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)

**public** String substring(**int** beginIndex) {

**if** (beginIndex < 0) {

**throw** **new** StringIndexOutOfBoundsException(beginIndex);

       }

**int** subLen = value.length - beginIndex;

**if** (subLen < 0) {

**throw** **new** StringIndexOutOfBoundsException(subLen);

       }

**return** (beginIndex == 0) ? **this** : **new** String(value, beginIndex, subLen);

   }

Internal implementation substring(int beginIndex, int endIndex)

**public** String substring(**int** beginIndex, **int** endIndex)

{

**if** (beginIndex < 0)

{

**throw** **new** StringIndexOutOfBoundsException(beginIndex);

}

**if** (endIndex > value.length)

{

**throw** **new** StringIndexOutOfBoundsException(endIndex);

}

**int** subLen = endIndex - beginIndex;

**if** (subLen < 0)

{

**throw** **new** StringIndexOutOfBoundsException(subLen);

}

**return** ((beginIndex == 0) && (endIndex == value.length)) ? **this** : **new** String(value, beginIndex, subLen);

}

Java String substring() method example

**FileName:** SubstringExample.java

**public** **class** SubstringExample{

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

String s1="javatpoint";

System.out.println(s1.substring(2,4));//returns va

System.out.println(s1.substring(2));//returns vatpoint

}}

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

**Output:**

va

vatpoint

Java String substring() Method Example 2

**FileName:** SubstringExample2.java

**public** **class** SubstringExample2 {

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

        String s1="Javatpoint";

        String substr = s1.substring(0); // Starts with 0 and goes to end

        System.out.println(substr);

        String substr2 = s1.substring(5,10); // Starts from 5 and goes to 10

        System.out.println(substr2);

        String substr3 = s1.substring(5,15); // Returns Exception

    }

}

**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

**public** **class** SubstringExample3

{

// main method

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

{

String str[] =

{

"Praveen Kumar",

"Yuvraj Singh",

"Harbhajan Singh",

"Gurjit Singh",

"Virat Kohli",

"Rohit Sharma",

"Sandeep Singh",

"Milkha Singh"

};

String surName = "Singh";

**int** surNameSize = surName.length();

**int** size = str.length;

**for**(**int** j = 0; j < size; j++)

{

**int** length = str[j].length();

    // extracting the surname

    String subStr = str[j].substring(length - surNameSize);

    // checks whether the surname is equal to "Singh" or not

**if**(subStr.equals(surName))

    {

        System.out.println(str[j]);

    }

}

}

}

**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

**public** **class** SubstringExample4

{

**public** **boolean** isPalindrome(String str)

{

**int** size = str.length();

// handling the base case

**if**(size == 0 || size == 1)

{

// an empty string

// or a string of only one character

// is always a palindrome

**return** **true**;

}

String f = str.substring(0, 1);

String l = str.substring(size - 1);

// comparing first and the last character of the string

**if**(l.equals(f))

{

// recursively finding the solution using the substring() method

// reducing the number of characters of the by 2 for the next recursion

**return** isPalindrome(str.substring(1, size - 1));

}

**return** **false**;

}

// main method

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

{

// instantiating the class SubstringExample4

SubstringExample4 obj = **new** SubstringExample4();

String str[] =

{

"madam",

"rock",

"eye",

"noon",

"kill"

};

**int** size = str.length;

**for**(**int** j = 0; j < size; j++)

{

**if**(obj.isPalindrome(str[j]))

{

System.out.println(str[j] + " is a palindrome.");

}

**else**

{

System.out.println(str[j] + " is not a palindrome.");

}

}

}

}

**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

**public** String[] split(String regex, **int** limit) {

        /\* fastpath if the regex is a

         (1)one-char String and this character is not one of the

            RegEx's meta characters ".$|()[{^?\*+\\", or

         (2)two-char String and the first char is the backslash and

            the second is not the ascii digit or ascii letter.

