**Java String :**

In Java, a string is a sequence of characters. For example, "hello" is a string containing a sequence of characters 'h', 'e', 'l', 'l', and 'o'.

We use double quotes to represent a string in Java. For example,

// create a string

String type = "Java programming";

Here, we have created a string variable named type. The variable is initialized with the string Java Programming.

Note: Strings in Java are not primitive types (like int, char, etc). Instead, all strings are objects of a predefined class named String.

And, all string variables are instances of the String class.

class Main {

public static void main(String[] args) {

// create strings

String first = "Java";

String second = "Python";

String third = "JavaScript";

// print strings

System.out.println(first); // print Java

System.out.println(second); // print Python

System.out.println(third); // print JavaScript

}

}

In the above example, we have created three strings named first, second, and third. Here, we are directly creating strings like primitive types.

However, there is another way of creating Java strings (using the new keyword).

Java String provides various methods to perform different operations on strings.

|  |  |
| --- | --- |
| **Methods** | **Description** |
| substring() | returns the substring of the string |
| replace() | replaces the specified old character with the specified new character |
| charAt() | returns the character present in the specified location |
| getBytes() | converts the string to an array of bytes |
| indexOf() | returns the position of the specified character in the string |
| compareTo() | compares two strings in the dictionary order |
| trim() | removes any leading and trailing whitespaces |
| format() | returns a formatted string |
| split() | breaks the string into an array of strings |
| toLowerCase() | converts the string to lowercase |
| toUpperCase() | converts the string to uppercase |
| valueOf() | returns the string representation of the specified argument |
| toCharArray() | converts the string to a char array |

**Escape character in Java Strings :**

The escape character is used to escape some of the characters present inside a string.

Suppose we need to include double quotes inside a string.

// include double quote

String example = "This is the "String" class";

Since strings are represented by double quotes, the compiler will treat "This is the " as the string. Hence, the above code will cause an error.

To solve this issue, we use the escape character \ in Java. For example,

// use the escape character

String example = "This is the \"String\" class.";

Now escape characters tell the compiler to escape double quotes and read the whole text.

**Java Strings are Immutable :** In Java, strings are immutable. This means, once we create a string, we cannot change that string. Immutable class means that once an object is created, we cannot change its content. In Java, all the wrapper classes (like Integer, Boolean, Byte, Short) and String class is immutable.

To understand it more deeply, consider an example:

// create a string

String example = "Hello! ";

Here, we have created a string variable named example. The variable holds the string "Hello! ".

Now suppose we want to change the string.

// add another string "World" to the previous tring example

example = example.concat(" World");

Here, we are using the concat() method to add another string World to the previous string.

It looks like we are able to change the value of the previous string. However, this is not true.

Let's see what has happened here,

JVM takes the first string "Hello! "

creates a new string by adding "World" to the first string

assign the new string "Hello! World" to the example variable

the first string "Hello! " remains unchanged

**Creating strings using the new keyword :**So far we have created strings like primitive types in Java. Since strings in Java are objects, we can create strings using the new keyword as well. For example,

// create a string using the new keyword

String name = new String("Java String");

In the above example, we have created a string name using the new keyword.

Here, when we create a string object, the String() constructor is invoked.

**Create String using literals vs new keyword :** Now that we know how strings are created using string literals and the new keyword, let's see what is the major difference between them.

In Java, the JVM maintains a string pool to store all of its strings inside the memory. The string pool helps in reusing the strings.

While creating strings using string literals, the value of the string is directly provided. Hence, the compiler first checks the string pool to see if the string already exists.

If the string already exists, the new string is not created. Instead, the new reference points to the existing string.

If the string doesn't exist, the new string is created.

However, while creating strings using the new keyword, the value of the string is not directly provided. Hence the new string is created all the time.

**Memory allotment of String :** Whenever a String Object is created as a literal, the object will be created in String constant pool. This allows JVM to optimize the initialization of String literal.

For example: String str = "Geeks";

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

For example: String str = new String("Geeks");

If you want to store this string in the constant pool then you will need to “**intern**” it.

For example: String internedString = **str.intern()**;

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

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

**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:** String is a sequence of characters. In java, objects of String are immutable which means a constant and cannot be changed once created.

**Creating a String :** There are two ways to create a string in Java:

**String literal :** String s = “GeeksforGeeks”;

**Using new keyword :** String s = new String (“GeeksforGeeks”);

**StringBuffer:** StringBuffer is a peer class of String that provides much of the functionality of strings. The string represents fixed-length, immutable character sequences while StringBuffer represents growable and writable character sequences.

