pool called String pool. This facilitates the sharing of storage for strings with the same contents to conserve storage. String objects allocated via the new operator are stored in the heap memory(all non-primitives created via the new operator are stored in heap memory), and there is no sharing of storage for the same contents.

Creating strings:

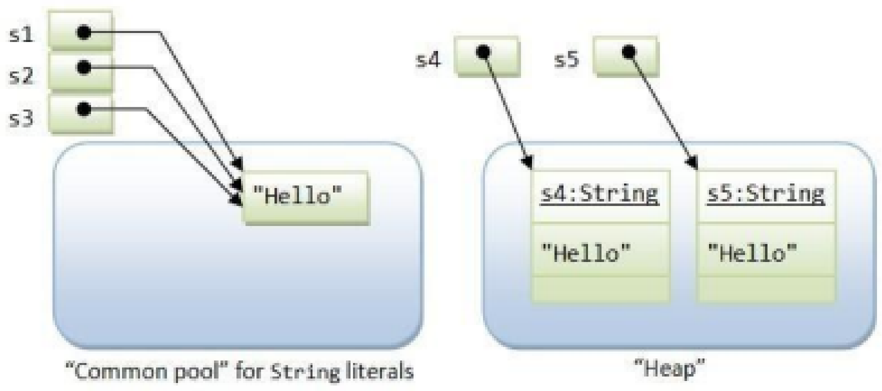
As mentioned, there are two ways to construct a string:
Implicit construction by assigning a string literal or explicitly creating a String object via the new operator and constructor.

For example,

```
String s1 = "Hello"; // String literal String s2 = "Hello"; // String

literal String s3 = s1; // same reference String s4 = new

String("Hello"); // String object String s5 = new String("Hello"); //
String object
```



Java has provided a special mechanism for keeping the String literals -- in a so-called string common pool. If two string literals have the same contents, they will share the same storage inside the common pool. This approach is adopted to conserve storage for frequently--used strings. On the other hand, String objects created via the new operator and constructor are kept in the heap memory. Each String object in the heap has its own storage just like any other object. There is no sharing of storage in the heap even if two String objects have the same contents.

Important Java Methods :

1 . String "Length" Method :

String class provides an inbuilt method to determine the length of the Java String.

For example

```
String str1 = "test string"; //Length of a String   System.out.println("Length of String: " + str.length());
```

2 . String "indexOf" Method:

String class provides an inbuilt method to get the index of a character in Java String.

For example

```
String str1 = "the string";
System.out.println("Index of character 's': "  + str_Sample.indexOf('s')); // returns 5
```

3 . String "charAt" Method:

Similar to the above question, given the index, how do I know the character at that location? Simple one again!! Use the “charAt” method and provide the index whose character you need to find.

For example :

```
String str1 = "test string";
System.out.println("char at index 3 : " + str.charAt()); // output - 't'
```

4. String "CompareTo" Method :

This method is used to compare two strings. Use the method “compareTo” and specify the String that you would like to compare. Use “compareToIgnoreCase” in case you don’t want the result to be case-sensitive. The result will have the value 0 if the argument string is equal to this string, a value less than 0 if this string is lexicographically less than the string argument and a value greater than 0 if this string is lexicographically greater than the string argument.

For example :

```
String str = "test";

System.out.println("Compare To "test": " + str.compareTo("test"));

//Compare to -- Ignore case System.out.println("Compare To "test": --

Case Ignored: " + str.compareToIgnoreCase("Test"));
```

5. String "Contain" Method :

Use the method “contains” to check if a string contains another string and specify the characters you need to check. Returns true if and only if this string contains the specified sequence of char values

For example:

```
String str = "test string";
System.out.println("Contains sequence ing: " + str.contains("ing"));
```

6. String "endsWith" Method :

This method is used to find whether a string ends with a particular prefix or not. Returns true if the character sequence represented by the argument is a suffix of the character sequence represented by this object

For example:

```
String str = "star";
System.out.println("EndsWith character 'r': " + str.endsWith("r"));
```

7. String "replaceAll" & "replaceFirst" Method:

Java String Replace, replaceAll and replaceFirst methods. You can specify the part of the String you want to replace and the replacement String in the arguments.

For example:

```
String str = "sample string"; System.out.println("Replace sample with test: " +
str.replace("sample", "test"));
```

