STRING IN JAVA

Generally, **String** is a sequence of characters. But in Java, **String** is an object that represents a sequence of characters. The **java.lang.String** class is used to create a string object.

How to create a string object?

There are two ways to create String object:

1. By string literal
2. By **new** keyword

1) String Literal

Java String literal is created by using double quotes.

String s="Bishal Patel";

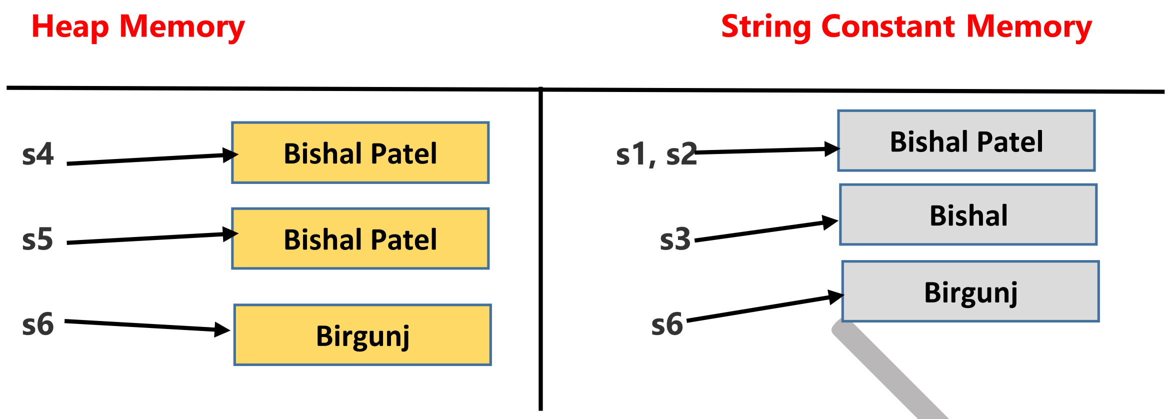
2) By new keyword

String s=**new String**("Bishal Patel");

Note: ***Each time when a String is created in java, sometimes the object of string is stored in the heap memory and sometimes the object is stored in the String constant pool.***

1. **When a String is created by String Literal, String object is created and stored in the String constant memory.**
2. **When a String is created by new keyword, Two String object is created and stored in the heap memory & String constant memory.**

String s1="Bishal Patel";
String s2="Bishal Patel"; //doesn't create new object
String s3="Bishal";
String s4=new **String**("Bishal Patel");
String s5=new **String**("Bishal Patel");
String s6=new **String**("Birgunj");



An array of characters works same as Java string:

```
char[] ch={'B','i','s','h','a','l'};
String s=new String(ch);
```

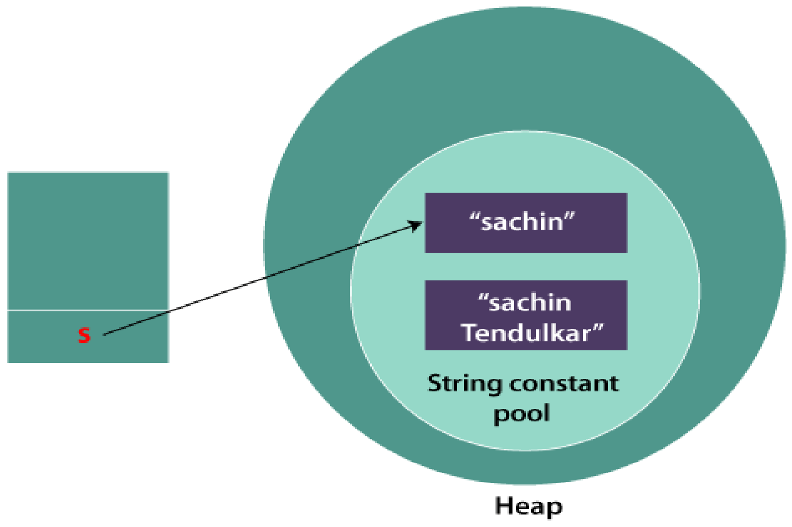
```
String s="Bishal";
```

Immutable String in Java

Immutable means **non-changeable** in nature. Once **String object** is created, its data or state can't be changed. but if you want to change the state of string forcefully, new object is created with that changeable object.

```
class Testimmutablestring{
 public static void main(String args[]){
   String s="Sachin";
   s.concat("Tendulkar");//concat() method appends the string at the end
   System.out.println(s);//will print Sachin because strings are immutable objects
 }
}
```

Now it can be understood by the diagram given below. Here Sachin is not changed but a new object is created with Sachin Tendulkar. That is why String is known as immutable.