         \*/

**char** ch = 0;

**if** (((regex.value.length == 1 &&

             ".$|()[{^?\*+\\".indexOf(ch = regex.charAt(0)) == -1) ||

             (regex.length() == 2 &&

              regex.charAt(0) == '\\' &&

              (((ch = regex.charAt(1))-'0')|('9'-ch)) < 0 &&

              ((ch-'a')|('z'-ch)) < 0 &&

              ((ch-'A')|('Z'-ch)) < 0)) &&

            (ch < Character.MIN\_HIGH\_SURROGATE ||

             ch > Character.MAX\_LOW\_SURROGATE))

        {

**int** off = 0;

**int** next = 0;

**boolean** limited = limit > 0;

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

**while** ((next = indexOf(ch, off)) != -1) {

**if** (!limited || list.size() < limit - 1) {

                    list.add(substring(off, next));

                    off = next + 1;

                } **else** {    // last one

                    //assert (list.size() == limit - 1);

                    list.add(substring(off, value.length));

                    off = value.length;

**break**;

                }

            }

            // If no match was found, return this

**if** (off == 0)

**return** **new** String[]{**this**};

            // Add remaining segment

**if** (!limited || list.size() < limit)

                list.add(substring(off, value.length));

            // Construct result

**int** resultSize = list.size();

**if** (limit == 0)

**while** (resultSize > 0 && list.get(resultSize - 1).length() == 0)

                    resultSize--;

            String[] result = **new** String[resultSize];

**return** list.subList(0, resultSize).toArray(result);

        }

**return** Pattern.compile(regex).split(**this**, limit);

    }

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.

**public** **class** SplitExample{

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

String s1="java string split method by javatpoint";

String[] words=s1.split("\\s");//splits the string based on whitespace

//using java foreach loop to print elements of string array

**for**(String w:words){

System.out.println(w);

}

}}

[**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

**public** **class** SplitExample2{

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

String s1="welcome to split world";

System.out.println("returning words:");

**for**(String w:s1.split("\\s",0)){

System.out.println(w);

}

System.out.println("returning words:");

**for**(String w:s1.split("\\s",1)){

System.out.println(w);

}

System.out.println("returning words:");

**for**(String w:s1.split("\\s",2)){

System.out.println(w);

}

}}

[**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.

**public** **class** SplitExample3 {

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

        String str = "Javatpointtt";

        System.out.println("Returning words:");

        String[] arr = str.split("t", 0);

**for** (String w : arr) {

            System.out.println(w);

        }

        System.out.println("Split array length: "+arr.length);

    }

}

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

**public** String trim() {

**int** len = value.length;

**int** st = 0;

**char**[] val = value;    /\* avoid getfield opcode \*/

**while** ((st < len) && (val[st] <= ' ')) {

            st++;

        }

**while** ((st < len) && (val[len - 1] <= ' ')) {

            len--;

        }

**return** ((st > 0) || (len < value.length)) ? substring(st, len) : **this**;

    }

Java String trim() Method Example

**FileName:** StringTrimExample.java

**public** **class** StringTrimExample{

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

String s1="  hello string   ";

System.out.println(s1+"javatpoint");//without trim()

System.out.println(s1.trim()+"javatpoint");//with trim()

}}

[**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

**public** **class** StringTrimExample2 {

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

        String s1 ="  hello java string   ";

        System.out.println(s1.length());

        System.out.println(s1); //Without trim()

        String tr = s1.trim();

        System.out.println(tr.length());

        System.out.println(tr); //With trim()

    }

}

**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

**public** **class** TrimExample3

{

// main method

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

{

String str = " abc ";

**if**((str.trim()).length() > 0)

{

System.out.println("The string contains characters other than white spaces \n");

}

**else**

{

System.out.println("The string contains only white spaces \n");

}

str = "    ";

**if**((str.trim()).length() > 0)

{

System.out.println("The string contains characters other than white spaces \n");

}

**else**

{

System.out.println("The string contains only white spaces \n");

}

}

}

**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

**public** **class** TrimExample4

{

// main method

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

{

// the string contains white spaces

// therefore, trimming the spaces leads to the

// generation of new string

String str = " abc ";

// str1 stores a new string

String str1 = str.trim();

// the hashcode of str and str1 is different

System.out.println(str.hashCode());

System.out.println(str1.hashCode() + "\n");

// no white space present in the string s

// therefore, the reference of the s is returned

// when the trim() method is invoked

String s = "xyz";

String s1 = s.trim();

// the hashcode of s and s1 is the same

System.out.println(s.hashCode());

System.out.println(s1.hashCode());

}

}

**Output**

The string contains characters other than white spaces

The string contains only white spaces

Java StringBuffer Class

Java StringBuffer class is used to create mutable (modifiable) String objects. The StringBuffer class in Java is the same as String class except it is mutable i.e. it can be changed.

Note: Java StringBuffer class is thread-safe i.e. multiple threads cannot access it simultaneously. So it is safe and will result in an order.

