Java String

In [Java](https://www.javatpoint.com/java-tutorial), string is basically an object that represents sequence of char values. An [array](https://www.javatpoint.com/array-in-java) of characters works same as Java string. For example:

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

String s=**new** String(ch);

is same as:

String s="javatpoint";

**Java String** class provides a lot of methods to perform operations on strings such as compare(), concat(), equals(), split(), length(), replace(), compareTo(), intern(), substring() etc.

The java.lang.String class implements *Serializable*, *Comparable* and *CharSequence* [interfaces](https://www.javatpoint.com/interface-in-java).



CharSequence Interface

The CharSequence interface is used to represent the sequence of characters. String, [StringBuffer](https://www.javatpoint.com/StringBuffer-class) and [StringBuilder](https://www.javatpoint.com/StringBuilder-class) classes implement it. It means, we can create strings in Java by using these three classes.



The Java String is immutable which means it cannot be changed. Whenever we change any string, a new instance is created. For mutable strings, you can use StringBuffer and StringBuilder classes.

We will discuss immutable string later. Let's first understand what String in Java is and how to create the String object.

What is 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. For Example:

1. String s="welcome";

Each time you 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:

1. String s1="Welcome";
2. String s2="Welcome";//It doesn't create a new instance



In the above example, only one object will be created. Firstly, JVM will not find any string object with the value "Welcome" in string constant pool that is why it will create a new object. After that it will find the string with the value "Welcome" in the pool, it will not create a new object but will return the reference to the same instance.

Note: String objects are stored in a special memory area known as the "string constant pool".

Why Java uses the concept of String literal?

To make Java more memory efficient (because no new objects are created if it exists already in the string constant pool).

2) By new keyword

1. String s=**new** String("Welcome");//creates two objects and one reference variable

In such case, [JVM](https://www.javatpoint.com/jvm-java-virtual-machine) will create a new string object in normal (non-pool) heap memory, and the literal "Welcome" will be placed in the string constant pool. The variable s will refer to the object in a heap (non-pool).

Java String Example

**StringExample.java**

**public** **class** StringExample{

**public** **static** **void** main(String args[]){

String s1="java";//creating string by Java string literal

**char** ch[]={'s','t','r','i','n','g','s'};

String s2=**new** String(ch);//converting char array to string

String s3=**new** String("example");//creating Java string by new keyword

System.out.println(s1);

System.out.println(s2);

System.out.println(s3);

}}

**Output:**

java

strings

example

The above code, converts a ***char*** array into a **String** object. And displays the String objects ***s1, s2***, and ***s3*** on console using ***println()*** method.

String methods

charAt(int index) method

The charAt(int index) method of Java **StringBuffer** returns the char value of the current sequence at the specified index. The first character starts at index 0, second at index 1, and so on. The index argument of the character sequence must be equal to or greater than 0 and less than the length of the current sequence.

Syntax:

**public** **char** charAt(**int** index)

Parameter:

|  |  |  |
| --- | --- | --- |
| **DataType** | **Parameter** | **Description** |
| Int | index | It is an index of specific character value. |

Returns:

The charAt(int index) method returns the character at a given index of string buffer.

Example 1

**public** **class** StringBufferCharAtExample1 {

**public** **static** **void** main(String[] args) {

        StringBuffer sb = **new** StringBuffer("stringbuffer");

        System.out.println("string: "+sb);

        // printing character at index value 2

        System.out.println("character at index 2: "+sb.charAt(2));

    }  }

**Output:**

string: stringbuffer

character at index 2: r

Example 2

When providing the index value as negative in charAt(int) throw an exception.

**public** **class** StringBufferCharAtExample2 {

**public** **static** **void** main(String[] args) {

        StringBuffer sb = **new** StringBuffer("stringbuffer");

        System.out.println("string: "+sb);

        // try to print character at index value negative

        System.out.println("character at index -1: "+sb.charAt(-1));

    }

}

**Output:**

Exception in thread "main" java.lang.StringIndexOutOfBoundsException: String index out of range: -1

at java.lang.StringBuffer.charAt(Unknown Source)

at StringBufferCharAtExample2.main(StringBufferCharAtExample2.java:6)

string: stringbuffer

Example 3

When providing the index value as greater than the length of string throws an exception.

**public** **class** StringBufferCharAtExample3 {

**public** **static** **void** main(String[] args) {

        StringBuffer sb = **new** StringBuffer("stringbuffer");

        System.out.println("string: "+sb);

        // try to print character at index value at size length

        System.out.println("character at index equal to length: "+sb.charAt(sb.length()));

    }

}

**Output:**

Exception in thread "main" java.lang.StringIndexOutOfBoundsException: String index out of range: 12

at java.lang.StringBuffer.charAt(Unknown Source)

at StringBufferCharAtExample3.main(StringBufferCharAtExample3.java:6)

string: stringbuffer

Example 4

In this example, we are providing string and index input from user.

