**How to Take Input From User in Java?**

Java brings various Streams with its I/O package that helps the user to perform all the input-output operations. These streams support all the types of objects, data types, characters, files, etc to fully execute the I/O operations. There are **two ways** by which we can take input from the user or from a file

• BufferedReader Class

• Scanner Class

1. BufferedReader

It is a simple class that is used to read a sequence of characters. It has a simple function that reads a character another read which reads, an array of characters, and a readLine() function which reads a line.

InputStreamReader() is a function that converts the input stream of bytes into a stream of characters so that it can be read as BufferedReader expects a stream of characters.

BufferedReader can throw checked Exceptions

// Java Program for taking user

// input using BufferedReader Class

**import** java.io.\*;

**class** brc {

// Main Method

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

**throws** IOException

{

// Creating BufferedReader Object

// InputStreamReader converts bytes to

// stream of character

BufferedReader bfn = **new** BufferedReader(

**new** InputStreamReader(System.in));

// String reading internally

String str = bfn.readLine();

// Integer reading internally

**int** it = Integer.parseInt(bfn.readLine());

// Printing String

System.out.println("Entered String : " + str);

// Printing Integer

System.out.println("Entered Integer : " + it); }

}

**Miraj Trivedi**

**222**

**Entered String : Miraj Trivedi**

**Entered Integer : 222**

2. Scanner

It is an advanced version of BufferedReader which was added in later versions of Java. The scanner can read formatted input. It has different functions for different types of data types.

• The scanner is much easier to read as we don’t have to write throws as there is no exception thrown by it.

• It was added in later versions of Java

• It contains predefined functions to read an Integer, Character, and other data types as well.

**Syntax:** Scanner

Scanner scn = new Scanner(System.in);

**Syntax: Importing Scanner Class:** To use the Scanner we need to import the Scanner Class

import java.util.Scanner;

**Inbuilt Scanner functions are as follows:**

• Integer: nextInt()

• Float: nextFloat()

• String : next() and nextLine()

Hence, in the case of Integer and String in Scanner, we don’t require parsing as we did require in BufferedReader.

// Java Program to show how to take

// input from user using Scanner Class

**import** java.util.\*;

**class** GFG {

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

{

// Scanner definition

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

// input is a string ( one word )

// read by next() function

String str1 = scn.next();

// print String

System.out.println("Entered String str1 : " + str1);

// input is a String ( complete Sentence )

// read by nextLine()function

String str2 = scn.nextLine();

// print string

System.out.println("Entered String str2 : " + str2);

// input is an Integer

// read by nextInt() function

**int** x = scn.nextInt();

// print integer

System.out.println("Entered Integer : " + x);

// input is a floatingValue

// read by nextFloat() function

**float** f = scn.nextFloat();

// print floating value

System.out.println("Entered FloatValue : " + f); }

}

**Output :**

Entered String str1 : Miraj Trivedi

Entered String str2 : Mirav rathi

Entered Integer : 123

Entered FloatValue : 123.004

Differences Between BufferedReader and Scanner

• BufferedReader is a very basic way to read the input generally used to read the stream of characters. It gives an edge over Scanner as it is faster than Scanner because Scanner does lots of post-processing for parsing the input; as seen in nextInt(), nextFloat()

• BufferedReader is more flexible as we can specify the size of stream input to be read. (In general, it is there that BufferedReader reads larger input than Scanner)

• These two factors come into play when we are reading larger input. In general, the Scanner Class serves the input.

• BufferedReader is preferred as it is synchronized. While dealing with multiple threads it is preferred.

• For decent input, easy readability. The Scanner is preferred over BufferedReader.

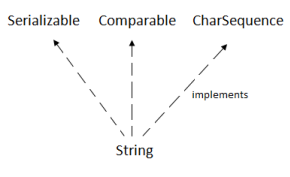
| **No. StringBuffer StringBuilder** | | |
| --- | --- | --- |
| 1) | StringBuffer is *synchronized* i.e. thread safe. It means two threads can't call the methods of StringBuffer simultaneously. | StringBuilder is *non-synchronized* i.e. not thread safe. It means two threads can call the methods of StringBuilder simultaneously. |
| 2) | StringBuffer is *less efficient* than StringBuilder. | StringBuilder is *more efficient* than StringBuffer. |
| 3) | StringBuffer was introduced in Java 1.0 | StringBuilder was introduced in Java 1.5 |

Java String

**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.

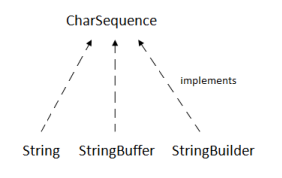


Object -> stream(serializable)

Stream -> object(deserializable)

**CharSequence Interface**

The CharSequence interface is used to represent the sequence of characters. String, StringBuffer and StringBuilder 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.