Important Constructors of StringBuffer Class

|  |  |
| --- | --- |
| **Constructor** | **Description** |
| StringBuffer() | It creates an empty String buffer with the initial capacity of 16. |
| StringBuffer(String str) | It creates a String buffer with the specified string.. |
| StringBuffer(int capacity) | It creates an empty String buffer with the specified capacity as length. |

What is a mutable String?

A String that can be modified or changed is known as mutable String. StringBuffer and StringBuilder classes are used for creating mutable strings.

1) StringBuffer Class append() Method

The append() method concatenates the given argument with this String.

**StringBufferExample.java**

**class** StringBufferExample{

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

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

sb.append("Java");//now original string is changed

System.out.println(sb);//prints Hello Java

}

}

**Output:**

Hello Java

2) StringBuffer insert() Method

The insert() method inserts the given String with this string at the given position.

**StringBufferExample2.java**

**class** StringBufferExample2{

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

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

sb.insert(1,"Java");//now original string is changed

System.out.println(sb);//prints HJavaello

}

}

**Output:**

HJavaello

3) StringBuffer replace() Method

The replace() method replaces the given String from the specified beginIndex and endIndex.

**StringBufferExample3.java**

**class** StringBufferExample3{

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

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

sb.replace(1,3,"Java");

System.out.println(sb);//prints HJavalo

}

}

**Output:**

HJavalo

4) StringBuffer delete() Method

The delete() method of the StringBuffer class deletes the String from the specified beginIndex to endIndex.

**StringBufferExample4.java**

**class** StringBufferExample4{

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

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

sb.delete(1,3);

System.out.println(sb);//prints Hlo

}

}

**Output:**

Hlo

5) StringBuffer reverse() Method

The reverse() method of the StringBuilder class reverses the current String.

**StringBufferExample5.java**

**class** StringBufferExample5{

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

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

sb.reverse();

System.out.println(sb);//prints olleH

}

}

**Output:**

olleH

6) StringBuffer capacity() Method

The capacity() method of the StringBuffer class returns the current capacity of the buffer. The default capacity of the buffer is 16. If the number of character increases from its current capacity, it increases the capacity by (oldcapacity\*2)+2. For example if your current capacity is 16, it will be (16\*2)+2=34.

**StringBufferExample6.java**

**class** StringBufferExample6{

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

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

System.out.println(sb.capacity());//default 16

sb.append("Hello");

System.out.println(sb.capacity());//now 16

sb.append("java is my favourite language");

System.out.println(sb.capacity());//now (16\*2)+2=34 i.e (oldcapacity\*2)+2

}

}

**Output:**

16

16

34

7) StringBuffer ensureCapacity() method

The ensureCapacity() method of the StringBuffer class ensures that the given capacity is the minimum to the current capacity. If it is greater than the current capacity, it increases the capacity by (oldcapacity\*2)+2. For example if your current capacity is 16, it will be (16\*2)+2=34.

**StringBufferExample7.java**

**class** StringBufferExample7{

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

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

System.out.println(sb.capacity());//default 16

sb.append("Hello");

System.out.println(sb.capacity());//now 16

sb.append("java is my favourite language");

System.out.println(sb.capacity());//now (16\*2)+2=34 i.e (oldcapacity\*2)+2

sb.ensureCapacity(10);//now no change

System.out.println(sb.capacity());//now 34

sb.ensureCapacity(50);//now (34\*2)+2

System.out.println(sb.capacity());//now 70

}

}

**Output:**

16

16

34

34

70

# Exception Handling in Java

The **Exception Handling in Java** is one of the powerful mechanism to handle the runtime errors so that the normal flow of the application can be maintained.

In this tutorial, we will learn about Java exceptions, it's types, and the difference between checked and unchecked exceptions.

## What is Exception in Java?

**Dictionary Meaning:** Exception is an abnormal condition.

In Java, an exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.

## What is Exception Handling?

Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException, IOException, SQLException, RemoteException, etc.

### Advantage of Exception Handling

The core advantage of exception handling is **to maintain the normal flow of the application**. An exception normally disrupts the normal flow of the application; that is why we need to handle exceptions. Let's consider a scenario:

1. statement 1;
2. statement 2;
3. statement 3;
4. statement 4;
5. statement 5;//exception occurs
6. statement 6;
7. statement 7;
8. statement 8;
9. statement 9;
10. statement 10;

Suppose there are 10 statements in a Java program and an exception occurs at statement 5; the rest of the code will not be executed, i.e., statements 6 to 10 will not be executed. However, when we perform exception handling, the rest of the statements will be executed. That is why we use exception handling in [Java](https://www.javatpoint.com/java-tutorial).