**import** java.util.Scanner;

**public** **class** StringBufferCharAtExample4 {

**public** **static** **void** main(String[] args) {

       StringBuffer sb = **new** StringBuffer("");          System.out.print("enter your string value: ");

       Scanner sc = **new** Scanner(System.in);

       sb.append(sc.nextLine());

       System.out.print("enter index value: ");

**int** index = sc.nextInt();

        // printing the character at input index

        System.out.println("character at index "+index+ ": "+sb.charAt(index));

        sc.close();

    }

}

**Output:**

enter your string value: hello

enter index value: 0

character at index 0: h

String Concatenation in Java

In Java, String concatenation forms a new String that is the combination of multiple strings. There are two ways to concatenate strings in Java:

1. By + (String concatenation) operator
2. By concat() method
3. By + (String concatenation) operator

Java String concatenation operator (+) is used to add strings. For Example:

**TestStringConcatenation1.java**

**class** TestStringConcatenation1{

**public** **static** **void** main(String args[]){

   String s="Sachin"+" Tendulkar";

   System.out.println(s);//Sachin Tendulkar

 }

}

**Output:**

Sachin Tendulkar

The **Java compiler transforms** above code to this:

1. String s=(**new** StringBuilder()).append("Sachin").append(" Tendulkar).toString();

In Java, String concatenation is implemented through the StringBuilder (or StringBuffer) class and it's append method. String concatenation operator produces a new String by appending the second operand onto the end of the first operand. The String concatenation operator can concatenate not only String but primitive values also. For Example:

**TestStringConcatenation2.java**

**class** TestStringConcatenation2{

**public** **static** **void** main(String args[]){

   String s=50+30+"Sachin"+40+40;

   System.out.println(s);//80Sachin4040

 }

}

**Output:**

80Sachin4040

Note: After a string literal, all the + will be treated as string concatenation operator.

2) String Concatenation by concat() method

The String concat() method concatenates the specified string to the end of current string. Syntax:

**public** String concat(String another)

**TestStringConcatenation3.java**

**class** TestStringConcatenation3{

**public** **static** **void** main(String args[]){

   String s1="Sachin ";

   String s2="Tendulkar";

   String s3=s1.concat(s2);

   System.out.println(s3);//Sachin Tendulkar

  }

}

**Output:**

Sachin Tendulkar

The above Java program, concatenates two String objects ***s1*** and ***s2*** using ***concat()*** method and stores the result into ***s3*** object.

There are some other possible ways to concatenate Strings in Java,

1. StringBuilder class

StringBuilder is class provides append() method to perform concatenation operation. The append() method accepts arguments of different types like Objects, StringBuilder, int, char, CharSequence, boolean, float, double. StringBuilder is the most popular and fastet way to concatenate strings in Java. It is mutable class which means values stored in StringBuilder objects can be updated or changed.

**StrBuilder.java**

**public** **class** StrBuilder

{

    /\* Driver Code \*/

**public** **static** **void** main(String args[])

    {

        StringBuilder s1 = **new** StringBuilder("Hello");    //String 1

        StringBuilder s2 = **new** StringBuilder(" World");    //String 2

        StringBuilder s = s1.append(s2);   //String 3 to store the result

            System.out.println(s.toString());  //Displays result

    }

}

**Output:**

Hello World

In the above code snippet, **s1, s2** and **s** are declared as objects of **StringBuilder** class. **s** stores the result of concatenation of **s1** and **s2** using **append**() method.

2. String concatenation using format() method

String.format() method allows to concatenate multiple strings using format specifier like %s followed by the string values or objects.

**StrFormat.java**

**public** **class** StrFormat

{

    /\* Driver Code \*/

**public** **static** **void** main(String args[])

    {

        String s1 = **new** String("Hello");    //String 1

        String s2 = **new** String(" World");    //String 2

        String s = String.format("%s%s",s1,s2);   //String 3 to store the result

            System.out.println(s.toString());  //Displays result

    }

}

**Output:**

Hello World

Here, the String objects **s** is assigned the concatenated result of Strings **s1** and **s2** using **String.format()** method. format() accepts parameters as format specifier followed by String objects or values.

3. String.join() method (Java Version 8+)

The String.join() method is available in Java version 8 and all the above versions. String.join() method accepts arguments first a separator and an array of String objects.

**StrJoin.java:**

**public** **class** StrJoin

{

    /\* Driver Code \*/

**public** **static** **void** main(String args[])

    {

        String s1 = **new** String("Hello");    //String 1

        String s2 = **new** String(" World");    //String 2

        String s = String.join("",s1,s2);   //String 3 to store the result

            System.out.println(s.toString());  //Displays result

    }

}

**Output:**

Hello World

In the above code snippet, the String object **s** stores the result of **String.join("",s1,s2)** method. A separator is specified inside quotation marks followed by the String objects or array of String objects.

4. StringJoiner class (Java Version 8+)

StringJoiner class has all the functionalities of String.join() method. In advance its constructor can also accept optional arguments, prefix and suffix.

**StrJoiner.java**

**public** **class** StrJoiner

{

    /\* Driver Code \*/

**public** **static** **void** main(String args[])

    {

        StringJoiner s = **new** StringJoiner(", ");   //StringeJoiner object

        s.add("Hello");    //String 1

        s.add("World");    //String 2

        System.out.println(s.toString());  //Displays result

    }

}

**Output:**

Hello, World

In the above code snippet, the StringJoiner object **s** is declared and the constructor StringJoiner() accepts a separator value. A separator is specified inside quotation marks. The add() method appends Strings passed as arguments.

5. Collectors.joining() method (Java (Java Version 8+)

The Collectors class in Java 8 offers joining() method that concatenates the input elements in a similar order as they occur.