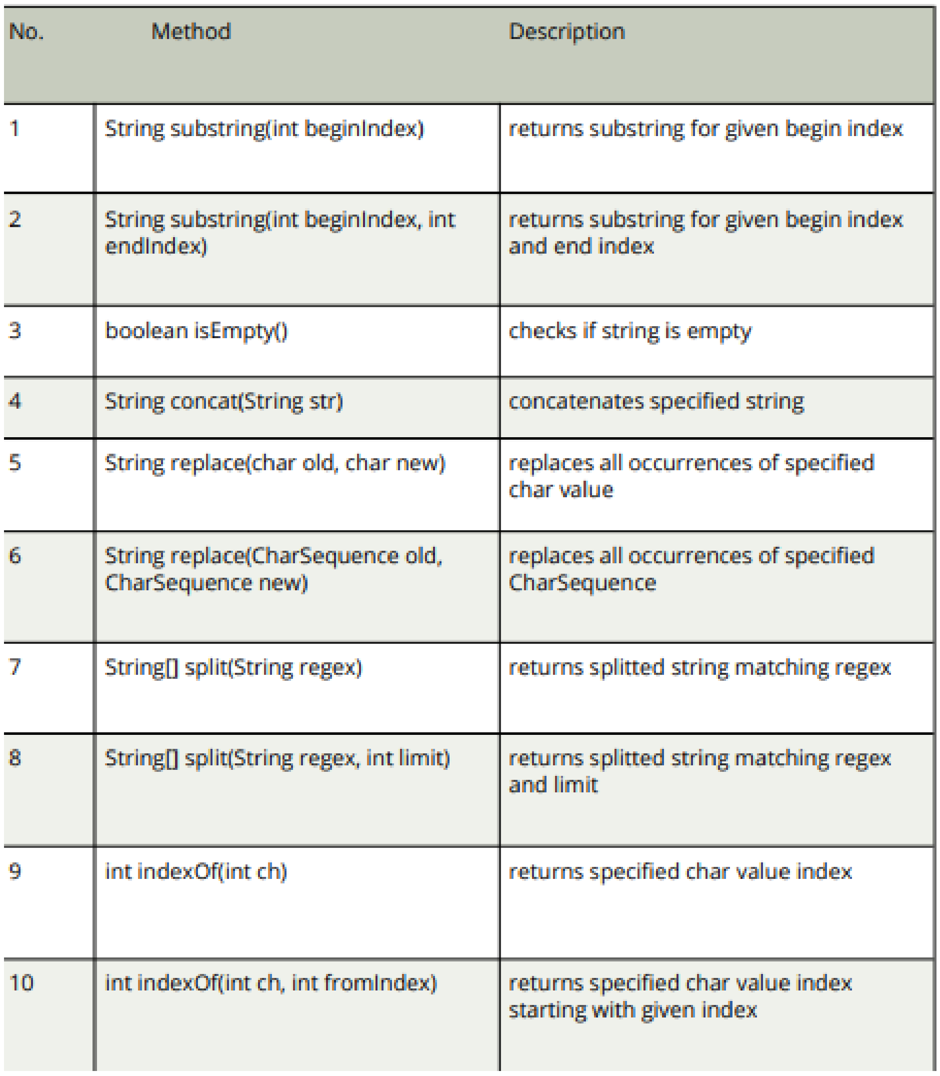
8. String Java "tolowercase" & Java "touppercase":

Use the “toLowercase()” or “ToUpperCase()” methods against the Strings that need to be converted

For example

```
String str = "TEST string";
System.out.println("Convert to LowerCase: " + str.toLowerCase());
System.out.println("Convert to UpperCase: " + str.toUpperCase());
```

Other Important Java Strings Methods:



No.	Method	Description
1	String substring(int beginIndex)	returns substring for given begin index
2	String substring(int beginIndex, int endIndex)	returns substring for given begin index and end index
3	boolean isEmpty()	checks if string is empty
4	String concat(String str)	concatenates specified string
5	String replace(char old, char new)	replaces all occurrences of specified char value
6	String replace(CharSequence old, CharSequence new)	replaces all occurrences of specified CharSequence
7	String[] split(String regex)	returns splitted string matching regex
8	String[] split(String regex, int limit)	returns splitted string matching regex and limit
9	int indexOf(int ch)	returns specified char value index
10	int indexOf(int ch, int fromIndex)	returns specified char value index starting with given index

Strings are Immutable

Since string literals with the same contents share storage in the common pool, Java's String is designed to be immutable. That is, once a String is constructed, its contents cannot be modified. Otherwise, the other String references sharing the same storage location will be affected by the change, which can be unpredictable and, therefore, undesirable. Methods such as toUpperCase() might appear to modify the contents of a String object. In fact, a completely new String object is created and returned to the caller. The original String object will be deallocated once there are no more references and subsequently garbage-collected. Because

String is immutable, it is not efficient to use String if you need to modify your string frequently (that would create many new Strings occupying new storage areas).

For Example,

// inefficient code

```
String str = "Hello";
for (int i = 1; i < 1000; ++i)
 {
 str = str + i;
 }
```

StringBuffer

As explained earlier, Strings are immutable because String literals with the same content share the same storage in the string common pool. Modifying the content of one String directly may cause adverse side effects to other Strings sharing the same storage.

StringBuffer object is just like any ordinary object, which is stored in the heap and not shared, and therefore, can be modified without causing adverse side effects to other objects. The StringBuilder class was introduced in JDK 1.5. It is the same as the StringBuffer class, except that StringBuilder is not synchronized for multi-thread operations(you can read more about multi-threading). However, for a single-thread program, StringBuilder, without the synchronization overhead, is more efficient.