Do You Know?

|  |
| --- |
| * What is the difference between checked and unchecked exceptions? * What happens behind the code int data=50/0;? * Why use multiple catch block? * Is there any possibility when the finally block is not executed? * What is exception propagation? * What is the difference between the throw and throws keyword? * What are the 4 rules for using exception handling with method overriding? |

## Hierarchy of Java Exception classes

The java.lang.Throwable class is the root class of Java Exception hierarchy inherited by two subclasses: Exception and Error. The hierarchy of Java Exception classes is given below:



### Types of Java Exceptions

There are mainly two types of exceptions: checked and unchecked. An error is considered as the unchecked exception. However, according to Oracle, there are three types of exceptions namely:

1. Checked Exception
2. Unchecked Exception
3. Error



## Difference between Checked and Unchecked Exceptions

### 1) Checked Exception

The classes that directly inherit the Throwable class except RuntimeException and Error are known as checked exceptions. For example, IOException, SQLException, etc. Checked exceptions are checked at compile-time.

### 2) Unchecked Exception

The classes that inherit the RuntimeException are known as unchecked exceptions. For example, ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException, etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

### 3) Error

Error is irrecoverable. Some example of errors are OutOfMemoryError, VirtualMachineError, AssertionError etc.

## Java Exception Keywords

Java provides five keywords that are used to handle the exception. The following table describes each.

|  |  |
| --- | --- |
| **Keyword** | **Description** |
| try | The "try" keyword is used to specify a block where we should place an exception code. It means we can't use try block alone. The try block must be followed by either catch or finally. |
| catch | The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later. |
| finally | The "finally" block is used to execute the necessary code of the program. It is executed whether an exception is handled or not. |
| throw | The "throw" keyword is used to throw an exception. |
| throws | The "throws" keyword is used to declare exceptions. It specifies that there may occur an exception in the method. It doesn't throw an exception. It is always used with method signature. |

## Java Exception Handling Example

Let's see an example of Java Exception Handling in which we are using a try-catch statement to handle the exception.

**JavaExceptionExample.java**

**public** **class** JavaExceptionExample{

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

**try**{

      //code that may raise exception

**int** data=100/0;

   }**catch**(ArithmeticException e){System.out.println(e);}

   //rest code of the program

   System.out.println("rest of the code...");

  }

}

**Output:**

Exception in thread main java.lang.ArithmeticException:/ by zero

rest of the code...

In the above example, 100/0 raises an ArithmeticException which is handled by a try-catch block.

## Common Scenarios of Java Exceptions

There are given some scenarios where unchecked exceptions may occur. They are as follows:

### 1) A scenario where ArithmeticException occurs

If we divide any number by zero, there occurs an ArithmeticException.

1. **int** a=50/0;//ArithmeticException

### 2) A scenario where NullPointerException occurs

If we have a null value in any [variable](https://www.javatpoint.com/java-variables), performing any operation on the variable throws a NullPointerException.

1. String s=**null**;
2. System.out.println(s.length());//NullPointerException

### 3) A scenario where NumberFormatException occurs

If the formatting of any variable or number is mismatched, it may result into NumberFormatException. Suppose we have a [string](https://www.javatpoint.com/java-string) variable that has characters; converting this variable into digit will cause NumberFormatException.

1. String s="abc";
2. **int** i=Integer.parseInt(s);//NumberFormatException

### 4) A scenario where ArrayIndexOutOfBoundsException occurs

When an array exceeds to it's size, the ArrayIndexOutOfBoundsException occurs. there may be other reasons to occur ArrayIndexOutOfBoundsException. Consider the following statements.

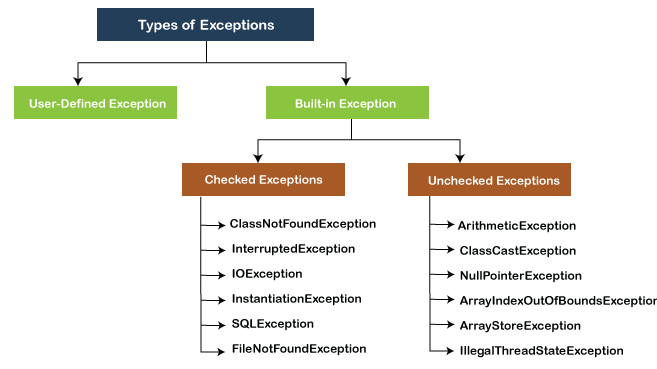
1. **int** a[]=**new** **int**[5];
2. a[10]=50; //ArrayIndexOutOfBoundsException

# Types of Exception in Java

In Java, **exception** is an event that occurs during the execution of a program and disrupts the normal flow of the program's instructions. Bugs or errors that we don't want and restrict our program's normal execution of code are referred to as **exceptions**. In this section, we will focus on the **types of exceptions in Java** and the differences between the two.