• String objects are immutable whereas StringBuffer and StringBuilder objects are mutable.

• StringBuffer and StringBuilder are same only difference is that StringBuilder is not synchronized whereas StringBuffer is. As StringBuilder is not synchronized, it is not thread safe but faster than StringBuffer.

• **Use String**, for a string which will be constant through out the application.

**Use StringBuffer**, for a string which can change in multi threaded applications.

**Use StringBuilder**, for a string which can change in single threaded applications.

**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

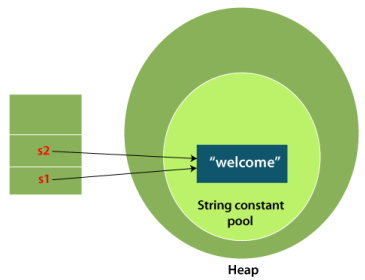
**1) String Literal**

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.

**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 varia ble

In such case, JVM 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).

1. **public class** StringExample{

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

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

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

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

6. String s3=**new** String("example");//creating Java string by new keyword 7. System.out.println(s1);

8. System.out.println(s2);

9. System.out.println(s3);

10. }}

**Output:**

java

strings

example

Immutable String in Java

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

A String is an unavoidable type of variable while writing any application program. String references are used to store various attributes like username, password, etc. In

Java, **String objects are immutable**. Immutable simply means unmodifiable or unchangeable.

Once String object is created its data or state can't be changed but a new String object is created.

1. **class** Testimmutablestring{

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

3. String s="Sachin";

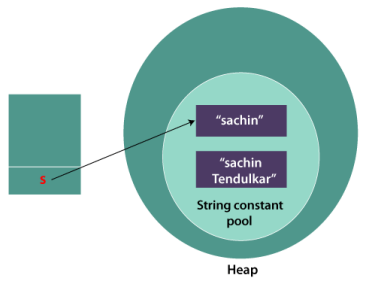
4. s.concat(" Tendulkar");//concat() method appends the string at the end 5. System.out.println(s);//will print Sachin because strings are immutable object s

6. }

7. }

**Output:**

Sachin



two objects are created but ***s*** reference variable still refers to "Sachin" not to "Sachin Tendulkar".

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

1. **class** Testimmutablestring1{

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

3. String s="Sachin";

4. s=s.concat(" Tendulkar");

5. System.out.println(s);

6. }

7. }

**Output:**

Sachin Tendulkar

**Why String objects are immutable in Java?**

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

Following are some features of String which makes String objects immutable. **1. ClassLoader:**

A ClassLoader in Java uses a String object as an argument. Consider, if the String object is modifiable, the value might be changed and the class that is supposed to be loaded might be different.

To avoid this kind of misinterpretation, String is immutable.

**2. Thread Safe:**

As the String object is immutable we don't have to take care of the synchronization that is required while sharing an object across multiple threads.

**3. Security:**

As we have seen in class loading, immutable String objects avoid further errors by loading the correct class. This leads to making the application program more secure. Consider an example of banking software. The username and password cannot be modified by any intruder because String objects are immutable. This can make the application program more secure.

**4. Heap Space:**

The immutability of String helps to minimize the usage in the heap memory. When we try to declare a new String object, the JVM checks whether the value already exists in the String pool or not. If it exists, the same value is assigned to the new object. This feature allows Java to use the heap space efficiently.

Java String compare

We can compare String in Java on the basis of content and reference.

It is used in **authentication** (by equals() method), **sorting** (by compareTo() method), **reference matching** (by == operator) etc.

There are three ways to compare String in Java:

1. By Using equals() Method

2. By Using == Operator

3. By compareTo() Method

**1) By Using equals() Method**

The String class equals() method compares the original content of the string. It compares values of string for equality. String class provides the following two methods:

o **public boolean equals(Object another)** compares this string to the specified object.

o **public boolean equalsIgnoreCase(String another)** compares this string to another string, ignoring case.

**Teststringcomparison1.java**

1. **class** Teststringcomparison1{

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

3. String s1="Sachin";

4. String s2="Sachin";

5. String s3=**new** String("Sachin");

6. String s4="Saurav";

7. System.out.println(s1.equals(s2));//true

8. System.out.println(s1.equals(s3));//true

9. System.out.println(s1.equals(s4));//false

10. }

11. }

**Output:**

true

true

false

In the above code, two strings are compared using **equals()** method of **String** class. And the result is printed as boolean values, **true** or **false**.

**Teststringcomparison2.java**

1. **class** Teststringcomparison2{

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

3. String s1="Sachin";

4. String s2="SACHIN";

5.

