String

Immutable: - Unchangeable

Mutable: - Changeable

#. The string is a sequence of characters.

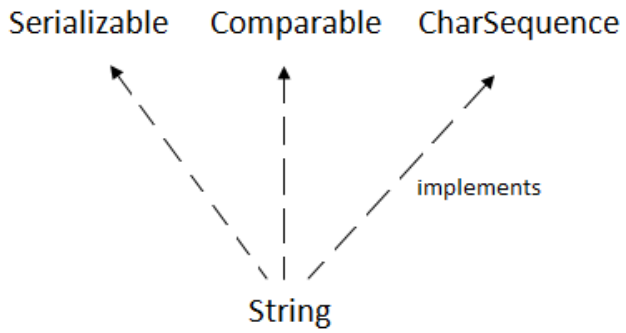
#. An array of character is a way to represent a string.

#. String is a Class in java and defined in "java.lang" package. It's not a primitive data type like int and long. String class represents character Strings. The string is a very important feature in Java because most of the time we get values as a string like when we take input from the user or fetch data from the database. Later we parse the string value and use.

#. There are 6 classes to represent a string.

1. **String** (Immutable)
2. **StringBuffer** Class (Mutable, Synchronized / Thread Safe)
3. **StringBuilder** class (Same as StringBuffer except it is not synchronized) that is inside **threading** package.
4. StringTokenizer, StringReader, and StringWriter class those are inside java.io package.

#. String, StringBuffer classes are inside **java.lang** package.



String Class

#. When to use String Class?

If we don't want to change some value in our program then we create an object of String Class and store our value like username, password etc.

#. There are usually four ways to change a string that we can't apply to the StringClass object.

1. Append
2. Delete
3. Insert
4. Replace

#. There are two ways to create String object:

1. by String Literal
2. by "new" Keyword

String Literal

#. Why we got the String Literal??

String objects are immutable so they can be shared.

"**new**" operator always creates a new memory, doesn't matter what is inside the value. With "new" operator, we are not able to re-use the immutable class. So, to share data/object it requires making an object by String Literal.

1. **String pool** is possible only because String is immutable in java, this way **Java Runtime** saves a lot of java heap space because different String variables can refer to same String variable in the pool. If String would not have been immutable, then String interning (Kaid Karna) would not have been possible because if any variable would have changed the value, it would have been reflected to other variables also.

2. If String is not immutable then it would cause severe security threat to the application. For example, database username, password are passed as String to get database connection and in **socket programming** host and port details passed as String. Since String is immutable its value can't be changed otherwise any hacker could change the referenced value to cause security issues in the application.
3. Since String is immutable, it is safe for multithreading and a single String instance can be shared across different threads. This avoid the usage of synchronization for thread safety, Strings are implicitly thread safe.

#. How to create String Literal?

Java String literal is created by using **double quotes**. All the String literals are treated as a String Class object in java. For Example:

String s="welcome";

Every **set of double quotes** (**"Ahmad"**) makes an object in String constant pool, no matter where it is written.

For example calling a function: **show("Sayeed");** now I have an object in string constant pool that value is "Sayeed".

We got value instead of reference id because String Class has overridden the **toString()** method of the object class.

#. When we create an object of a String Class using a string literal then they are kept in a special memory area called String Constant Pool.

Q Why does String literal go on String Constant Pool?