**ColJoining.java**

**import** java.util.\*;

**import** java.util.stream.Collectors;

**public** **class** ColJoining

{

    /\* Driver Code \*/

**public** **static** **void** main(String args[])

    {

    List<String> liststr = Arrays.asList("abc", "pqr", "xyz"); //List of String array

    String str = liststr.stream().collect(Collectors.joining(", ")); //performs joining operation

    System.out.println(str.toString());  //Displays result

    }

}

**Output:**

abc, pqr, xyz

Here, a list of String array is declared. And a String object **str** stores the result of **Collectors.joining()** method.

Java String 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.

Signature

publicboolean equals(Object anotherObject)

Parameter

**anotherObject** : another object, i.e., compared with this string.

Returns

**true** if characters of both strings are equal otherwise **false**.

Internal implementation

**public** **boolean** equals(Object anObject) {

**if** (**this** == anObject) {

**return** **true**;

      }

**if** (anObject **instanceof** String) {

          String anotherString = (String) anObject;

**int** n = value.length;

**if** (n == anotherString.value.length) {

**char** v1[] = value;

**char** v2[] = anotherString.value;

**int** i = 0;

**while** (n-- != 0) {

**if** (v1[i] != v2[i])

**return** **false**;

                  i++;

              }

**return** **true**;

          }

      }

**return** **false**;

  }

equals() Method Example

**FileName:** EqualsExample.java

**public** **class** EqualsExample{

**public** **static** **void** main(String args[]){

String s1="javatpoint";

String s2="javatpoint";

String s3="JAVATPOINT";

String s4="python";

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

}}  **[Test it Now](https://www.javatpoint.com/opr/test.jsp?filename=EqualsExample" \t "_blank)**

**Output:**

true

false

false

equals() Method Example 2

The equals() method compares two strings and can be used in if-else control structure.

**FileName:** EqualsExample2.java

**public** **class** EqualsExample2 {

**public** **static** **void** main(String[] args) {

        String s1 = "javatpoint";

        String s2 = "javatpoint";

        String s3 = "Javatpoint";

        System.out.println(s1.equals(s2)); // True because content is same

**if** (s1.equals(s3)) {

            System.out.println("both strings are equal");

        }**else** System.out.println("both strings are unequal");

    }

}

**Output:**

true

both strings are unequal

Java String equals() Method Example 3

**FileName:** EqualsExample3.java

**import** java.util.ArrayList;

**public** **class** EqualsExample3 {

**public** **static** **void** main(String[] args) {

        String str1 = "Mukesh";

        ArrayList<String> list = **new** ArrayList<>();

        list.add("Ravi");

        list.add("Mukesh");

        list.add("Ramesh");

        list.add("Ajay");

**for** (String str : list) {

**if** (str.equals(str1)) {

                System.out.println("Mukesh is present");

            }

        }

    }

}

**Output:**

Mukesh is present

AD

equals() Method Example 4

The internal implementation of the equals() method shows that one can pass the reference of any object in the parameter of the method. The following example shows the same.

**FileName:** EqualsExample4.java

**public** **class** EqualsExample4

{

// main method

**public** **static** **void** main(String argvs[])

{

// Strings

String str = "a";

String str1 = "123";

String str2 = "45.89";

String str3 = "false";

Character c = **new** Character('a');

Integer i = **new** Integer(123);

Float f = **new** Float(45.89);

Boolean b = **new** Boolean(**false**);

// reference of the Character object is passed

System.out.println(str.equals(c));

// reference of the Integer object is passed

System.out.println(str1.equals(i));

// reference of the Float object is passed

System.out.println(str2.equals(f));

// reference of the Boolean object is passed

System.out.println(str3.equals(b));

// the above print statements show a false value because

// we are comparing a String with different data types

// To achieve the true value, we have to convert

// the different data types into the string using the toString() method

System.out.println(str.equals(c.toString()));

System.out.println(str1.equals(i.toString()));

System.out.println(str2.equals(f.toString()));

System.out.println(str3.equals(b.toString()));

}

}

**Output:**

false

false

false

false

true

true

true

true

indexOf() Method

The indexOf() method of List interface returns the index of the first occurrence of the specified element in this list. It returns -1 if the specified element is not present in this list.

Syntax

**public** **int** indexOf(Object o)

Parameters

The parameter 'o' represents the element to be searched.

Throws:

**ClassCastException**- If the type of the specified element is not compatible with this list.

**NullPointerException**- If the specified element is null and this list does not allow null elements.

Return

The indexOf() method returns the index of the first occurrence of the specified element if it is present in this list, else it returns -1.

Example 1

**import** java.util.LinkedList;

**import** java.util.List;

**public** **class** JavaListIndexOfExample1 {

**public** **static** **void** main(String[] args) {

        List<Integer> list= **new** LinkedList<>();

**for** (**int** i=0;i<6;i++){

list.add(i);

// returns the element at the specified position in this list

**int** value =list.indexOf(i);

System.out.println("Element stored at index "+i+" : "+value);

        }

    }

}

**Output:**

Element stored at index 0 : 0

Element stored at index 1 : 1

Element stored at index 2 : 2

Element stored at index 3 : 3

Element stored at index 4 : 4

Element stored at index 5 : 5

Example 2

**import** java.util.LinkedList;

**import** java.util.List;

**public** **class** JavaListIndexOfExample2 {

**public** **static** **void** main(String[] args) {

        List<Integer> list= **new** LinkedList<>();

list.add(**null**);

list.add(**null**);

list.add(**null**);

// returns -1 if the no value is present in the specified index

**int** value =list.indexOf(90);

System.out.println("Element stored at Index "+90+" : "+value);

    }

}

**Output:**

Element stored at Index 90 : -1

Example 3

**import** java.util.LinkedList;

**import** java.util.List;

**public** **class** JavaListIndexOfExample3 {

**public** **static** **void** main(String[] args) {

        List<Integer> list= **new** LinkedList<>();

list.add(67);

list.add(89);

// returns -1 if the no value is present in the specified index

**int** value =list.indexOf(**null**);

System.out.println("Element stored at "+**null**+" : "+value);

    }

}

**Output:**

Element stored at null : -1

isEmpty() Method

The **isEmpty()** method of List interface returns a Boolean value 'true' if this list contains no elements.