6. System.out.println(s1.equals(s2));//false

7. System.out.println(s1.equalsIgnoreCase(s2));//true

8. }

9. }

**Output:**

false

true

**2) By Using == operator**

The == operator compares references not values.

1. **class** Teststringcomparison3{

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

3. String s1="Sachin";

4. String s2="Sachin";

5. String s3=**new** String("Sachin");

6. System.out.println(s1==s2);//true (because both refer to same instance) 7. System.out.println(s1==s3);//false(because s3 refers to instance created in n onpool)

8. }

9. }

**Output:**

true

false

**3) By Using compareTo() method**

The String class compareTo() method compares values lexicographically and returns an integer value that describes if first string is less than, equal to or greater than second string.

Suppose s1 and s2 are two String objects. If:

o **s1 == s2** : The method returns 0.

o **s1 > s2** : The method returns a positive value.

o **s1 < s2** : The method returns a negative value.

1. **class** Teststringcomparison4{

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

3. String s1="Sachin";

4. String s2="Sachin";

5. String s3="Ratan";

6. System.out.println(s1.compareTo(s2));//0

7. System.out.println(s1.compareTo(s3));//1(because s1>s3)

8. System.out.println(s3.compareTo(s1));//-1(because s3 < s1 ) 9. }

10. }

**Output:**

0

1

-1

**Java String charAt() Method Examples** 1. **public class** CharAtExample{

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

3. String name="Rotaract";

4. **char** ch=name.charAt(3);//returns the char value at the 4th index

5. System.out.println(ch);

6. }}

**Output:**

t

Java String concat

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

1. **public class** ConcatExample{

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

3. String s1="java string";

4. // The string s1 does not get changed, even though it is invoking the method

5. // concat(), as it is immutable. Therefore, the explicit assignment is required he re.

6. s1.concat("is immutable");

7. System.out.println(s1);

8. s1=s1.concat(" is immutable so assign it explicitly");

9. System.out.println(s1);

10. }}

**Output:**

java string

java string is immutable so assign it explicitly

1. **public class** ConcatExample2 {

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

3. String str1 = "Hello";

4. String str2 = "world";

5. String str3 = "good morning";

6. // Concatenating one string

7. String str4 = str1.concat(str2);

8. System.out.println(str4);

9. // Concatenating multiple strings

10. String str5 = str1.concat(str2).concat(str3);

11. System.out.println(str5);

12. }

13. }

**Output:**

Helloworld

Helloworldgoodmorning

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.

1. **public class** EqualsExample{

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

3. String s1="hello";

4. String s2="hello";

5. String s3="HELLO";

6. String s4="java";

7. System.out.println(s1.equals(s2));//true because content and case is same 8. System.out.println(s1.equals(s3));//false because case is not same 9. System.out.println(s1.equals(s4));//false because content is not same 10. }}

**Output:**

true

false

false

Java String indexOf()

The **Java String class indexOf()** method returns the position of the first occurrence of the specified character or string in a specified string.

There are four overloaded indexOf() method in Java. The signature of indexOf() methods are given below:

| **No. Method Description** | | |
| --- | --- | --- |
| 1 | int indexOf(int ch) | It returns the index position for the given char value |
| 2 | int indexOf(int ch, int fromIndex) | It returns the index position for the given char value and from index |
| 3 | int indexOf(String substring) | It returns the index position for the given substring |
| 4 | int indexOf(String substring, int fromIndex) | It returns the index position for the given substring and from index |

1. **public class** IndexOfExample{

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

3. String s1="this is index of example";

4. //passing substring

5. **int** index1=s1.indexOf("is");//returns the index of is substring

6. **int** index2=s1.indexOf("index");//returns the index of index substring 7. System.out.println(index1+" "+index2);//2 8

8.

9. //passing substring with from index

10. **int** index3=s1.indexOf("is",4);//returns the index of is substring after 4th index

11. System.out.println(index3);//5 i.e. the index of another is

12.

13. //passing char value

14. **int** index4=s1.indexOf('s');//returns the index of s char value

15. System.out.println(index4);//3

16. }}

**Output:**

2 8

5

3

Java String isEmpty()

The **Java String class isEmpty()** method checks if the input string is empty or not. Note that here empty means the number of characters contained in a string is zero.

1. **public class** IsEmptyExample{

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

3. String s1="";

4. String s2="helloworld";

5.

6. System.out.println(s1.isEmpty());

7. System.out.println(s2.isEmpty());

8. }}

**Output:**

true

false

Java String 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.