As you can see in the above figure that two objects are created but s reference variable still refers to "Sachin" not to "Sachin Tendulkar".

But if we explicitly assign it to the reference variable, it will refer to "Sachin Tendulkar" object.

```
class Testimmutablestring1{
 public static void main(String args[]){
   String s="Sachin";
   s=s.concat(" Tendulkar");
   System.out.println(s);
 }
}
```

Why String objects are immutable in Java?

As Java uses the concept of String literal. Suppose there are 5 reference variables, all refer to one object "Sachin". If one reference variable changes the value of the object, it will be affected by all the reference variables. That is why String objects are **immutable** in Java.

String Length() function

The Java String **length()** method finds the length of a string.

String s="Bishal";

int a=s.length();

String Constructor in Java

Java String Constructors allows us to create a string object from different type of arguments.

1. **String**()
2. **String**("String_literals")
3. **String**(**byte[]** b)
4. **String**(**char[]** ch)
5. **String**(**char[]** ch, **int** offset, **int** count)
6. **String**(**StringBuffer** s)
7. **String**(**StringBuilde**r s)

String()

String s=**new String**();

It will create an empty string object and the reference variable s refers to an empty string object. The length of the empty string is **0.**

System.out.println(s.**length()**);   // prints 0

String("String_literals")

String s=**new String**(**"Bishal"**);

It will create a string object "Bishal" and the reference variable s referes the object Bishal. The string object is created by string literals.

System.out.println(s.**length()**);   // prints 5

String(byte[] b)

If we want to create a string from a byte array then we go for String(byte[] b) constructor. It is used to create a string from byte array.

Eg: -

byte[] **b**={119, 101, 108, 99, 111, 109, 101};

String s=**new String**(**b**);

System.out.println(s);   // prints welcome

System.out.println(s.**length()**);   // prints 7

String(char[] b)

If we want to create a string from a char array then we go for String(char[] ch) constructor. It is used to create a string from char array.

Eg: -

```
char[] ch={'B','i','s','h','a','l'};

String s=new String(ch);

System.out.println(s);   // prints Bishal

System.out.println(s.length());   // prints 6
```

String(char[] b, int offset, int count)

If we want to create a string from another string as a substring then we go for String(**char[]** ch, **int** offset, **int** count) constructor. **Offset** means index and **count** means number of character. It is used to create a string from another string as a substring.

Eg: -

```
char[] ch={'B','i','s','h','a','l,'p','a','t','e','l'};

String s=new String(ch, 6, 5);

System.out.println(s);   // prints patel

System.out.println(s.length());   // prints 5
```

String(StringBuffer sb)

If we want to create a string from a StringBuffer then we go for String(StrinBuffer sb) constructor. It is used to create a string from StringBuffer.

StringBuffer is a Mutable String which means it can be changed.

Eg: -

```
StringBuffer sb=new StringBuffer("Bishal");

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

System.out.println(s); // prints Bishal

System.out.println(s.**length()**); // prints 6

String(StringBuilder sb)

If we want to create a string from a StringBuffer then we go for String(StrinBuffer sb) constructor. It is used to create a string from StringBuffer.

StringBuffer is a Mutable String which means it can be changed.

Eg: -

StringBuilder **sb**=**new** StringBuilder("Bishal");

String s=**new** **String**(**sb**);

System.out.println(s); // prints Bishal

System.out.println(s.**length()**); // prints 6

Some of String Methods

charAt()

The Java String **charAt()** method returns a char value at the given index number.

The index number starts from 0 and goes to n-1, where n is the length of the string. It returns **StringIndexOutOfBoundsException**, if the given index number is greater than or equal to this string length or a negative number.

Syntax

```
public char charAt(int index)
```

```
public class CharAtExample{
   public static void main(String args[]){
        String name="Bishal Patel";
        char ch=name.charAt(4);  //returns the char value at the 4th index
         System.out.println(ch);
    }
}
```

getChars()

The Java String **getChars**() method copies the content of the string into a specified **char** array. There are four arguments passed in the **getChars**() method.

This method throws **StringIndexOutOfBoundsException** if we give wrong index, so compulsory we have to handle it by try catch block.

The signature of the **getChars**() method is given below:

```
public void getChars(int src, int srcEndIndex, char[] destination, int dstBeginIndex)
```

Parameters

int src: The index from where copying of characters is started.

int srcEndIndex: The index which is next to the last character that is getting copied.

Char[] destination: The char array where characters from the string that invokes the **getChars**() method is getting copied.

int dstEndIndex: It shows the position in the destination array from where the characters from the string will be pushed.

