**Strings can be created by literal,intern and new object**

1 String s2 = new String("xyz");

2 String s3 ="xyz";

3 String s4 = new String("xyz");

4 String s5 ="xyz";

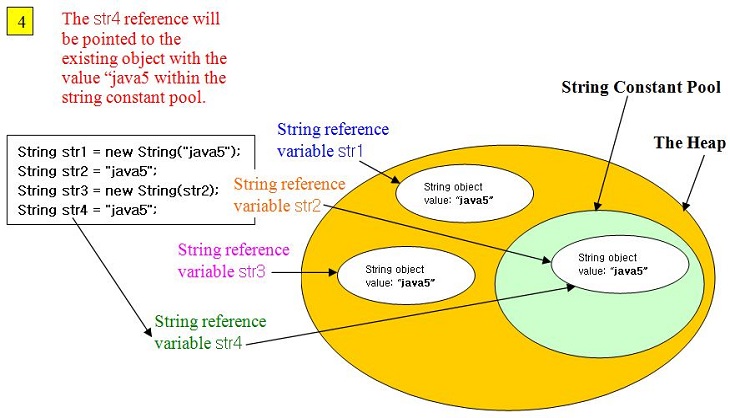
If we execute the above line we will have three object.

* The first and as mentioned will create **two objects** one in heap and another in String constant pool. Using **string literal** "xyz" in the constructor, a new string value is stored in string constant pool. Using **new operator**, a new string object is created in the heap with "xyz" as value
* When the second line executes it checks in the string constant pool  
  and find with "xyz" so it returns the same object, so till second line we have two objects.
* when the third line executes it will create a new object in the heap since new operator creates object in the heap so till third line will have 3 objects.
* When the fourth line executes it checks in the string constant pool  
  and find with "xyz" so it returns the same object, so fourth line we have three objects.

**intern()**

* The intern() method creates an exact copy of a string that is present in the heap memory and stores it in the String constant pool if not already present. If the string is already present, it returns the reference.
* A string literal always invokes the intern() method, whether one mention the intern() method along with the string literal or not.
* String s1 = "Javatpoint";
* String s2 = s1.intern();
* String s3 = **new** String("Javatpoint");
* String s4 = s3.intern();
* System.out.println(s1==s2); // True
* System.out.println(s1==s3); // False
* System.out.println(s1==s4); // True
* System.out.println(s2==s3); // False
* System.out.println(s2==s4); // True
* System.out.println(s3==s4); // False

**Example 2**



String str1 = new String("java5");

Using string literal "java5" in the constructor, a new string value is stored in string constant pool. Using new operator, a new string object is created in the heap with "java5" as value.

String str2 = "java5"

Reference "str2" is pointed to already stored value in string constant pool

String str3 = new String(str2);

A new string object is created in the heap with the same value as reference by "str2"

String str4 = "java5";

Reference "str4" is pointed to already stored value in string constant pool

Total objects : Heap - 2, Pool – 1

String str1 = **new** String("abc");

String str2 = **new** String("abc");

Three objects are created. For the first statement(str1) two objects are created one in String constant pool and one in heap memory.  
But for the second statement(str2), compulsory 1 new object is created in heap memory but no new object is created in string constant pool as it is already present.  
  
Hence , a total of 2+1 = 3 objects are created.

**String Vs StringBuilder Vs StringBuffer :**

**String:**

String is **immutable**, means **instance of an object can not be modified without creating the new object**. whenever we do String manipulation like concatenation, substring, etc. it generates a new String and discards the older String for garbage collection. These are heavy operations and generate a lot of garbage in heap. String belongs to java.lang package.

Public final class String extends Object implements Serializable,Comparable<String>,CharSequence

String a ="javadomain";  
String b = "javadomain"; // object already present in the string pool, so it will make a reference instead of creating the new object.

String stringVar = "hello";  
stringVar.concat(" world"); //String is immutable so must be assigned to new object, since it wont allow us to change the existing object reference without creating new object. So here it creates new object which increases

StringBuilder stringBuilderVar = new StringBuilder("hello");  
stringBuilderVar.append(" world"); //StringBuilder is mutable, so can be modified without creating new object.

**StringBuffer:**

* StringBuffer is mutable, means **instance of an object can be modified without creating new object**.
* StringBuffer is **synchronized.**
* StringBuffer is**thread safe**.
* synchronized methods are present in StringBuffer making control of one thread access at a time while it is not seen in StringBuilder, hence thread-unsafe
* StringBuffer is **slow** than StringBuilder.

**StringBuilder:**

* StringBuilder is also mutable, means in StringBuilder also **instance of an object can be modified without creating new object.**
* StringBuilder is **not synchronized**
* StringBuilder is **not thread safe.**
* StringBuilder is **faster** than StringBuffer, since it is not synchronized and not thread safe.