Syntax:

StringBuffer s = new StringBuffer("GeeksforGeeks");

**StringBuilder:** The StringBuilder in Java represents a mutable sequence of characters. Since the String Class in Java creates an immutable sequence of characters, the StringBuilder class provides an alternate to String Class, as it creates a mutable sequence of characters.

Syntax:

StringBuilder str = new StringBuilder();

str.append("GFG");

**How to make class Immutable? : What is an immutable class in Java?**

Immutable objects are instances whose state doesn’t change after it has been initialized. For example, String is an immutable class and once instantiated its value never changes.

An immutable class is good for caching purposes because you don’t have to worry about the value changes.

Another **benefit** of immutable class is that it is inherently thread-safe, so you don’t have to worry about thread safety in case of multi-threaded environment.

**To create an immutable class in Java, you have to do the following steps.**

* Declare the class as **final** so it can’t be extended.
* Make all fields **private** so that direct access is not allowed.
* Make all mutable fields **final** so that its value can be assigned only once.
* Don’t provide setter methods for variables.
* Initialize all the fields via a **constructor** performing deep copy.
* Perform cloning of objects in the getter methods to return a copy rather than returning the actual object reference.

**Differene between String ,StringBuffer and StringBuilder :**

* If a string is going to remain constant throughout the program, then use the String class object because a String object is immutable.
* If a string can change (for example: lots of logic and operations in the construction of the string) and will only be accessed from a single thread, using a StringBuilder is good enough.
* If a string can change and will be accessed from multiple threads, use a StringBuffer because StringBuffer is synchronous, so you have thread-safety.
* If you don’t want thread-safety than you can also go with StringBuilder class as it is not synchronized.

**Conversion between types of strings in Java :**

**From String to StringBuffer and StringBuilder:** This one is easy. We can directly pass the String class object to StringBuffer and StringBuilder class constructors. As the String class is immutable in java, so for editing a string, we can perform the same by converting it to StringBuffer or StringBuilder class objects.

String str = "Geeks";

// conversion from String object to StringBuffer

StringBuffer sbr = new StringBuffer(str);

sbr.reverse();

System.out.println(sbr);

// conversion from String object to StringBuilder

StringBuilder sbl = new StringBuilder(str);

sbl.append("ForGeeks");

System.out.println(sbl);

**From StringBuffer and StringBuilder to String:**

This conversion can be performed using toString() method which is overridden in both StringBuffer and StringBuilder classes.

Note that while we use toString() method, a new String object(in Heap area) is allocated and initialized to the character sequence currently represented by the StringBuffer object, which means the subsequent changes to the StringBuffer object do not affect the contents of the String object.

StringBuffer sbr = **new** StringBuffer("Geeks");

        StringBuilder sbdr = **new** StringBuilder("Hello");

        // conversion from StringBuffer object to String

        String str = sbr.toString();

        System.out.println(

            "StringBuffer object to String : ");

        System.out.println(str);

        // conversion from StringBuilder object to String

        String str1 = sbdr.toString();

        System.out.println(

            "StringBuilder object to String : ");

        System.out.println(str1);

        // changing StringBuffer object sbr

        // but String object(str) doesn't change

        sbr.append("ForGeeks");

**From StringBuffer to StringBuilder or vice-versa:**  This conversion is tricky. There is no direct way to convert the same. In this case, We can use a String class object. We first convert the StringBuffer/StringBuilder object to String using toString() method and then from String to StringBuilder/StringBuffer using constructors.

StringBuffer sbr = **new** StringBuffer("Geeks");

        // conversion from StringBuffer

        // object to StringBuilder

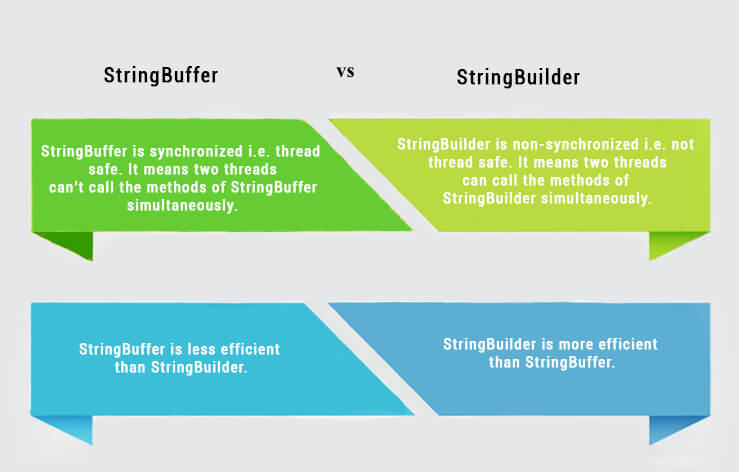
        String str = sbr.toString();