1. **public class** StringJoinExample2 {

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

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

4. System.out.print(date);

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

6. System.out.println(" "+time);

7. }

8. }

**Output:**

25/06/2018 12:10:10

Java String lastIndexOf()

The **Java String class lastIndexOf()** method returns the last index of the given character value or substring. If it is not found, it returns -1. The index counter starts from zero.

| **No. Method Description** | | |
| --- | --- | --- |
| 1 | int lastIndexOf(int ch) | It returns last index position for the given char value |
| 2 | int lastIndexOf(int ch, int fromIndex) | It returns last index position for the given char value and from index |
| 3 | int lastIndexOf(String substring) | It returns last index position for the given substring |
| 4 | int lastIndexOf(String substring, int fromIndex) | It returns last index position for the given substring and from index |

1. **public class** LastIndexOfExample{

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

3. String s1="this is index of example";//there are 2 's' characters in this sentence

4. **int** index1=s1.lastIndexOf('s');//returns last index of 's' char value 5. System.out.println(index1);//6

6. }}

**Output:**

6

Java String 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.

1. **public class** LengthExample{

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

3. String s1="helloworld";

4. String s2="morning";

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

6. System.out.println("string length is: "+s2.length());//7 is the length of python st ring

7. }}

**Output:**

string length is: 10

string length is: 7

Java String split()

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

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. }}

Output

java

string

split

method

by

javatpoint

Java String 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.

1. **public** String substring(**int** startIndex) // type - 1

2. and

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

**Exception Throws**

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

o if the start index is negative value

o end index is lower than starting index.

o Either starting or ending index is greater than the total number of characters present in the string.

o **public class** SubstringExample{

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

o String s1="javastrings";

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

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

o }}

**Output:**

va

vastrings

1. **public class** SubstringExample2 {

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

3. String s1=" javastrings ";

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:**

javastrings

tring

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

Java String 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.

1. **public class** StringTrimExample{

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

3. String s1=" hello string ";

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

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

6. }}

**Output**

hello string goodmorning

hello stringgoodmorning

| **No. StringBuffer StringBuilder** | | |
| --- | --- | --- |
| 1) | StringBuffer is *synchronized* i.e. thread safe. It means two threads can't call the methods of StringBuffer simultaneously. | StringBuilder is *non-synchronized* i.e. not thread safe. It means two threads can call the methods of StringBuilder simultaneously. |
| 2) | StringBuffer is *less efficient* than StringBuilder. | StringBuilder is *more efficient* than StringBuffer. |

| 3) | StringBuffer was introduced in Java 1.0 | StringBuilder was introduced in Java 1.5 |
| --- | --- | --- |

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.

**Important Constructors of StringBuffer Class**

| **Constructor Description** | |
| --- | --- |
| StringBuffer() | It creates an empty String buffer with the initial capacity of 16 characters. |
| 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. |

| **Modifier and**  **Method Description**  **Type** | | |
| --- | --- | --- |
| public  synchronized  StringBuffer | append(String s) | It is used to append the specified string with this string. The append() method is overloaded like append(char), append(boolean), append(int), append(float), append(double) etc. |
| public  synchronized  StringBuffer | insert(int offset, String s) | It is used to insert the specified string with this string at the specified position. The insert() method is overloaded like insert(int, char), insert(int, boolean), insert(int, int), insert(int, float), insert(int, double) etc. |

| public  synchronized  StringBuffer | replace(int startIndex, int endIndex, String str) | It is used to replace the string from specified startIndex and endIndex. |
| --- | --- | --- |
| public  synchronized  StringBuffer | delete(int startIndex, int endIndex) | It is used to delete the string from specified startIndex and endIndex. |
| public  synchronized  StringBuffer | reverse() | is used to reverse the string. |
| public int | capacity() | It is used to return the current capacity. |
| public void | ensureCapacity(int  minimumCapacity) | It is used to ensure the capacity at least equal to the given minimum. |
| public char | charAt(int index) | It is used to return the character at the specified position. |
| public int | length() | It is used to return the length of the string i.e. total number of characters. |
| public String | substring(int beginIndex) | It is used to return the substring from the specified beginIndex. |
| public String | substring(int beginIndex, int endIndex) | It is used to return the substring from the specified beginIndex and endIndex. |

**StringBuffer Methods**

**1) StringBuffer Class append() Method**

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

1. **class** StringBufferExample{

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

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

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

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

6. }

7. }

**Output:**

Hello Java

**2) StringBuffer insert() Method**

The insert() method inserts the given String with this string at the given position. **StringBufferExample2.java**

1. **class** StringBufferExample2{

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

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

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

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

6. }

7. }

**Output:**

HJavaello

**3) StringBuffer replace() Method**

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

**StringBufferExample3.java**

1. **class** StringBufferExample3{

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

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

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

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

6. }

7. }

**Output:**

HJavalo

**4) StringBuffer delete() Method**

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

1. **class** StringBufferExample4{

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

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

4. sb.delete(1,3);