**StringBuilder is faster than StringBuffer,** you can understand this term by the below program,

**StringTokenizer** is used to break string into tokens. the delimiters can be provided at the time of creation or one by one to the tokens.

**To split string**

String s="Good morning everyone";

String output=s.split(" ");

StringUtils.split(s);//best way compared to above bcz here null-safe method

and by default, assumes a whitespace delimiter

**To join strings**

String s[]={"Hi","world"};

method 1

Arrays.toString(s);

output => "[Hi, world]"

method 2

StringBuilder sb=new StringBuilder();

for(String s1:s)

{

sb.append(s);

}

sb.toString();

output => "Hi world"

method 3

String.join(" ",s);

method 4

Arrays.stream(s).collect(Collectors.joining());

---------------------------------------------------------------

conversions of different datatypes

String to respective datatypes

int i=Integer.parseInt(string); //returns primitive(by using parseInt method from Integer class)

Boolean.parseBoolean(string),Long.parseLong(string),Double.parseDouble(),Float.parseFloat()

Integer i=Integer.valueOf(string);//return Object

Long.valueOf(string),.............

Int to String

String s=Integer.toString(12);

String s4=String.valueOf(int);

String s=new Integer(12).toString();

**char**[] ch={'j','a','v','a','t','p','o','i','n','t'};

String s=**new** String(ch);

char[] c= s.toCharArray();

replace(char oldChar, char newChar)

replaceFirst(regex,string),replaceAll(regex,string) these 2 methods uses regex inside methods

**For StringBuilder,StringBuffer will not work equals() but String will work as shown below**

String str="madam";

StringBuilder str1=**new** StringBuilder(str);

StringBuilder str2=**new** StringBuilder(str);

str2.reverse();

if(str1.equals(str2))//will not work bcz by String Builder you will be getting object

**if**(str1.toString().equals(str2.toString())) {

System.***out***.println("Plaindrome");}

**else**

System.***out***.println("not plaindrome");

**Important String methods**

|  |  |  |
| --- | --- | --- |
| charAt() | indexOf() | compareTo(),  compareToIgnoreCase() |
| contains() | trim() | format() |
| startsWith(),endsWith() | replace,replaceFirst(),replaceAll() | codePointAt() |

Important String Builder methods mainly manipulating methods are used here

String buffer also uses these methods with synchronized

|  |  |  |
| --- | --- | --- |
| append(String s) | capacity() default is 16  if you add no.of chars then  capacity+1 | public String substring(int beginIndex)  public String substring(int beginIndex, int endindex) |
| public StringBuilder insert(int offset,String s) | public StringBuilder replace(int startIndex, int endIndex, String str) | public StringBuilder delete(int startIndex, int endIndex) |

String Tokenizer

String mydelim = " : ";

String mystr = "JAVA : Code : String : Tokenizer : Geeks";

// Here we are passing Delimiter - "mydelim"

StringTokenizer geeks3 =

new StringTokenizer(mystr, mydelim);

int count = geeks3.countTokens();

System.out.println("Number of tokens : " + count + "\n");

for (int i = 0; i < count; i++)

System.out.println("token at [" + i + "] : "

+ geeks3.nextToken());

// checks for more tokens using hasMoreTokens() method

// which holds true till there is single element remaining

while (geeks3.hasMoreTokens())

// Returning the next token

// using nextToken() method

System.out.println(geeks3.nextToken());

Conversions

String str = "Geeks";

// Converting String object to StringBuffer object

        // creating object of StringBuffer class

        StringBuffer sbr = **new** StringBuffer(str)

        // Converting String object to StringBuilder object

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

String s=sbr.toString(); //StringBuffer to String

String s=sb1.toString();//StringBuilder to String

**Similar to all conversions octal,binary also**

Hexadecimal numbers 0-9 -> 0-9 from 10-> a to 15->f

Integer.toHexString(10)) //Converts int to hexadecimal number

**Integer.toString(number, 16)** //16 is base & you can convert any base

289 convert to hexadecimal

LCM by 16

16|289

---

16| 18-1

-----

1-2

289%16 -> 1

289/16 = 18

18%16 -> 2

18/16 =1

1%16->1

1/16=0

Answer=121

1. **public** **static** String toHex(**int** decimal){
2. **int** rem;
3. String hex="";
4. **char** hexchars[]={'0','1','2','3','4','5','6','7','8','9','A','B','C','D','E','F'};
5. **while**(decimal>0)
6. {
7. rem=decimal%16;
8. hex=hexchars[rem]+hex;
9. decimal=decimal/16;
10. }
11. **return** hex;
12. }

String s="amazon";

String op="",op1="";

//clockwise rotation 2 times

op=op+s.substring(s.length()-2,s.length())+s.substring(0, s.length()-2);