Exceptions can be categorized into two ways:

1. Built-in Exceptions
   * Checked Exception
   * Unchecked Exception
2. User-Defined Exceptions



## Built-in Exception

[Exceptions](https://www.javatpoint.com/exception-handling-in-java) that are already available in **Java libraries** are referred to as **built-in exception**. These exceptions are able to define the error situation so that we can understand the reason of getting this error. It can be categorized into two broad categories, i.e., **checked exceptions** and **unchecked exception**.

### Checked Exception

**Checked** exceptions are called **compile-time** exceptions because these exceptions are checked at compile-time by the compiler. The compiler ensures whether the programmer handles the exception or not. The programmer should have to handle the exception; otherwise, the system has shown a compilation error.

**CheckedExceptionExample.java**

**import** java.io.\*;

**class** CheckedExceptionExample {

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

        FileInputStream file\_data = **null**;

        file\_data = **new** FileInputStream("C:/Users/ajeet/OneDrive/Desktop/Hello.txt");

**int** m;

**while**(( m = file\_data.read() ) != -1) {

            System.out.print((**char**)m);

        }

        file\_data.close();

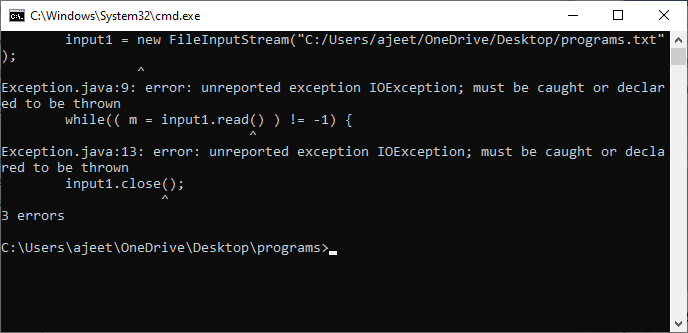
    }

}

In the above code, we are trying to read the **Hello.txt** file and display its data or content on the screen. The program throws the following exceptions:

1. The **FileInputStream(File filename)** constructor throws the **FileNotFoundException** that is checked exception.
2. The **read()** method of the **FileInputStream** class throws the **IOException**.
3. The **close()** method also throws the IOException.

**Output:**



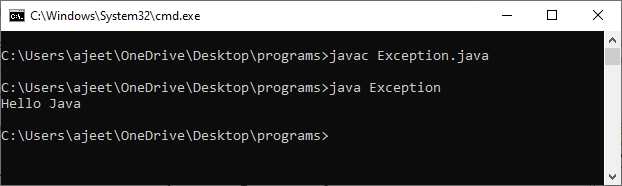
### How to resolve the error?

There are basically two ways through which we can solve these errors.

1) The exceptions occur in the main method. We can get rid from these compilation errors by declaring the exception in the main method using **the throws** We only declare the IOException, not FileNotFoundException, because of the child-parent relationship. The IOException class is the parent class of FileNotFoundException, so this exception will automatically cover by IOException. We will declare the exception in the following way:

1. **class** Exception{
2. **public** **static** **void** main(String args[])  **throws** IOException {
3. ...
4. ...
5. }

If we compile and run the code, the errors will disappear, and we will see the data of the file.



2) We can also handle these exception using **try-catch** However, the way which we have used above is not correct. We have to a give meaningful message for each exception type. By doing that it would be easy to understand the error. We will use the try-catch block in the following way:

**Exception.java**

**import** java.io.\*;

**class** Exception{

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

        FileInputStream file\_data = **null**;

**try**{

            file\_data = **new** FileInputStream("C:/Users/ajeet/OneDrive/Desktop/programs/Hell.txt");

        }**catch**(FileNotFoundException fnfe){

            System.out.println("File Not Found!");

        }

**int** m;

**try**{

**while**(( m = file\_data.read() ) != -1) {

                System.out.print((**char**)m);

            }

            file\_data.close();

        }**catch**(IOException ioe){

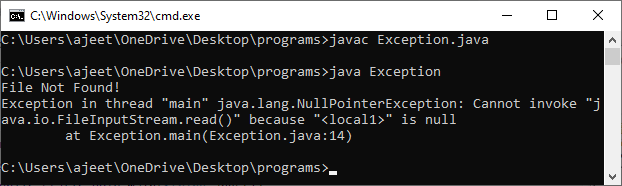
            System.out.println("I/O error occurred: "+ioe);

        }

    }

}

We will see a proper error message **"File Not Found!"** on the console because there is no such file in that location.