5. System.out.println(sb);//prints Hlo

6. }

7. }

**Output:**

Hlo

**5) StringBuffer reverse() Method**

The reverse() method of the StringBuilder class reverses the current String. **StringBufferExample5.java**

1. **class** StringBufferExample5{

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

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

4. sb.reverse();

5. System.out.println(sb);//prints olleH

6. }

7. }

**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**

1. **class** StringBufferExample6{

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

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

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

5. sb.append("Hello");

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

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

8. System.out.println(sb.capacity());//now (16\*2)+2=34 i.e (oldcapacity\*2)+2 9. }

10. }

**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**

1. **class** StringBufferExample7{

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

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

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

5. sb.append("Hello");

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

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

8. System.out.println(sb.capacity());//now (16\*2)+2=34 i.e (oldcapacity\*2)+2 9. sb.ensureCapacity(10);//now no change

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

11. sb.ensureCapacity(50);//now (34\*2)+2

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

13. }

14. }

**Output:**

16

16

34

34

70

**Introduction to Exceptions:-**

**Exception Handling** in Java is one of the effective means to handle the runtime errors so that the regular flow of the application can be preserved. Java Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException, IOException, SQLException, RemoteException, etc.

**Exception** is an unwanted or unexpected event, which occurs during the execution of a program, i.e. at run time, that disrupts the normal flow of the program’s instructions. Exceptions can be caught and handled by the program. When an exception occurs within a method, it creates an object. This object is called the exception object. It contains information about the exception, such as the name and description of the exception and the state of the program when the exception occurred.

**Major reasons why an exception Occurs**

• Invalid user input

• Device failure

• Loss of network connection

• Physical limitations (out of disk memory)

• Code errors

• Opening an unavailable file

**Errors** represent irrecoverable conditions such as Java virtual machine (JVM) running out of memory, memory leaks, stack overflow errors, library incompatibility, infinite recursion, etc. Errors are usually beyond the control of the programmer, and we should not try to handle errors.

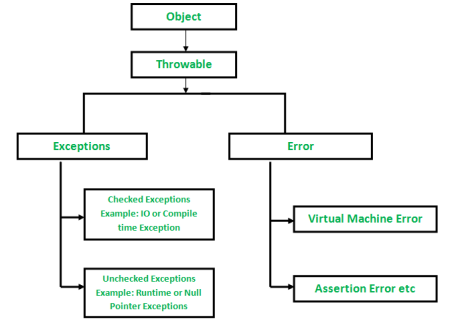
**differences between Error and Exception**

• **Error:** An Error indicates a serious problem that a reasonable application should not try to catch.

• **Exception:** Exception indicates conditions that a reasonable application might try to catch.

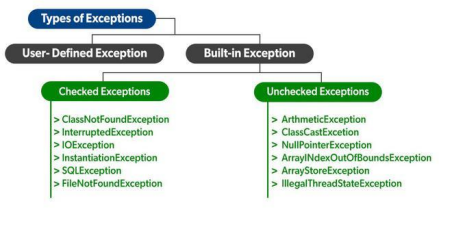
Exception Hierarchy

All exception and error types are subclasses of class **Throwable**, which is the base class of the hierarchy. One branch is headed by **Exception**. This class is used for exceptional conditions that user programs should catch. NullPointerException is an example of such an exception. Another branch, **Error** is used by the Java run-time system(JVM) to indicate errors having to do with the run-time environment itself(JRE). StackOverflowError is an example of such an error.



**Types of Exceptions**

Java defines several types of exceptions that relate to its various class libraries. Java also allows users to define their own exceptions.



**Exceptions can be categorized in two ways:**

1. **Built-in Exceptions**

• Checked Exception

• Unchecked Exception

2. **User-Defined Exceptions**

**A. Built-in Exceptions:**

Built-in exceptions are the exceptions that are available in Java libraries. These exceptions are suitable to explain certain error situations.

• **Checked Exceptions:** Checked exceptions are called compile-time exceptions because these exceptions are checked at compile-time by the compiler.

• **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.

**B. User-Defined Exceptions:**

Sometimes, the built-in exceptions in Java are not able to describe a certain situation. In such cases, users can also create exceptions, which are called ‘user-defined Exceptions’.

The ***advantages of Exception Handling in Java*** are as follows: 1. Provision to Complete Program Execution

2. Easy Identification of Program Code and Error-Handling Code 3. Propagation of Errors

4. Meaningful Error Reporting

5. Identifying Error Types

**Methods to print the Exception information:**

**1.printStackTrace()**– This method prints exception information in the format of Name of the exception: description of the exception, stack trace.