//anticlockwise rotation 2 times

op1= op1 + s.substring(2) + s.substring(0, 2) ;

--------------------------------------------------------

**int** ar[] = {10, 5, 15, 20, 12, 90, 100};

Arrays.*sort*(ar,Collections.*reverseOrder*()); //works only on non primitive datatypes as collection is used

Arrays.*sort*(ar,1,5); //sorts from 1 to 5 elements

//works only on non primitive datatypes as collection is used

Arrays.*sort*(ar,1,5,Collections.*reverseOrder*()); //sorts from 1 to 5 elements in reverse order

//To convert from int[] to Integer[]

int[] data = {1,2,3,4,5,6,7,8,9,10};

// To boxed array

Integer[] what = Arrays.stream( data ).boxed().toArray( Integer[]::new );

**String different from C and Java**

In C, string is just array of chars and terminated with Null character. But in Java String is a object of class java.lang

**Why String is mostly used as key in map?**

When you create a HashMap object and try to store key-value pair in it. While store hashcode of the given key is calculated and its value is placed at the position represented by resultant hashcode of the key. Since String is immutable, it is used as key in map.

**Why char Array is preferred over String in storing passwords?**  
  
One of the main reason to prefer char Array over String is security risk of stealing passwords. Since String are reusable in the constant pool , there are high chances that they remain in the memory for the long duration. Anyone who has access to the memory dump can find the password in clear text.  
That's why password should be encrypted.

**concatenation**

**String str1 = "Hello";**

**String str2 = "World";**

**String result = str1.concat(str2);**

Best Use:   When concatenating string objects.  
Worst Use: When using on dynamic objects, if str1 becomes null before concat() method it will throw NullPointerException.

String result=str+”world”;

Best Use:  When concatenating the constant strings.  
Worst Use:  When using + operator inside loop.

**Why String is immutable in java ?**

**Answer : To sava space, Security, Caching, Synchronization**

* As we cannot modify String in constant pool. A lot of heap space is saved by [JRE](https://www.javatpoint.com/java-jre)
* For database usernames, passwords,hosts,ports are passed as strings to receive database connections. If the String doesn't remain immutable, any hacker can cause a security issue in the application by changing the reference value.
* The String objects are cached in the String pool. The cached String literals are accessed by multiple clients. So, there is always a risk, where action performs by one client affects all other clients. For example, if one client performs an action and changes the string value from Pressure to PRESSURE, all remaining clients will also read that value. For the performance reason, caching of String objects was important, so to remove that risk, we have to make the String Immutable.
* String is safe for multithreading

**Regex**

? 0 or 1

+ 1 or more

\*0 or more

Need to use \ for Special characters ? . \* + ^ { } ( ) [] $

/kittens/.{1,3}/ use curly braces for ranges 1 to 3 . should be there

{3} exactly 3

{3,} 3 or more

Ex: Kittens.

[0-9A-Z]+ i.e 0 to 9 or A-Z one or more

[A-Z][^0-9][/.?!] i.e first letter should be capital in second letter there should not be number and last letter should be one of those character

^[A-Z][^\.?!]+[\.?!]$ i.e ^ starting letter $ ending letter and in middle no punctuation should be there

\s whitespace and \S not whitespace

\d digit \D not digit

\w[0-9A-Z a-z \_] word \W not word

\b word boundary means before the \b is \w and after is \W or viceversa

\B not word boundary

Example /bstem/b “rose has lovely stem!” (correct)

System!(not correct)

\m\g\i\s

Multiline not just a string

Global(matches all occurencies not only 1)

I case insensitive , s single line(matches new line \n also)

Greedy vs lazy quantifiers

By default its greedy

Gre\*(0 or more) -> Greeee (greedy as many as) ->matches all

Gre\*?->lazy(as few as)-> In Greeting it only matches Gr bcz 0 or more. Here ? acts for lazy

Multicharacter quantifiers

I love (kittens|dogs|birds) | means or

24 hour clock example

(1?\d|2[0-3]):[0-5]\d

Html tags verify

\1 is same as first group

<(\w+)>.\*?<[\\\1](file:///\\\1)>

**Positive Look behind** (?<=) means which contain immediate before match

Ex: match words after meta\_\_

a\_tag, meta\_\_another\_tag, meta\_\_third\_tag, fourth\_tag

solution -> /(?<=meta\_\_)\w+/g

**Negative Look behind** (?<!) means doesnot contain immediate before match

**Positive Look ahead** (?=) means which contain immediate after match

Ex: match words before\_tag

a\_tag, meta\_\_another\_tag, meta\_\_third\_tag, fourth\_tag

solution -> /\w+(?=\_tag)/g

**Negative Look ahead** (?!) means doesnot contain immediate after match

Match all words in sentence except last word

Solution-> "\w+\b(?!\W?$)"g