Syntax

**public** **boolean** isEmpty()

Specified By

isEmpty in interface Collection<E>

Return

The isEmpty () method returns the Boolean value 'true' if this list contains no elements, else it returns false.

Example 1

**import** java.util.LinkedList;

**import** java.util.List;

**public** **class** JavaListIsEmptyExample1 {

**public** **static** **void** main(String[] args) {

        List<Character> list = **new** LinkedList<Character>();

//this methods checks whether the invoked list is empty or not

Boolean bool = list.isEmpty();

**if**(bool){

System.out.println("Enter elements in this list as it is empty.");

        }

**else**{

System.out.println("Elements are already present in this list.\nList : "+list);

        }

    }

}

**Output:**

Enter elements as this list is empty.

Example 2

**import** java.util.LinkedList;

**import** java.util.List;

**public** **class** JavaListIsEmptyExample2 {

**public** **static** **void** main(String[] args) {

        List<Character> list = **new** LinkedList<Character>();

list.add(**null**);

//this methods checks whether the invoked list is empty or not

Boolean bool = list.isEmpty();

//even if there are null elements it will return false

**if**(bool){

System.out.println("Enter elements in this list as it is empty.");

        }

**else**{

System.out.println("Elements are already present in this list.\nList : "+list);

        }

    }

}

**Output:**

Elements are already present in this list.

List : [null]

join()

The **Java String class join()** method returns a string joined with a given delimiter. In the String join() method, the delimiter is copied for each element. The join() method is included in the Java string since JDK 1.8.

There are two types of join() methods in the Java String class.

Signature

The signature or syntax of the join() method is given below:

**public** **static** String join(CharSequence delimiter, CharSequence... elements)

and

**public** **static** String join(CharSequence delimiter, Iterable<? **extends** CharSequence> elements)

Parameters

**delimiter** : char value to be added with each element

**elements** : char value to be attached with delimiter

Returns

joined string with delimiter

Exception Throws

**NullPointerException** if element or delimiter is null.

Internal Implementation

// type - 1

**public** **static** String join(CharSequence delimiter, CharSequence... elements)

{

        Objects.requireNonNull(elements);

        Objects.requireNonNull(delimiter);

        StringJoiner jnr = **new** StringJoiner(delimiter);

**for** (CharSequence c: elements)

        {

            jnr.add(c);

        }

**return** jnr.toString();

}

// type - 2

**public** **static** String join(CharSequence delimiter, CharSequence... elements)

{

        Objects.requireNonNull(elements);

        Objects.requireNonNull(delimiter);

        StringJoiner jnr = **new** StringJoiner(delimiter);

**for** (CharSequence c: elements)

        {

            jnr.add(c);

        }

**return** jnr.toString();

}

**public** **static** String join(CharSequence delimiter, Iterable<? **extends** CharSequence> elements)

{

        Objects.requireNonNull(elements);

        Objects.requireNonNull(delimiter);

        StringJoiner jnr = **new** StringJoiner(delimiter);

**for** (CharSequence c: elements)

        {

            joiner.add(c);

        }

**return** jnr.toString();

    }

Java String join() Method Example

**FileName:** StringJoinExample.java

**public** **class** StringJoinExample{

**public** **static** **void** main(String args[]){

String joinString1=String.join("-","welcome","to","javatpoint");

System.out.println(joinString1);

}}

**Output:**

welcome-to-javatpoint

Java String join() Method Example 2

**FileName:** StringJoinExample2.java

**public** **class** StringJoinExample2 {

**public** **static** **void** main(String[] args) {

        String date = String.join("/","25","06","2018");

        System.out.print(date);

        String time = String.join(":", "12","10","10");

        System.out.println(" "+time);

    }

}

**Output:**

25/06/2018 12:10:10

Java String join() Method Example 3

In the case of using null as a delimiter, we get the null pointer exception. The following example confirms the same.

**FileName:** StringJoinExample3.java

**public** **class** StringJoinExample3

{

// main method

**public** **static** **void** main(String argvs[])

{

String str = **null**;

str = String.join(**null**, "abc", "bcd", "apple");

System.out.println(str);

}

}

**Output:**

Exception in thread "main" java.lang.NullPointerException

at java.base/java.util.Objects.requireNonNull(Objects.java:221)

at java.base/java.lang.String.join(String.java:2393)

at StringJoinExample3.main(StringJoinExample3.java:7)

However, if the elements that have to be attached with the delimiter are *null* then, we get the ambiguity. It is because there are two join() methods, and *null* is acceptable for both types of the join() method. Observe the following example.