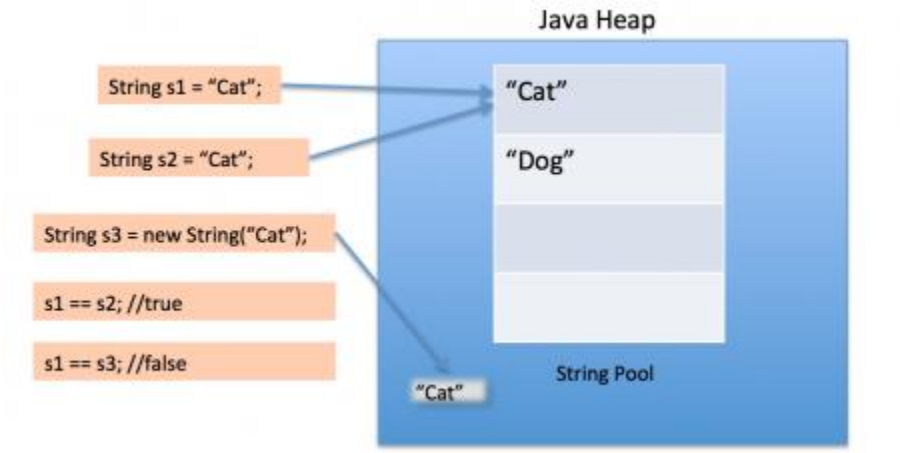
Let's suppose all objects are going in heap area.

```
String s1 = "Hello";

String s2 = new String (" Hello");

String s3 = new String ("Hello");
```

As you notice, there are three objects in heap area.

String s4 = "Hello";

Now, s4 will go and look for object value "**Hello**" but it gets three objects with the same value and this is the reason to create a new memory "String Constant Pool" **where no duplicate object creates**.



#. String Constant pool does not contain duplicate objects of a StringClass.

Why??????

To make Java more memory efficient and re-use the created data/Objects.

#. Each time we create a string literal, the JVM checks the string constant pool first. If the string already exists in the pool, a reference to the pooled instance is returned. If the string doesn't exist in the pool, a new string instance is created and placed in the pool. For example:

```
String s1="Welcome";
String s2="Welcome";  // will not create new instance
```

Garbage collection is never done on a String Class object whenever they got in a String Constant Pool.

Object Creation By new keyword

String s = new String ("Welcome");

The above example creates two objects and one reference variable.

In such case, JVM will create a new string object in normal heap memory and the literal "Welcome" will be placed in the string constant pool. The variable **s** will refer to the object in heap.

```
String str = new String("Cat");
```

In above statement, either 1 or 2 string will be created. If there is already a string literal “Cat” in the pool, then only one string “str” will be created in the pool. If there is no string literal “Cat” in the pool, then it will be first created in the pool and then in the heap space, so total 2 string objects will be created.

Q. Can we make our own class Immutable???

Yes, we can make our own Immutable class.

There are several ways that we can follow to make our own class immutable.

1. If we can’t change the value that is inside object then that will called immutable class. As we know in heap area only data member of class get memory and store value. So if we want to make our own class immutable then we have to make all data member of our class as a **blank final variable**. Always remember “blank final” not only final.

2. Make all fields of our class as blank final and private. Like String class has done. It receives our data via constructor and convert them in to byte array then give the reference id to a final variable named **value**. In JDK 8, it was “**private final char[] value;”.** Once the reference variable is declared as final, we can’t change the value and it will become immutable.

3. The other way is no need to provide "setter" methods that modify fields or objects referred to by fields.

4. Don't allow subclasses to override methods. The simplest way to do this is to declare the class as final.

5. A more sophisticated way to approach is to make the constructor private and construct instance in the factory method.

   It means, doesn't allow anyone to make a class object. Create a static method and provide them that will create an object and return a copy of reference id.
6. If the instance fields include reference to mutable object don't allow those objects to change.

7. Don't provide methods that modify the mutable objects.

8. Don't share references to the mutable objects.

9. Never store references to the external mutable object passed to the constructor; if necessary create copies and store reference to the copies.

Similarly, create a copy of your internal mutable objects when necessary to avoid returning the originals in your methods.

String Methods:

#. The String Class has a number of methods, some of that appears to modify strings. Since strings are immutable, what these methods really do is create and return a new string that contains the result of the operation.

Few of the most used methods are mentioned below.