### Unchecked Exceptions

The **unchecked** exceptions are just opposite to the **checked** exceptions. The compiler will not check these exceptions at compile time. In simple words, if a program throws an unchecked exception, and even if we didn't handle or declare it, the program would not give a compilation error. Usually, it occurs when the user provides bad data during the interaction with the program.

#### Note: The RuntimeException class is able to resolve all the unchecked exceptions because of the child-parent relationship.

**UncheckedExceptionExample1.java**

**class** UncheckedExceptionExample1 {

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

   {

**int** postive = 35;

**int** zero = 0;

**int** result = positive/zero;

    //Give Unchecked Exception here.

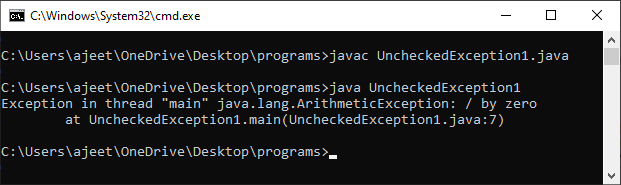
System.out.println(result);

   }

}

In the above program, we have divided 35 by 0. The code would be compiled successfully, but it will throw an ArithmeticException error at runtime. On dividing a number by 0 throws the divide by zero exception that is a uncheck exception.

**Output:**



**UncheckedException1.java**

**class** UncheckedException1 {

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

   {

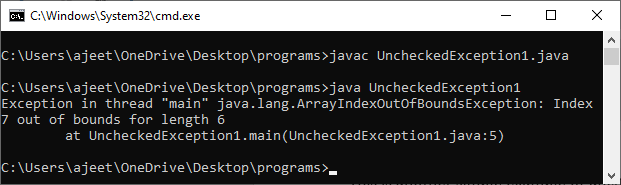
**int** num[] ={10,20,30,40,50,60};

    System.out.println(num[7]);

   }

}

**Output:**



In the above code, we are trying to get the element located at position 7, but the length of the array is 6. The code compiles successfully, but throws the ArrayIndexOutOfBoundsException at runtime.

## User-defined Exception

In [Java](https://www.javatpoint.com/java-tutorial), we already have some built-in exception classes like [**ArrayIndexOutOfBoundsException**](https://www.javatpoint.com/arrayindexoutofboundsexception-in-java)**, NullPointerException**, and **ArithmeticException**. These exceptions are restricted to trigger on some predefined conditions. In Java, we can write our own exception class by extends the Exception class. We can throw our own exception on a particular condition using the throw keyword. For creating a user-defined exception, we should have basic knowledge of **the**[**try-catch**](https://www.javatpoint.com/try-catch-block) block and [**throw** keyword](https://www.javatpoint.com/throw-keyword).

Let's write a [Java program](https://www.javatpoint.com/java-programs) and create user-defined exception.

**UserDefinedException.java**

**import** java.util.\*;

**class** UserDefinedException{

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

**try**{

**throw** **new** NewException(5);

        }

**catch**(NewException ex){

            System.out.println(ex) ;

        }

    }

}

**class** NewException **extends** Exception{

**int** x;

    NewException(**int** y) {

        x=y;

    }

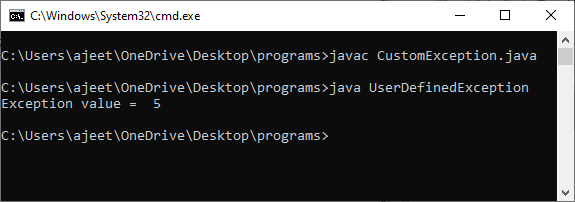
**public** String toString(){

**return** ("Exception value =  "+x) ;

    }

}

**Output:**



**Description:**

In the above code, we have created two classes, i.e., **UserDefinedException** and **NewException**. The **UserDefinedException** has our main method, and the **NewException** class is our user-defined exception class, which extends **exception**. In the **NewException** class, we create a variable **x** of type integer and assign a value to it in the constructor. After assigning a value to that variable, we return the exception message.