//program to print the exception information using printStackTrace() method **import** java.io.\*;

**class** GFG {

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

**int** a=5;

**int** b=0;

**try**{

System.out.println(a/b);

}

**catch**(ArithmeticException e){

e.printStackTrace();

}

}

}

**Output:**

java.lang.ArithmeticException: / by zero

at GFG.main(File.java:10)

**2.toString() – This** method prints exception information in the format of Name of the exception: description of the exception.

//program to print the exception information using toString() method **import** java.io.\*;

**class** GFG1 {

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

**int** a=5;

**int** b=0;

**try**{

System.out.println(a/b);

}

**catch**(ArithmeticException e){

System.out.println(e.toString());

}

}

}

**Output:**

java.lang.ArithmeticException: / by zero

**3.getMessage()** -This method prints only the description of the exception. //program to print the exception information using getMessage() method

**import** java.io.\*;

**class** GFG1 {

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

**int** a=5;

**int** b=0;

**try**{

System.out.println(a/b);

}

**catch**(ArithmeticException e){

System.out.println(e.getMessage());

}

}

}

**Output:**

/ by zero

How Does JVM handle an Exception?

**Default Exception Handling:** Whenever inside a method, if an exception has occurred, the method creates an Object known as an Exception Object and hands it off to the run-time system(JVM). The exception object contains the name and description of the exception and the current state of the program where the exception has occurred. Creating the Exception Object and handling it in the run-time system is called throwing an Exception. There might be a list of the methods that had been called to get to the method where an exception occurred. This ordered list of the methods is called **Call Stack**. Now the following procedure will happen.

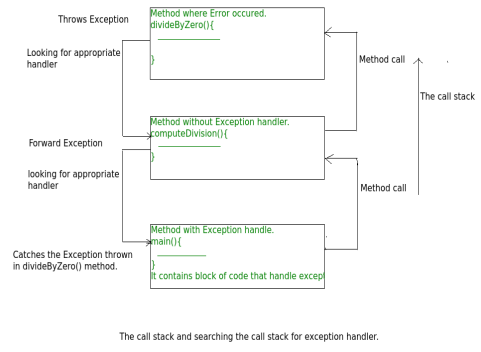
• The run-time system searches the call stack to find the method that contains a block of code that can handle the occurred exception. The block of the code is called an **Exception handler**.

• The run-time system starts searching from the method in which the exception occurred, and proceeds through the call stack in the reverse order in which methods were called.

• If it finds an appropriate handler, then it passes the occurred exception to it. An appropriate handler means the type of the exception object thrown matches the type of the exception object it can handle.

• If the run-time system searches all the methods on the call stack and couldn’t have found the appropriate handler, then the run-time system handover the Exception Object to the **default exception handler**, which is part of the run-time system. This handler prints the

exception information in the following format and terminates the program **abnormally**.



// Java Program to Demonstrate How Exception Is Thrown

// Class

// ThrowsExecp

**class** GFG {

// Main driver method

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

{

// Taking an empty string

String str = **null**;

// Getting length of a string

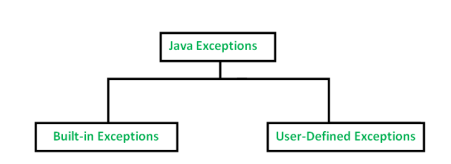
System.out.println(str.length());

}

}



**Types of Exception**

****Built-in Exception

**A. Arithmetic exception**

// Java program to demonstrate ArithmeticException

**class** ArithmeticException\_Demo

{

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

{

**try** {

**int** a = 30, b = 0;

**int** c = a/b; // cannot divide by zero

System.out.println ("Result = " + c);

}

**catch**(ArithmeticException e) {

System.out.println ("Can't divide a number by 0"); }

}

}

**Output**

Can't divide a number by 0

**B. NullPointer Exception**

//Java program to demonstrate NullPointerException

**class** NullPointer\_Demo

{

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

{

**try** {

String a = **null**; //null value

System.out.println(a.charAt(0));

} **catch**(NullPointerException e) {

System.out.println("NullPointerException.."); }

}

}

**Output**

NullPointerException..