For example: String str = "Ahmad Sayeed";

No.	Method	Description
1	char charAt(int index) char c = str.charAt(2); "m"	returns char value for the particular index
2	int length() int len = str.length();	returns string length
3	static String format(String format, Object... args)	returns formatted string
4	static String format(Locale l, String format, Object... args)	returns formatted string with given locale
5	String substring(int beginIndex)	returns substring for given begin index
6	String substring(int beginIndex, int endIndex)	returns substring for given begin index and end index
7	boolean contains(CharSequence s)	returns true or false after matching the sequence of char value
8	static String join(CharSequence delimiter, CharSequence... elements)	returns a joined string

9	static String join(CharSequence delimiter, Iterable<? extends CharSequence> elements)	returns a joined string
10	boolean equals(Object another)	checks the equality of string value with object
11	boolean isEmpty()	checks if string is empty
12	String concat(String str)	concatenates specified string
13	String replace(char old, char new)	replaces all occurrences of specified char value
14	String replace(CharSequence old, CharSequence new)	replaces all occurrences of specified CharSequence
15	static String equalsIgnoreCase(String another)	compares another string. It doesn't check case.
16	String[] split(String regex)	returns splitted string matching regex
17	String[] split(String regex, int limit)	returns splitted string matching regex and limit
18	String intern()	returns interned string
19	int indexOf(int ch)	returns specified char value index
20	int indexOf(int ch, int fromIndex)	returns specified char value index starting with given index
21	int indexOf(String substring)	returns specified substring index

22	int indexOf(String substring, int fromIndex)	returns specified substring index starting with given index
23	String toLowerCase()	Returns string in lowercase.
24	String toLowerCase(Locale l)	Returns string in lowercase using specified locale.
25	String toUpperCase()	Returns string in uppercase.
26	String toUpperCase(Locale l)	Returns string in uppercase using specified locale.
27	String trim()	Removes beginning and ending spaces of this string.
28	static String valueOf(int value)	Converts given type into string. It is overloaded.

String Constructors:

1. String(byte[] byte_arr) –

Construct a new String by decoding the *byte array*. It uses the platform's default character set for decoding.

Example:

byte[] b_arr = {71, 101, 101, 107, 115};

String s_byte =new String(b_arr);

2. String(byte[] byte_arr, Charset char_set) –

Construct a new String by decoding the *byte array*. It uses the *char_set* for decoding.

Example:

```
byte[] b_arr = {71, 101, 101, 107, 115};

Charset cs = Charset.defaultCharset();

String s_byte_char = new String(b_arr, cs);
```

3. String(byte[] byte_arr, String char_set_name) –

Construct a new String by decoding the *byte array*. It uses the *char_set_name* for decoding.
It looks similar to the above constructs and they before similar functions but it takes the *String(which contains char_set_name)* as parameter while the above constructor takes *CharSet.*

Example:

```
byte[] b_arr = {71, 101, 101, 107, 115};

String s = new String(b_arr, "US-ASCII");
```

4. 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).*

Example:

```
byte[] b_arr = {71, 101, 101, 107, 115};

String s = new String(b_arr, 1, 3);
```

5. 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.

Example:

```
byte[] b_arr = {71, 101, 101, 107, 115};
```

Charset cs = Charset.defaultCharset();

String s = new String(b_arr, 1, 3, cs);

6. String(byte[] byte_arr, int start_index, int length, String char_set_name) –

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_name* for decoding.

Example:

byte[] b_arr = {71, 101, 101, 107, 115};

String s = new String(b_arr, 1, 4, "US-ASCII");

7. String(char[] char_arr) –

Allocates a new String from the given *Character array*.

Example:

char char_arr[] = {'G', 'e', 'e', 'k', 's'};

String s = new String(char_arr);

8. **String(char[] char_array, int start_index, int count)** –

Allocates a String from a given *character array* but choose *count* characters from the *start_index*.

Example:

char char_arr[] = {'G', 'e', 'e', 'k', 's'};

String s = new String(char_arr , 1, 3);

9. **String(int[] uni_code_points, int offset, int count)** –

Allocates a String from a *uni_code_array* but choose *count* characters from the *start_index*.

Example:

int[] uni_code = {71, 101, 101, 107, 115};

String s = new String(uni_code, 1, 3);

10 **String(StringBuffer s_buffer)** –

Allocates a new string from the string in *s_buffer*

Example:

String s_buffer = "Geeks";

String s = new String(s_buffer);

11. **String(StringBuilder s_builder)** –

Allocates a new string from the string in *s_builder*

Example:

String s_builder = "Geeks";

String s = new String(s_builder);

Few Points to Remember:

#. What does String intern () method do?