In the **UserDefinedException** class, we have added a **try-catch** block. In the try section, we throw the exception, i.e., **NewException** and pass an integer to it. The value will be passed to the NewException class and return a message. We catch that message in the catch block and show it on the screen.

## Difference Between Checked and Unchecked Exception

|  |  |  |
| --- | --- | --- |
| **S.No** | **Checked Exception** | **Unchecked Exception** |
| 1. | These exceptions are checked at compile time. These exceptions are handled at compile time too. | These exceptions are just opposite to the checked exceptions. These exceptions are not checked and handled at compile time. |
| 2. | These exceptions are direct subclasses of exception but not extended from RuntimeException class. | They are the direct subclasses of the RuntimeException class. |
| 3. | The code gives a compilation error in the case when a method throws a checked exception. The compiler is not able to handle the exception on its own. | The code compiles without any error because the exceptions escape the notice of the compiler. These exceptions are the results of user-created errors in programming logic. |
| 4. | These exceptions mostly occur when the probability of failure is too high. | These exceptions occur mostly due to programming mistakes. |
| 5. | Common checked exceptions include IOException, DataAccessException, InterruptedException, etc. | Common unchecked exceptions include ArithmeticException, InvalidClassException, NullPointerException, etc. |
| 6. | These exceptions are propagated using the throws keyword. | These are automatically propagated. |
| 7. | It is required to provide the try-catch and try-finally block to handle the checked exception. | In the case of unchecked exception it is not mandatory. |

Bugs or errors that we don't want and restrict the normal execution of the programs are referred to as **exceptions**.

**ArithmeticException, ArrayIndexOutOfBoundExceptions, ClassNotFoundExceptions** etc. are come in the category of **Built-in Exception**. Sometimes, the built-in exceptions are not sufficient to explain or describe certain situations. For describing these situations, we have to create our own exceptions by creating an exception class as a subclass of the **Exception** class. These types of exceptions come in the category of **User-Defined Exception**.

# Difference between throw and throws in Java

The throw and throws is the concept of exception handling where the throw keyword throw the exception explicitly from a method or a block of code whereas the throws keyword is used in signature of the method.

There are many differences between [throw](https://www.javatpoint.com/throw-keyword) and [throws](https://www.javatpoint.com/throws-keyword-and-difference-between-throw-and-throws) keywords. A list of differences between throw and throws are given below:

|  |  |  |  |
| --- | --- | --- | --- |
| **Sr. no.** | **Basis of Differences** | **throw** | **throws** |
| 1. | Definition | Java throw keyword is used throw an exception explicitly in the code, inside the function or the block of code. | Java throws keyword is used in the method signature to declare an exception which might be thrown by the function while the execution of the code. |
| 2. | Type of exception Using throw keyword, we can only propagate unchecked exception i.e., the checked exception cannot be propagated using throw only. | Using throws keyword, we can declare both checked and unchecked exceptions. However, the throws keyword can be used to propagate checked exceptions only. |  |
| 3. | Syntax | The throw keyword is followed by an instance of Exception to be thrown. | The throws keyword is followed by class names of Exceptions to be thrown. |
| 4. | Declaration | throw is used within the method. | throws is used with the method signature. |
| 5. | Internal implementation | We are allowed to throw only one exception at a time i.e. we cannot throw multiple exceptions. | We can declare multiple exceptions using throws keyword that can be thrown by the method. For example, main() throws IOException, SQLException. |

## Java throw Example

**TestThrow.java**

**public** **class** TestThrow {

    //defining a method

**public** **static** **void** checkNum(**int** num) {

**if** (num < 1) {

**throw** **new** ArithmeticException("\nNumber is negative, cannot calculate square");

        }

**else** {

            System.out.println("Square of " + num + " is " + (num\*num));

        }

    }

    //main method

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

            TestThrow obj = **new** TestThrow();

            obj.checkNum(-3);

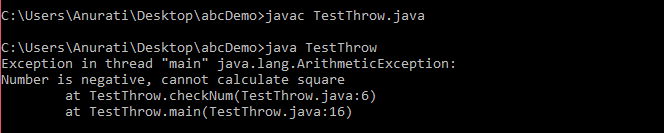
            System.out.println("Rest of the code..");