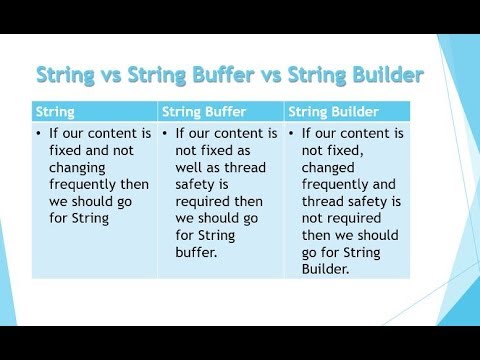
        StringBuilder sbl = **new** StringBuilder(str);

**Conclusion:**

Objects of String are immutable, and objects of StringBuffer and StringBuilder are mutable.

StringBuffer and StringBuilder are similar, but StringBuilder is faster and preferred over StringBuffer for the single-threaded program. If thread safety is needed, then StringBuffer is used.





**Java: String is Immutable. What exactly is the meaning?**

**String str = "knowledge";**

This, as usual, creates a string containing "knowledge" and assigns it a reference str.

// assigns a new reference to the

// same string "knowledge"

**String s = str;**

Let’s see how the below statement works:

**str = str.concat(" base");**

This appends a string " base" to str. But wait, how is this possible, since String objects are immutable? Well to your surprise, it is.

When the above statement is executed, the VM takes the value of String str, i.e. "knowledge" and appends " base", giving us the value "knowledge base". Now, since Strings are immutable, the VM can’t assign this value to str, so it creates a new String object, gives it a value "knowledge base", and gives it a reference str.

An important point to note here is that, while the String object is immutable, its reference variable is not. So that’s why, in the above example, the reference was made to refer to a newly formed String object.

At this point in the example above, we have two String objects: the first one we created with value "knowledge", pointed to by s, and the second one "knowledge base", pointed to by str. But, technically, we have three String objects, the third one being the literal "base" in the concat statement.

# Important Facts about String and Memory usage

What if we didn’t have another reference s to "knowledge"? We would have lost that String. However, it still would have existed, but would be considered lost due to having no references.

|  |
| --- |
| **import** java.io.\*;    **class** GFG {  **public** **static** **void** main(String[] args)      {          String s1 = "java";          s1.concat(" rules");            // Yes, s1 still refers to "java"          System.out.println("s1 refers to " + s1);      }  } |

**Output**:

s1 refers to java

**What’s happening:**

1. The first line is pretty straightforward: create a new String "java" and refer s1 to it.
2. Next, the VM creates another new String "java rules", but nothing refers to it. So, the second String is instantly lost. We can’t reach it.

The reference variable s1 still refers to the original String "java".

Almost every method, applied to a String object in order to modify it, creates new String object. So, where do these String objects go? Well, these exist in memory, and one of the key goals of any programming language is to make efficient use of memory.

As applications grow, it’s very common for String literals to occupy large area of memory, which can even cause redundancy. So, in order to make Java more efficient, the JVM sets aside a special area of memory called the “[String constant pool](https://www.geeksforgeeks.org/how-to-initialize-and-compare-strings-in-java/)“.

When the compiler sees a String literal, it looks for the String in the pool. If a match is found, the reference to the new literal is directed to the existing String and no new String object is created. The existing String simply has one more reference. Here comes the point of making String objects immutable:

In the String constant pool, a String object is likely to have one or many references. If several references point to same String without even knowing it, it would be bad if one of the references modified that String value. That’s why String objects are immutable.

**How to Initialize and Compare Strings in Java?**

**Initializing Strings in Java**

1. **Direct Initialization(String Constant) :** In this method, a String constant object will be created in String pooled area which is inside heap area in memory. As it is a constant, we can’t modify it, i.e. String class is immutable.  
   Examples:

String str = "GeeksForGeeks";