```
public class StringGetCharsExample{
  public static void main(String args[]){
     String str = new String("hello javatpoint how r u");
     char[] ch = new char[10];
     try{
       str.getChars(6, 16, ch, 0);
       System.out.println(ch);
     }catch(Exception ex){System.out.println(ex);}
  }
}
```

getBytes()

The Java String **getBytes**() method does the encoding of string into the sequence of bytes and keeps it in an array of bytes.

Syntax

```
public byte[] getBytes()
public class StringGetBytesExample{
  public static void main(String args[]){
     String s1="ABCDEFG";
     byte[] b=s1.getBytes();
     for(int i=0;i<b.length;i++){
        System.out.println(b[i]);
     }
  }
}
```

toCharArray()

The java string **toCharArray**() method converts the string into character array. its length is similar to this string and its contents are initialized with the characters of this string.

Syntax

`public char[] toCharArray()`

```
public class StringToCharArrayExample{
   public static void main(String args[]){
     String s1="hello";
     char[] ch=s1.toCharArray();
     for(int i=0;i<ch.length;i++){
        System.out.print(ch[i]);
     }
   }
}
```

Some String Comparison Methods

equals()

The Java String class **equals()** method compares the two given strings based on the **content** of the string. If any character is not matched, it returns **false**. If all characters are matched, it returns **true**.

The String **equals()** method overrides the **equals()** method of the **Object class**.

Syntax

public boolean equals(Object anotherObject)

```
public class EqualsExample{
   public static void main(String args[]){
     String s1="bishal";
     String s2="bishal";
     String s3="BISHAL";
     String s4="kunal";
     System.out.println(s1.equals(s2));//true because content and case is same
     System.out.println(s1.equals(s3));//false because case is not same
     System.out.println(s1.equals(s4));//false because content is not same
   }
}
```

equalsIgnoreCase()

The Java String class **equalsIgnoreCase**() method compares the two given strings on the basis of the **content** of the string irrespective of the case (**lower and upper**) of the string. It is just like the **equals**() method but doesn't check the case sensitivity.

If any character is not matched, it returns false, else returns true.

Syntax

```
publicboolean equalsIgnoreCase(String str)
```

```
public class EqualsExample{
   public static void main(String args[]){
     String s1="bishal";
     String s2="bishal";
     String s3="BISHAL";
     String s4="kunal";
     System.out.println(s1.equalsIgnoreCase(s2));
     System.out.println(s1.equalsIgnoreCase(s3));
     System.out.println(s1.equalsIgnoreCase(s4));
   }
}
```

startsWith()

The Java String class **startsWith**() method checks if this string starts with the given prefix. It returns true if this string starts with the given prefix; else returns false.

Syntax

```
public boolean startsWith(String prefix)
public boolean startsWith(String prefix, int offset)
```

```
public class StartsWithExample
{
    public static void main(String args[])
    {
         String s1="java string split method by Bishal";
         System.out.println(s1.startsWith("ja"));  // true
         System.out.println(s1.startsWith("java string"));   // true
         System.out.println(s1.startsWith("Java string"));  // false as 'j' and 'J' are different

         System.out.println(str.startsWith("a",1)); // True
         System.out.println(str.startsWith("s",5)); // True
         System.out.println(str.startsWith("a",0)); // false
    }
}
```

endswith()

The Java String class **endsWith**() method checks if this string ends with a given suffix. It returns **true** if this string ends with the given suffix; else returns **false**.

Syntax

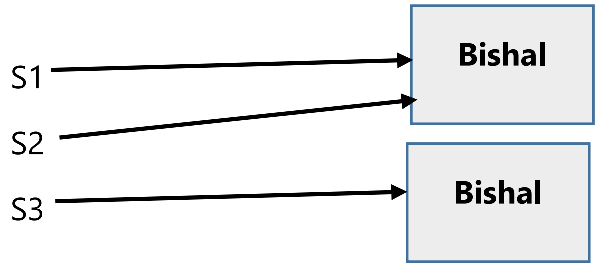
```
public boolean endsWith(String suffix)
```

```
public class EndsWithExample{
    public static void main(String args[]){
        String s1="java by Bishal";
        System.out.println(s1.endsWith("l"));
        System.out.println(s1.endsWith("Bishal"));
 }}
```

equals() vs ==

equals() method meant for content comparision. If the content of the reference is same then it will return **true** orelse it returns **false**.

== is meant for reference comparision. If multiple reference variable refers to the same object then only **==** return true orelse it returns false.