    }

}

**Output:**

Play Video



## Java throws Example

**TestThrows.java**

**public** **class** TestThrows {

    //defining a method

**public** **static** **int** divideNum(**int** m, **int** n) **throws** ArithmeticException {

**int** div = m / n;

**return** div;

    }

    //main method

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

        TestThrows obj = **new** TestThrows();

**try** {

            System.out.println(obj.divideNum(45, 0));

        }

**catch** (ArithmeticException e){

            System.out.println("\nNumber cannot be divided by 0");

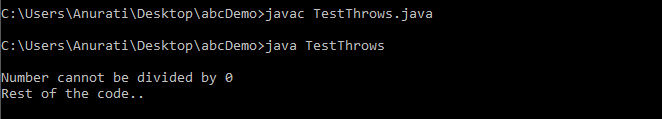
        }

        System.out.println("Rest of the code..");

    }

}

**Output:**



## Java throw and throws Example

**TestThrowAndThrows.java**

**public** **class** TestThrowAndThrows

{

    // defining a user-defined method

    // which throws ArithmeticException

**static** **void** method() **throws** ArithmeticException

    {

        System.out.println("Inside the method()");

**throw** **new** ArithmeticException("throwing ArithmeticException");

    }

    //main method

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

    {

**try**

        {

            method();

        }

**catch**(ArithmeticException e)

        {

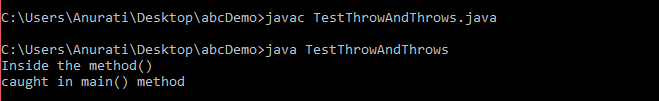
            System.out.println("caught in main() method");

        }

    }

}

**Output:**



Java finally block

**Java finally block** is a block used to execute important code such as closing the connection, etc.

Java finally block is always executed whether an exception is handled or not. Therefore, it contains all the necessary statements that need to be printed regardless of the exception occurs or not.

The finally block follows the try-catch block.

Flowchart of finally block



Note: If you don't handle the exception, before terminating the program, JVM executes finally block (if any).

Why use Java finally block?

* finally block in Java can be used to put "**cleanup**" code such as closing a file, closing connection, etc.
* The important statements to be printed can be placed in the finally block.

Usage of Java finally

Case 1: When an exception does not occur

Let's see the below example where the Java program does not throw any exception, and the finally block is executed after the try block.

**TestFinallyBlock.java**

**class** TestFinallyBlock {

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

**try**{

//below code do not throw any exception

**int** data=25/5;

   System.out.println(data);

  }

//catch won't be executed

**catch**(NullPointerException e){

System.out.println(e);

}

//executed regardless of exception occurred or not

**finally** {

System.out.println("finally block is always executed");

}

System.out.println("rest of phe code...");

  }

}

**Output:**



Case 2: When an exception occurr but not handled by the catch block

Let's see the the fillowing example. Here, the code throws an exception however the catch block cannot handle it. Despite this, the finally block is executed after the try block and then the program terminates abnormally.

**TestFinallyBlock1.java**

**public** **class** TestFinallyBlock1{

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

**try** {

        System.out.println("Inside the try block");

        //below code throws divide by zero exception

**int** data=25/0;

       System.out.println(data);

      }

      //cannot handle Arithmetic type exception

      //can only accept Null Pointer type exception

**catch**(NullPointerException e){

        System.out.println(e);

      }

      //executes regardless of exception occured or not

**finally** {

        System.out.println("finally block is always executed");

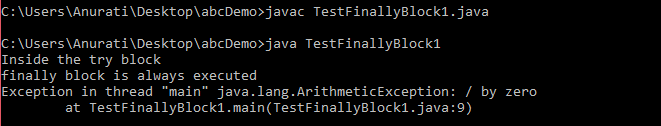
      }

      System.out.println("rest of the code...");

      }

    }

**Output:**



Case 3: When an exception occurs and is handled by the catch block

**Example:**

Let's see the following example where the Java code throws an exception and the catch block handles the exception. Later the finally block is executed after the try-catch block. Further, the rest of the code is also executed normally.

**TestFinallyBlock2.java**

**public** **class** TestFinallyBlock2{

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

**try** {

        System.out.println("Inside try block");

        //below code throws divide by zero exception

**int** data=25/0;

       System.out.println(data);

      }

      //handles the Arithmetic Exception / Divide by zero exception

**catch**(ArithmeticException e){

        System.out.println("Exception handled");

        System.out.println(e);

      }

      //executes regardless of exception occured or not

**finally** {

        System.out.println("finally block is always executed");

      }

      System.out.println("rest of the code...");

      }

    }

**Output:**