**FileName:** StringJoinExample4.java

**public** **class** StringJoinExample4

{

// main method

**public** **static** **void** main(String argvs[])

{

String str = **null**;

str = String.join("India", **null**);

System.out.println(str);

}

}

**Output:**

/StringJoinExample4.java:7: error: reference to join is ambiguous

str = String.join("India", null);

^

both method join(CharSequence,CharSequence...) in String and method join(CharSequence,Iterable<? extends CharSequence>) in String match

/StringJoinExample4.java:7: warning: non-varargs call of varargs method with inexact argument type for last parameter;

str = String.join("India", null);

^

cast to CharSequence for a varargs call

cast to CharSequence[] for a non-varargs call and to suppress this warning

1 error

1 warning

AD

Java String join() Method Example 4

If the elements that have to be attached with the delimiter have some strings, in which a few of them are null, then the null elements are treated as a normal string, and we do not get any exception or error. Let's understand it through an example.

**FileName:** StringJoinExample5.java

**public** **class** StringJoinExample5

{

// main method

**public** **static** **void** main(String argvs[])

{

String str = **null**;

// one of the element is null however it will be treated as normal string

str = String.join("-", **null**, " wake up ", " eat ", " write content for JTP ", " eat ", " sleep ");

System.out.println(str);

}

}

**Output:**

null- wake up - eat - write content for JTP - eat - sleep

lastIndexOf() Method

The **lastIndexOf()** Java Vector class method is used to get the index of the last occurrence of the specified element in the vector. There are two different types of Java lastIndexOf() method which can be differentiated depending on its parameter. These are:

Java Vector lastIndexOf(Object o) Method

Java Vector lastIndexOf(Object o, int index) Method

lastIndexOf(Object o) Method:

It returns the index of the last occurrence of the specified element in this vector. If the element is not found, it returns -1.

lastIndexOf(Object o, int index> c) Method:

This method is used to get the index of the last occurrence of the specified element in this vector. It starts searching for an element in the backward direction from the specified index. If the element is not found, it returns -1.

Syntax

Following is the declaration of **lastIndexOf()** method:

**public** **int** lastIndexOf(Object obj)

**public** **int** lastIndexOf(Object obj, **int** index)

AD

Parameter

|  |  |  |
| --- | --- | --- |
| **Parameter** | **Description** | **Required/Optional** |
| index | It is an index where to start searching for an element in the backward direction. | Required |
| obj | It is an element to search for. | Required |

Return

The **lastIndexOf()** method returns the index of the last occurrence of the specified element in this vector or returns -1 if the vector does not contain the element.

Exceptions

**IndexOutOfBoundsException**- This method has thrown an exception if the index of an array is out of range i.e. (index >= size()).

Example 1

**import** java.util.\*;

**public** **class** VectorLastIndexOfExample1 {

**public** **static** **void** main(String arg[]) {

        //Create an empty Vector

        Vector < Integer > in = **new** Vector < > ();

        //Add elements in the vector

        in.add(100);

        in.add(200);

        in.add(300);

        in.add(100);

        in.add(400);

            //Obtain an index of last occurrence of the specified element

        System.out.println("Index of element is: " +in.lastIndexOf(100));

          }

}

**Output:**

Index of element is: 3

Example 2

**import** java.util.\*;

**public** **class** VectorLastIndexOfExample2 {

**public** **static** **void** main(String arg[]) {

        //Create an empty vector

          Vector<String> vec = **new** Vector<>(4);

          //Add elements in the vector

          vec.add("Java");

          vec.add("JavaScript");

          vec.add("Android");

          vec.add("Python");

            //Obtain an index of the last occurrence of the specified element

        System.out.println("Index of element is: " +vec.lastIndexOf("C"));

        System.out.println("The element is not found.");

          }

}

**Output:**

Index of element is: -1

The element is not found.

Example 3

**import** java.util.\*;

**public** **class** VectorLastIndexOfExample3 {

**public** **static** **void** main(String arg[]) {

        //Create a first empty vector

          Vector<String> vec = **new** Vector<>(4);

          //Add elements in the first vector

          vec.add("Facebook");

          vec.add("Whatsapp");

          vec.add("Twitter");

          vec.add("Instagram");

          vec.add("Skype");

            //This would start searching of element in the backward direction from index -2

        System.out.println("Index of element is found at: " +vec.lastIndexOf("Skype", 6));

    }

}

**Output:**

Exception in thread "main" java.lang.IndexOutOfBoundsException: 6 >= 5

at java.base/java.util.Vector.lastIndexOf(Vector.java:469)

at myPackage.VectorLastIndexOfExample3.main(VectorLastIndexOfExample3.java:14)

Example 4

**import** java.util.\*;

**public** **class** VectorLastIndexOfExample4 {

**public** **static** **void** main(String arg[]) {

        //Create an empty Vector

        Vector < Integer > in = **new** Vector < > ();

        //Add elements in the vector

        in.add(101);

        in.add(201);

        in.add(301);

        in.add(401);

        in.add(501);

            //This would start searching of element in the backward direction from index 3

       System.out.println("Index of element is found at: " +in.lastIndexOf(201, 3));

    }

}

**Output:**

Index of element is found at: 1

length()

The **Java String class length()** method finds the length of a string. The length of the Java string is the same as the Unicode code units of the string.

Signature

The signature of the string length() method is given below:

**public** **int** length()

Specified by

CharSequence interface

Returns

Length of characters. In other words, the total number of characters present in the string.