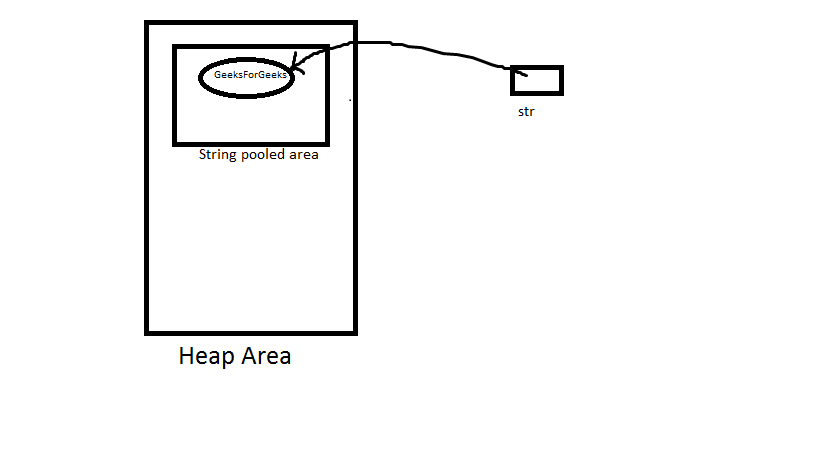
str = "geeks"; // This statement will make str

// point to new String constant("geeks")

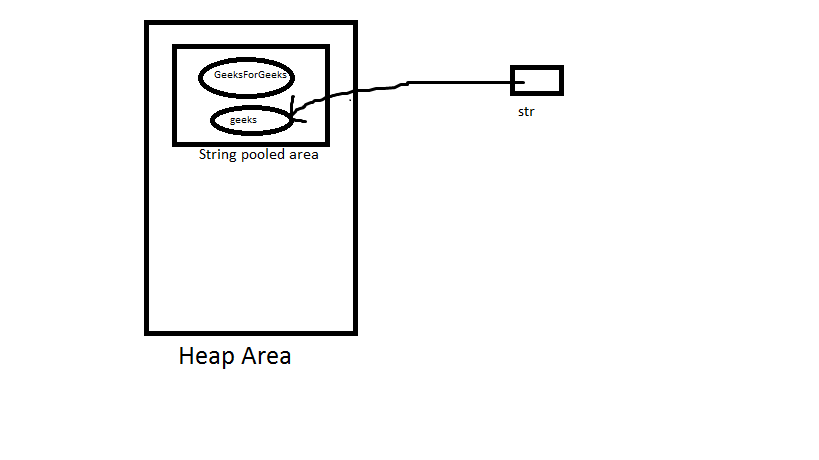
// rather than modifying the previous

// String constant.

**String str = “GeeksForGeeks”;**



**str = “geeks”;**



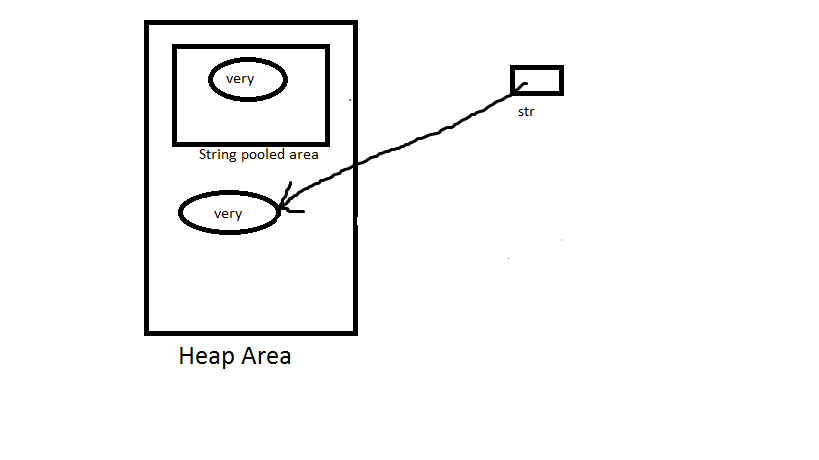
**Note**: If we again write str = “GeeksForGeeks” as next line, then it first check that if given String constant is present in String pooled area or not. If it present then str will point to it, otherwise creates a new String constant.

**2. Object Initialization (Dynamic):** In this method, a String object will be created in heap area (not inside String pooled area as in upper case). We can’t modify it(like in upper case). Also with same value, a String constant is also created in String pooled area, but the variable will point to String object in heap area only.  
Examples:

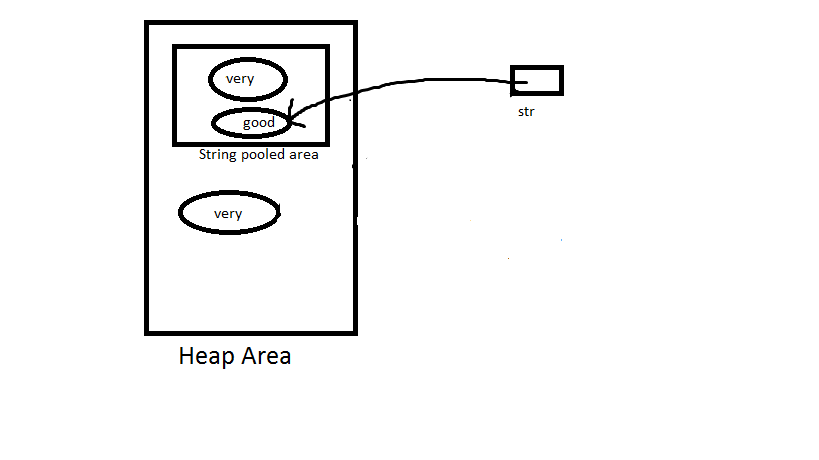
String str = new String("very");

str = "good";

**String str = new String(“very”)**



**str = “good”**  
Now this is a direct assignment, so String constant with value “good” is created in String pooled area and str will point to that.



**Note**: If we again write str = new String(“very”), then it will create a new object with value “very”, rather than pointing to the available objects in heap area with same value.But if we write str = “very”, then it will point to String constant object with value “very”, present in String pooled area.

**Comparing Strings and their References**

1. **equals()** method: It compares values of string for equality. Return type is boolean. In almost all the situation you can use useObjects.equals().

**2. == operator:** It compares references not values. Return type is boolean. == is used in rare situations where you know you’re dealing with interned strings.  
**3. compareTo()** method: It compares values lexicographically and returns an integer value that describes if first string is less than, equal to or greater than second string.For example, if str1 and str2 are two string variables then if   
str1 == str2 : return 0   
str1 > str2 : return a positive value   
str1 < str2 : return a negative value

**Note**: The positive and negative values returned by compareTo method is the difference of first unmatched character in the two strings.

**Compare two Strings in Java**

String is a sequence of characters. In Java, objects of String are immutable which means they are constant and cannot be changed once created.

**Below are 5 ways to compare two Strings in Java:**

**Using user-defined function :** Define a function to compare values with following conditions :

if (string1 > string2) it returns a positive value.

if both the strings are equal lexicographically

i.e.(string1 == string2) it returns 0.

if (string1 < string2) it returns a negative value.

The value is calculated as (int)str1.charAt(i) – (int)str2.charAt(i)

**Using String.equals() :**In Java, string equals() method compares the two given strings based on the data/content of the string. If all the contents of both the strings are same then it returns true. If any character does not match, then it returns false.

**Syntax:**

**str1.equals(str2);**

Here str1 and str2 both are the strings which are to be compared.

**Using String.equalsIgnoreCase() :** The String.equalsIgnoreCase() method compares two strings irrespective of the case (lower or upper) of the string. This method returns true if the argument is not null and the contents of both the Strings are same ignoring case, else false.

**Syntax:**

**str2.equalsIgnoreCase(str1);**

Here str1 and str2 both are the strings which are to be compared.

**Using Objects.equals() :** Object.equals(Object a, Object b) method returns true if the arguments are equal to each other and false otherwise. Consequently, if both arguments are null, true is returned and if exactly one argument is null, false is returned. Otherwise, equality is determined by using the equals() method of the first argument.

**Syntax:**

**public static boolean equals(Object a, Object b)**

Here a and b both are the string objects which are to be compared.

**Using String.compareTo() :**

Syntax:

int str1.compareTo(String str2)

Working:

It compares and returns the following values as follows:

if (string1 > string2) it returns a positive value.

if both the strings are equal lexicographically

i.e.(string1 == string2) it returns 0.

if (string1 < string2) it returns a negative value.

**Why not to use == for comparison of Strings?**

In general both equals() and “==” operator in Java are used to compare objects to check equality but here are some of the differences between the two:

Main difference between .equals() method and == operator is that one is method and other is operator.

One can use == operators for reference comparison (address comparison) and .equals() method for content comparison.

In simple words, == checks if both objects point to the same memory location whereas .equals() evaluates to the comparison of values in the objects.

Here two String objects are being created namely s1 and s2.

Both s1 and s2 refers to different objects.

When one uses == operator for s1 and s2 comparison then the result is false as both have different addresses in memory.

Using equals, the result is true because its only comparing the values given in s1 and s2.

public class Test {

public static void main(String[] args) {

String s1 = new String("HELLO");

String s2 = new String("HELLO");

System.out.println(s1 == s2); // FALSE

System.out.println(s1.equals(s2)); // TRUE

}

}