**C. StringIndexOutOfBound Exception**

// Java program to demonstrate StringIndexOutOfBoundsException **class** StringIndexOutOfBound\_Demo

{

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

{

**try** {

String a = "This is like chipping "; // length is 22 **char** c = a.charAt(24); // accessing 25th element System.out.println(c);

}

**catch**(StringIndexOutOfBoundsException e) {

System.out.println("StringIndexOutOfBoundsException"); }

}

}

**Output**

StringIndexOutOfBoundsException

**D. FileNotFound Exception**

//Java program to demonstrate FileNotFoundException **import** java.io.File;

**import** java.io.FileNotFoundException;

**import** java.io.FileReader;

**class** File\_notFound\_Demo {

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

**try** {

// Following file does not exist

File file = **new** File("E://file.txt");

FileReader fr = **new** FileReader(file); } **catch** (FileNotFoundException e) {

System.out.println("File does not exist"); }

}

}

**Output:**

File does not exist

**E. NumberFormat Exception**

// Java program to demonstrate NumberFormatException **class** NumberFormat\_Demo

{

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

{

**try** {

// "akki" is not a number

**int** num = Integer.parseInt ("akki") ;

System.out.println(num);

} **catch**(NumberFormatException e) {

System.out.println("Number format exception"); }

}

}

**Output**

Number format exception

**F. ArrayIndexOutOfBounds Exception**

// Java program to demonstrate ArrayIndexOutOfBoundException **class** ArrayIndexOutOfBound\_Demo

{

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

{

**try**{

**int** a[] = **new int**[5];

a[6] = 9; // accessing 7th element in an array of // size 5

}

**catch**(ArrayIndexOutOfBoundsException e){

System.out.println ("Array Index is Out Of Bounds"); }

}

}

**Output**

Array Index is Out Of Bounds

**G. IO Exception**

// Java program to demonstrate IOException

**class** IOException\_Demo {

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

{

// Create a new scanner with the specified String // Object

Scanner scan = **new** Scanner("Hello Geek!");

// Print the line

System.out.println("" + scan.nextLine());

// Check if there is an IO exception

System.out.println("Exception Output: "

+ scan.ioException());

scan.close();

}

}

**Output:**

Hello Geek!

Exception Output: null

**H. NoSuchMethod Exception**

**I. IllegalArgumentException:** This program, checks whether the person is eligible for voting or not. If the age is greater than or equal to 18 then it will

not throw any error. If the age is less than 18 then it will throw an error with the error statement.

Also, we can specify “throw new IllegalArgumentException()” without the error message. We can also specify Integer.toString(variable\_name) inside the IllegalArgumentException() and It will print the argument name which is not satisfied the given condition.

/\*package whatever //do not write package name here \*/

**import** java.io.\*;

**class** GFG {

**public static void** print(**int** a)

{

**if**(a>=18){

System.out.println("Eligible for Voting");

}

**else**{

**throw new** IllegalArgumentException("Not Eligible for Voting");

}

}

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

GFG.print(14);

}

}

**Output :**

Exception in thread "main" java.lang.IllegalArgumentException: Not Eligible for Voting

at GFG.print(File.java:13)

at GFG.main(File.java:19)

**J. IllegalStateException:** This program, displays the addition of numbers only for Positive integers. If both the numbers are positive then only it will call the print method to print the result otherwise it will throw the IllegalStateException with an error statement. Here, the method is not accessible for non-positive integers.

Also, we can specify the “throw new IllegalStateException()” without the error statement.

/\*package whatever //do not write package name here \*/

**import** java.io.\*;

**class** GFG {

**public static void** print(**int** a,**int** b)

{

System.out.println("Addition of Positive Integers :"+(a+b)); }

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

**int** n1=7;

**int** n2=-3;

**if**(n1>=0 && n2>=0)

{

GFG.print(n1,n2);

}

**else**

{

**throw new** IllegalStateException("Either one or two numbers are not Positive Integer");

}

}

}

**Output :**

Exception in thread "main" java.lang.IllegalStateException: Either one or two numbers are not Positive Integer

at GFG.main(File.java:20)

**k. ClassNotFound Exception :**

// Java program to demonstrate ClassNotFoundException

**public class** ClassNotFoundException\_Demo

{

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

**try**{

Class.forName("Class1"); // Class1 is not defined }

**catch**(ClassNotFoundException e){

System.out.println(e);

System.out.println("Class Not Found...");

}

}

}

**Output**

java.lang.ClassNotFoundException: Class1

Class Not Found...

**User-Defined Exceptions**

Sometimes, the built-in exceptions in Java are not able to describe a certain situation. In such cases, the user can also create exceptions which are called ‘user-defined Exceptions’.

| class SampleException{ |
| --- |

| public static void main(String args[]){ |
| --- |

| try{ |
| --- |

| throw new UserException(400); |
| --- |

| } |
| --- |

| catch(UserException e){ |
| --- |

| System.out.println(e) ; |
| --- |

| } |
| --- |

| } |
| --- |

| } |
| --- |

| class UserException extends Exception{ |
| --- |

| int num1; |
| --- |

| UserException(int num2) { |
| --- |

| num1=num2; |
| --- |

| } |
| --- |

| public String toString(){ |
| --- |

| return ("Status code = "+num1) ; |
| --- |

| } |
| --- |

| } |
| --- |

**Output:**

Keyword ‘throw’ is used to create a new Exception and throw it to catch block.