Internal implementation

**public** **int** length() {

**return** value.length;

  }

The String class internally uses a char[] array to store the characters. The length variable of the array is used to find the total number of elements present in the array. Since the Java String class uses this char[] array internally; therefore, the length variable can not be exposed to the outside world. Hence, the Java developers created the length() method, the exposes the value of the length variable. One can also think of the length() method as the getter() method, that provides a value of the class field to the user. The internal implementation clearly depicts that the length() method returns the value of then the length variable.

Java String length() method example

**FileName:** LengthExample.java

**public** **class** LengthExample{

**public** **static** **void** main(String args[]){

String s1="javatpoint";

String s2="python";

System.out.println("string length is: "+s1.length());//10 is the length of javatpoint string

System.out.println("string length is: "+s2.length());//6 is the length of python string

}}

**Output:**

string length is: 10

string length is: 6

Java String length() Method Example 2

Since the length() method gives the total number of characters present in the string; therefore, one can also check whether the given string is empty or not.

**FileName:** LengthExample2.java

**public** **class** LengthExample2 {

**public** **static** **void** main(String[] args) {

        String str = "Javatpoint";

**if**(str.length()>0) {

            System.out.println("String is not empty and length is: "+str.length());

        }

        str = "";

**if**(str.length()==0) {

            System.out.println("String is empty now: "+str.length());

        }

    }

}

**Output:**

String is not empty and length is: 10

String is empty now: 0

Java String length() Method Example 3

The length() method is also used to reverse the string.

**FileName:** LengthExample3.java

**class** LengthExample3

{

// main method

**public** **static** **void** main(String argvs[])

{

String str = "Welcome To JavaTpoint";

**int** size = str.length();

System.out.println("Reverse of the string: " + "'" + str + "'" + " is");

**for**(**int** i = 0; i < size; i++)

{

// printing in reverse order

System.out.print(str.charAt(str.length() - i - 1));

}

}

}

**Output:**

Reverse of the string: 'Welcome To JavaTpoint' is

tniopTavaJ oT emocleW

AD

Java String length() Method Example 4

The length() method can also be used to find only the white spaces present in the string. Observe the following example.

**FileName:** LengthExample4.java

**public** **class** LengthExample4

{

// main method

**public** **static** **void** main(String argvs[])

{

String str = " Welcome To JavaTpoint ";

**int** sizeWithWhiteSpaces = str.length();

System.out.println("In the string: " + "'" + str + "'");

str = str.replace(" ", "");

**int** sizeWithoutWhiteSpaces = str.length();

// calculating the white spaces

**int** noOfWhieSpaces = sizeWithWhiteSpaces - sizeWithoutWhiteSpaces;

System.out.print("Total number of whitespaces present are: " + noOfWhieSpaces);

}

}

**Output:**

In the string: ' Welcome To JavaTpoint '

Total number of whitespaces present are: 4

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.

Signature

1. **public** String substring(**int** startIndex)  // type - 1
2. and
3. **public** String substring(**int** startIndex, **int** endIndex)  // type - 2

If we don't specify endIndex, the method will return all the characters from startIndex.

Parameters

**startIndex** : starting index is inclusive

**endIndex** : ending index is exclusive

Returns

specified string

Exception Throws

**StringIndexOutOfBoundsException** is thrown when any one of the following conditions is met.

* if the start index is negative value
* end index is lower than starting index.
* Either starting or ending index is greater than the total number of characters present in the string.

Internal implementation substring(int beginIndex)

1. **public** String substring(**int** beginIndex) {
2. **if** (beginIndex < 0) {
3. **throw** **new** StringIndexOutOfBoundsException(beginIndex);
4. }
5. **int** subLen = value.length - beginIndex;
6. **if** (subLen < 0) {
7. **throw** **new** StringIndexOutOfBoundsException(subLen);
8. }
9. **return** (beginIndex == 0) ? **this** : **new** String(value, beginIndex, subLen);
10. }

Internal implementation substring(int beginIndex, int endIndex)

1. **public** String substring(**int** beginIndex, **int** endIndex)
2. {
3. **if** (beginIndex < 0)
4. {
5. **throw** **new** StringIndexOutOfBoundsException(beginIndex);
6. }
7. **if** (endIndex > value.length)
8. {
9. **throw** **new** StringIndexOutOfBoundsException(endIndex);
10. }
11. **int** subLen = endIndex - beginIndex;
12. **if** (subLen < 0)
13. {
14. **throw** **new** StringIndexOutOfBoundsException(subLen);
15. }
16. **return** ((beginIndex == 0) && (endIndex == value.length)) ? **this** : **new** String(value, beginIndex, subLen);
17. }

Java String substring() method example

**FileName:** SubstringExample.java

1. **public** **class** SubstringExample{
2. **public** **static** **void** main(String args[]){
3. String s1="javatpoint";
4. System.out.println(s1.substring(2,4));//returns va
5. System.out.println(s1.substring(2));//returns vatpoint
6. }}

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=SubstringExample)

**Output:**

va

vatpoint

Java String substring() Method Example 2

**FileName:** SubstringExample2.java

1. **public** **class** SubstringExample2 {
2. **public** **static** **void** main(String[] args) {
3. String s1="Javatpoint";
4. String substr = s1.substring(0); // Starts with 0 and goes to end
5. System.out.println(substr);
6. String substr2 = s1.substring(5,10); // Starts from 5 and goes to 10
7. System.out.println(substr2);
8. String substr3 = s1.substring(5,15); // Returns Exception
9. }
10. }

**Output:**

Javatpoint

point

Exception in thread "main" java.lang.StringIndexOutOfBoundsException: begin 5, end 15, length 10

Applications of substring() Method

1) The substring() method can be used to do some prefix or suffix extraction. For example, we can have a list of names, and it is required to filter out names with surname as "singh". The following program shows the same.