When the intern method is invoked, if the pool already contains a string equal to this String object as determined by the equals (Object) method, then the string from the pool is returned. Otherwise, this String object is added to the pool and a reference to this String object is returned.
This method always return a String that has the same contents as this string, but is guaranteed to be from a pool of unique strings.

#. Name the classes that extend String class. It's a tricky question. String is a final class, so we can't extend it.

#. String is immutable and hence thread-safe in java. String is case sensitive, so "**abc**" is not equal to "**ABC**".

#. The given statements output will be "false" because in java + operator precedence is more than == operator. So the given expression will be evaluated to "s1 == s2 is:abc" == "abc" i.e false.

```
String s1 = "abc";
String s2 = "abc";
System.out.println("s1 == s2 is:" + s1 == s2);
```

#. interfaces implemented by String class are **Comparable, CharSequence, Serializable.**

#. How many String objects created in below statements? 3

```
String s = "abc"; // line 1

String s1 = new String("abcd"); // line 2
```

In line 1, "abc" created in String pool.

In line 2, first "abcd" created in string pool, then passed as argument to String new operator and another string gets created in heap memory.

So total 3 string objects gets created.

#. What will be the output of below statements?

```
String s1 = "abc";

StringBuffer s2 = new StringBuffer(s1);

System.out.println(s1.equals(s2));
```

False:

It will print false because **s2 is not of type String**. If you will look at the equals() method implementation in the String class, you will find a check using **instanceof operator** to check if the type of passed object is String? If not, then return false.

#. x.concat(y) will create a new string but it's not assigned to x, so value of x is not changed.

```
String x = "abc";
String y = "abc";
x.concat(y);
System.out.print(x);
```

#. compareTo() method of String class return 0 if value is completely same. It returns the positive value if first value is big. It return negative value if first value is small.

#. String is allowed in **switch** case for **Java 1.7 or higher versions.**

#. equals() method is used by switch-case implementation, so add null check to avoid NullPointerException.

#. Some other interesting things about String is the way we can instantiate a String object using double quotes and overloading of "+" operator for concatenation.

#. String class doesn’t provide any method to reverse the String
but **StringBuffer** and **StringBuilder** class has reverse method that we can use to check if String is palindrome or not.

#. Difference between String, StringBuffer and StringBuilder?

String is immutable and final in java, so whenever we do String manipulation, it creates a new String. String manipulations are resource consuming, so java provides two utility classes for String manipulations - StringBuffer and StringBuilder.

StringBuffer and StringBuilder are mutable classes. StringBuffer operations are thread-safe and synchronized where StringBuilder operations are not thread-safe. So when multiple threads are working on same String, we should use StringBuffer but in single threaded environment we should use StringBuilder.
StringBuilder performance is fast than StringBuffer because of no overhead of synchronization.

StringBuffer was the only choice for String manipulation till Java 1.4 but it has one disadvantage that all of its public methods are synchronized. StringBuffer provides Thread safety but on a performance cost.

In most of the scenarios, we don’t use String in multithreaded environment, so Java 1.5 introduced a new class StringBuilder that is similar with StringBuffer except thread safety and synchronization.

String concat + operator internally uses StringBuffer or StringBuilder class.

#. Why Char array is preferred over String for storing password?

String is immutable in java and stored in String pool. Once it's created it stays in the pool until unless garbage collected, so even though we are done with password it's available in memory for longer duration and there is no way to avoid it. It's a security risk because anyone having access to memory dump can find the password as clear text.
If we use char array to store password, we can set it to blank once we are done with it. So we can control for how long it's available in memory that avoids the security threat with String.

#. What is the output of below program?

```
package com.journaldev.strings;
public class Test {

        public void foo(String s) {
            System.out.println("String");
        }

        public void foo(StringBuffer sb){
              System.out.println("StringBuffer");
        }

       public static void main(String[] args) {
              new Test().foo(null);
       }
}
```

The above program will not compile: "**error as The method foo(String) is ambiguous for the type Test**".