```
public class EqualsAndDoubleEquals
{
   public static void main(String[] args)
   {
      String s1=new String("bishal");
      String s2=new String("bishal");

      System.out.println(s1.equals(s2));  //true
      System.out.println(s1==s2);     //false
   }
}
```

Another Example

```
public class EqualsAndDoubleEquals
{
   public static void main(String[] args)
   {
      String s1="Bishal";
      String s2="Bishal";

      System.out.println(s1.equals(s2));  //true
      System.out.println(s1==s2);      //true
   }
}
```

regionMatches()

The **regionMatches()** method of the String class has two **variants** that can be used to test if two string regions are matching or equal. There are two variants of this method, i.e., one is case sensitive test method, and the other ignores the case-sensitive method.

Syntax:

1. Case sensitive test method:

public Boolean regionMatches(**int** toffset, **String** other, **int** ooffset, **int** len)

2. It has the option to consider or ignore the case method:

public boolean regionMatches(**boolean** **ignoreCase**, **int** toffset, **String** other, **int** ooffset, **int** len)

Parameters:

- **ignoreCase**: if **true**, ignore the case when comparing characters.
- **toffset**: the starting offset of the subregion in this string.
- **other**: the string argument being compared.
- **ooffset**: the starting offset of the subregion in the string argument.
- **len**: the number of characters to compare.

```
public class Test {
  public static void main(String args[]) {
    String Str1 = new String("Welcome to BIRGUNJ");
    String Str2 = new String("Birgunj");
    String Str3 = new String("BIRGUNJ");

    System.out.println(Str1.regionMatches(11, Str2, 0, 7));   //false
    System.out.println(Str1.regionMatches(11, Str3, 0, 7));   //true
    System.out.println(Str1.regionMatches(true,11, Str2, 0, 7));    //true
  }
}
```

String modification Methods

1. Substring()
2. concat()
3. replace()
4. trim()

substring()

The Java String class **substring()** method returns a part of the string.

We pass beginIndex and endIndex number position in the Java **substring()** method where **beginIndex** is **inclusive**, and **endIndex** is **exclusive**. In other words, the **beginIndex** starts from **0**, whereas the **endIndex** starts from **1**.

There are two types of substring methods in Java string.

```
public String substring(int startIndex);
public String substring(int startIndex, int endIndex);
```

```
public class SubstringExample{
    public static void main(String args[]){
        String s1="Kathmandu";
        System.out.println(s1.substring(2,4));  // th
        System.out.println(s1.substring(2));  // thmandu
    }
}
```

```
public class SubstringExample2 {
    public static void main(String[] args) {
        String s1="Kathmandu";
        String substr = s1.substring(0); // Starts with 0 and goes to end
        System.out.println(substr);
        String substr2 = s1.substring(4,9); // Starts from 4 and goes to 9
        System.out.println(substr2);
        String substr3 = s1.substring(4,15); // Returns Exception
    }
}
```

concat()

The Java String class **concat()** method combines specified string at the end of this string. It returns a combined string. It is like appending another string.

```
public String concat(String anotherString)
```

```
public class ConcatExample2 {
   public static void main(String[] args) {
      String str1 = "Hello";
      String str2 = "Birgunj";
      String str3 = "Chandalchowk";
      // Concatenating one string
      String str4 = str1.concat(str2);
      System.out.println(str4);
      // Concatenating multiple strings
      String str5 = str1.concat(str2).concat(str3);
      System.out.println(str5);
   }
}
```

replace()

The Java String class **replace()** method returns a string replacing all the **old char** or **CharSequence** to **new char** or **CharSequence**.

```
public String replace(char oldChar, char newChar)
public String replace(CharSequence target, CharSequence replacement)
```

```
public class ReplaceExample1{
     public static void main(String args[]){
        String s1="Birgunj is a very good place";
        String s2="my name is Programming my name is java";

        String replaceString=s1.replace('a', 'e');  //replaces all occurrences of 'a' to 'e'
        System.out.println(replaceString);
        String replaceString1=s2.replace("is", "was");
        System.out.println(replaceString1);

    }
}
```

trim()

The **trim()** method in Java String is a built-in function that eliminates **leading** and **trailing** spaces. The Unicode value of space character is **'\u0020'**. The **trim()** method in java checks this Unicode value before and after the string, if it exists then removes the spaces and returns the omitted string.

Note: ***The trim() method doesn't eliminate middle spaces***.

Syntax

public String trim()

```
public class StringTrimExample{

   public static void main(String args[]){

      String s1="  hello Bishal     ";

      System.out.println(s1+"i am from birgunj");  //without trim()

      System.out.println(s1.trim()+"i am from birgunj");  //with trim()

}}
```