**FileName:** SubstringExample3.java

1. **public** **class** SubstringExample3
2. {
3. // main method
4. **public** **static** **void** main(String argvs[])
5. {
6. String str[] =
7. {
8. "Praveen Kumar",
9. "Yuvraj Singh",
10. "Harbhajan Singh",
11. "Gurjit Singh",
12. "Virat Kohli",
13. "Rohit Sharma",
14. "Sandeep Singh",
15. "Milkha Singh"
16. };
18. String surName = "Singh";
19. **int** surNameSize = surName.length();
21. **int** size = str.length;
23. **for**(**int** j = 0; j < size; j++)
24. {
25. **int** length = str[j].length();
26. // extracting the surname
27. String subStr = str[j].substring(length - surNameSize);
29. // checks whether the surname is equal to "Singh" or not
30. **if**(subStr.equals(surName))
31. {
32. System.out.println(str[j]);
33. }
34. }
36. }
37. }

**Output:**

Yuvraj Singh

Harbhajan Singh

Gurjit Singh

Sandeep Singh

Milkha Singh

2) The substring() method can also be used to check whether a string is a palindrome or not.

**FileName:** SubstringExample4.java

1. **public** **class** SubstringExample4
2. {
3. **public** **boolean** isPalindrome(String str)
4. {
5. **int** size = str.length();
7. // handling the base case
8. **if**(size == 0 || size == 1)
9. {
10. // an empty string
11. // or a string of only one character
12. // is always a palindrome
13. **return** **true**;
14. }
15. String f = str.substring(0, 1);
16. String l = str.substring(size - 1);
17. // comparing first and the last character of the string
18. **if**(l.equals(f))
19. {
20. // recursively finding the solution using the substring() method
21. // reducing the number of characters of the by 2 for the next recursion
22. **return** isPalindrome(str.substring(1, size - 1));
23. }
24. **return** **false**;
25. }
26. // main method
27. **public** **static** **void** main(String argvs[])
28. {
29. // instantiating the class SubstringExample4
30. SubstringExample4 obj = **new** SubstringExample4();
31. String str[] =
32. {
33. "madam",
34. "rock",
35. "eye",
36. "noon",
37. "kill"
38. };
39. **int** size = str.length;
41. **for**(**int** j = 0; j < size; j++)
42. {
43. **if**(obj.isPalindrome(str[j]))
44. {
45. System.out.println(str[j] + " is a palindrome.");
46. }
47. **else**
48. {
49. System.out.println(str[j] + " is not a palindrome.");
50. }
51. }
52. }
53. }

**Output:**

madam is a palindrome.

rock is not a palindrome.

eye is a palindrome.

noon is a palindrome.

kill is not a palindrome.

split()

The **java string split()** method splits this string against given regular expression and returns a char array.

Internal implementation

1. **public** String[] split(String regex, **int** limit) {
2. /\* fastpath if the regex is a
3. (1)one-char String and this character is not one of the
4. RegEx's meta characters ".$|()[{^?\*+\\", or
5. (2)two-char String and the first char is the backslash and
6. the second is not the ascii digit or ascii letter.
7. \*/
8. **char** ch = 0;
9. **if** (((regex.value.length == 1 &&
10. ".$|()[{^?\*+\\".indexOf(ch = regex.charAt(0)) == -1) ||
11. (regex.length() == 2 &&
12. regex.charAt(0) == '\\' &&
13. (((ch = regex.charAt(1))-'0')|('9'-ch)) < 0 &&
14. ((ch-'a')|('z'-ch)) < 0 &&
15. ((ch-'A')|('Z'-ch)) < 0)) &&
16. (ch < Character.MIN\_HIGH\_SURROGATE ||
17. ch > Character.MAX\_LOW\_SURROGATE))
18. {
19. **int** off = 0;
20. **int** next = 0;
21. **boolean** limited = limit > 0;
22. ArrayList<String> list = **new** ArrayList<>();
23. **while** ((next = indexOf(ch, off)) != -1) {
24. **if** (!limited || list.size() < limit - 1) {
25. list.add(substring(off, next));
26. off = next + 1;
27. } **else** {    // last one
28. //assert (list.size() == limit - 1);
29. list.add(substring(off, value.length));
30. off = value.length;
31. **break**;
32. }
33. }
34. // If no match was found, return this
35. **if** (off == 0)
36. **return** **new** String[]{**this**};
38. // Add remaining segment
39. **if** (!limited || list.size() < limit)
40. list.add(substring(off, value.length));
42. // Construct result
43. **int** resultSize = list.size();
44. **if** (limit == 0)
45. **while** (resultSize > 0 && list.get(resultSize - 1).length() == 0)
46. resultSize--;
47. String[] result = **new** String[resultSize];
48. **return** list.subList(0, resultSize).toArray(result);
49. }
50. **return** Pattern.compile(regex).split(**this**, limit);
51. }

Signature

There are two signature for split() method in java string.

1. **public** String split(String regex)
2. and,
3. **public** String split(String regex, **int** limit)

Parameter

**regex** : regular expression to be applied on string.

**limit** : limit for the number of strings in array. If it is zero, it will returns all the strings matching regex.

Returns

array of strings

Throws

**PatternSyntaxException** if pattern for regular expression is invalid

Since

1.4

Java String split() method example

The given example returns total number of words in a string excluding space only. It also includes special characters.

1. **public** **class** SplitExample{
2. **public** **static** **void** main(String args[]){
3. String s1="java string split method by javatpoint";
4. String[] words=s1.split("\\s");//splits the string based on whitespace
5. //using java foreach loop to print elements of string array
6. **for**(String w:words){
7. System.out.println(w);
8. }
9. }}

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=SplitExample)

java

string

split

method

by

javatpoint

Java String split() method with regex and length example

1. **public** **class** SplitExample2{
2. **public** **static** **void** main(String args[]){
3. String s1="welcome to split world";
4. System.out.println("returning words:");
5. **for**(String w:s1.split("\\s",0)){
6. System.out.println(w);
7. }
8. System.out.println("returning words:");
9. **for**(String w:s1.split("\\s",1)){
10. System.out.println(w);
11. }
12. System.out.println("returning words:");
13. **for**(String w:s1.split("\\s",2)){
14. System.out.println(w);
15. }
17. }}

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=SplitExample2)

returning words:

welcome

to

split

world

returning words:

welcome to split world

returning words:

welcome

to split world

Java String split() method with regex and length example 2

Here, we are passing split limit as a second argument to this function. This limits the number of splitted strings.

1. **public** **class** SplitExample3 {
2. **public** **static** **void** main(String[] args) {
3. String str = "Javatpointtt";
4. System.out.println("Returning words:");
5. String[] arr = str.split("t", 0);
6. **for** (String w : arr) {
7. System.out.println(w);
8. }
9. System.out.println("Split array length: "+arr.length);
10. }
11. }

Returning words:

Java

poin

Split array length: 2

# trim()

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

The string trim() method doesn't omit middle spaces.

Signature

The signature or syntax of the String class trim() method is given below:

1. **public** String trim()

Returns

string with omitted leading and trailing spaces

Internal implementation

1. **public** String trim() {
2. **int** len = value.length;
3. **int** st = 0;
4. **char**[] val = value;    /\* avoid getfield opcode \*/
6. **while** ((st < len) && (val[st] <= ' ')) {
7. st++;
8. }
9. **while** ((st < len) && (val[len - 1] <= ' ')) {
10. len--;
11. }
12. **return** ((st > 0) || (len < value.length)) ? substring(st, len) : **this**;
13. }

Java String trim() Method Example

**FileName:** StringTrimExample.java

1. **public** **class** StringTrimExample{
2. **public** **static** **void** main(String args[]){
3. String s1="  hello string   ";
4. System.out.println(s1+"javatpoint");//without trim()
5. System.out.println(s1.trim()+"javatpoint");//with trim()
6. }}

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=StringTrimExample)

**Output**

hello string javatpoint

hello stringjavatpoint

Java String trim() Method Example 2

The example demonstrates the use of the trim() method. This method removes all the trailing spaces so the length of the string also reduces. Let's see an example.

**FileName:** StringTrimExample2.java

1. **public** **class** StringTrimExample2 {
2. **public** **static** **void** main(String[] args) {
3. String s1 ="  hello java string   ";
4. System.out.println(s1.length());
5. System.out.println(s1); //Without trim()
6. String tr = s1.trim();
7. System.out.println(tr.length());
8. System.out.println(tr); //With trim()
9. }
10. }

**Output**

22

hello java string

17

hello java string

Java String trim() Method Example 3

The trim() can be used to check whether the string only contains white spaces or not. The following example shows the same.

**FileName:** TrimExample3.java

1. **public** **class** TrimExample3
2. {
3. // main method
4. **public** **static** **void** main(String argvs[])
5. {
7. String str = " abc ";
9. **if**((str.trim()).length() > 0)
10. {
11. System.out.println("The string contains characters other than white spaces \n");
12. }
13. **else**
14. {
15. System.out.println("The string contains only white spaces \n");
16. }
18. str = "    ";
20. **if**((str.trim()).length() > 0)
21. {
22. System.out.println("The string contains characters other than white spaces \n");
23. }
24. **else**
25. {
26. System.out.println("The string contains only white spaces \n");
27. }
29. }
30. }

**Output**

The string contains characters other than white spaces

The string contains only white spaces

AD

Java String trim() Method Example 4

Since strings in Java are immutable; therefore, when the trim() method manipulates the string by trimming the whitespaces, it returns a new string. If the manipulation is not done by the trim() method, then the reference of the same string is returned. Observe the following example.

**FileName:** TrimExample4.java

1. **public** **class** TrimExample4
2. {
3. // main method
4. **public** **static** **void** main(String argvs[])
5. {
7. // the string contains white spaces
8. // therefore, trimming the spaces leads to the
9. // generation of new string
10. String str = " abc ";
12. // str1 stores a new string
13. String str1 = str.trim();
15. // the hashcode of str and str1 is different
16. System.out.println(str.hashCode());
17. System.out.println(str1.hashCode() + "\n");
19. // no white space present in the string s
20. // therefore, the reference of the s is returned
21. // when the trim() method is invoked
22. String s = "xyz";
23. String s1 = s.trim();
25. // the hashcode of s and s1 is the same
26. System.out.println(s.hashCode());
27. System.out.println(s1.hashCode());
29. }
30. }

**Output**

The string contains characters other than white spaces

The string contains only white spaces