Status Code = 400

**Checked Exceptions**

These are the exceptions that are checked at compile time. If some code within a method throws a checked exception, then the method must either handle the exception or it must specify the exception using the *throws* keyword.

In checked exception, there are two types: fully checked and partially checked exceptions.

A fully checked exception is a checked exception where all its child classes are also checked, like IOException, InterruptedException.

A partially checked exception is a checked exception where some of its child classes are unchecked, like Exception.

For example, consider the following Java program that opens the file at location “C:\test\a.txt” and prints the first three lines of it. The program doesn’t compile, because the function main() uses FileReader() and FileReader() throws a checked exception *FileNotFoundException*. It also uses readLine() and close() methods, and these methods also throw checked exception *IOException*

// Java Program to Illustrate Checked Exceptions

// Where FileNotFoundException occurred

// Importing I/O classes

**import** java.io.\*;

// Main class

**class** GFG {

// Main driver method

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

{

// Reading file from path in local directory

FileReader file = **new** FileReader("C:\\test\\a.txt");

// Creating object as one of ways of taking input BufferedReader fileInput = **new** BufferedReader(file);

// Printing first 3 lines of file "C:\test\a.txt" **for** (**int** counter = 0; counter < 3; counter++)

System.out.println(fileInput.readLine());

// Closing file connections

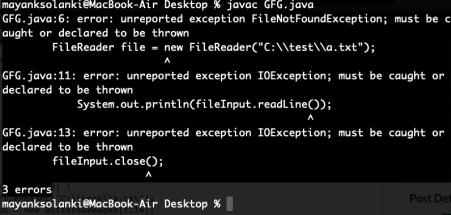
// using close() method

fileInput.close();

}

}

**Output:**

****

// Java Program to Illustrate Checked Exceptions

// Where FileNotFoundException does not occur

// Importing I/O classes

**import** java.io.\*;

// Main class

**class** GFG {

// Main driver method

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

**throws** IOException

{

// Creating a file and reading from local repository FileReader file = **new** FileReader("C:\\test\\a.txt");

// Reading content inside a file

BufferedReader fileInput = **new** BufferedReader(file);

// Printing first 3 lines of file "C:\test\a.txt" **for** (**int** counter = 0; counter < 3; counter++)

System.out.println(fileInput.readLine());

// Closing all file connections

// using close() method

// Good practice to avoid any memory leakage

fileInput.close();

}

}

**Output:**

First three lines of file "C:\test\a.txt"

**Unchecked Exceptions**

These are the exceptions that are not checked at compile time. In C++, all exceptions are unchecked, so it is not forced by the compiler to either handle or specify the exception. It is up to the programmers to be civilized, and specify or catch the exceptions.

In Java, exceptions under *Error* and *RuntimeException* classes are unchecked exceptions, everything else under throwable is checked.

Consider the following Java program. It compiles fine, but it

throws *ArithmeticException* when run. The compiler allows it to compile because *ArithmeticException* is an unchecked exception.

// Java Program to Illustrate Un-checked Exceptions

// Main class

**class** GFG {

// Main driver method

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

{

// Here we are dividing by 0

// which will not be caught at compile time

// as there is no mistake but caught at runtime

// because it is mathematically incorrect

**int** x = 0;

**int** y = 10;

**int** z = y / x;

}

}

**Output:**

Exception in thread "main" java.lang.ArithmeticException: / by zero at Main.main(Main.java:5)

Java Result: 1

In short unchecked exceptions are runtime exceptions that are not required to be caught or declared in a throws clause. These exceptions are usually

caused by programming errors, such as attempting to access an index out of bounds in an array or attempting to divide by zero.

Unchecked exceptions include all subclasses of the RuntimeException class, as well as the Error class and its subclasses.

Here are some examples of unchecked exceptions in Java:

**throw**: The throw keyword is used to transfer control from the try block to the catch block.

**throws**: The throws keyword is used for exception handling without try & catch block. It specifies the exceptions that a method can throw to the caller and does not handle itself.

**finally**: It is executed after the catch block. We use it to put some common code (to be executed irrespective of whether an exception has occurred or not ) when there are multiple catch blocks.

**throw**

The throw keyword in Java is used to explicitly throw an exception from a method or any block of code. We can throw either checked or unchecked exception. The throw keyword is mainly used to throw custom exceptions.

// Java program to demonstrate working of throws

**class** ThrowsExecp

{

**static void** fun() **throws** IllegalAccessException

{

System.out.println("Inside fun(). ");

**throw new** IllegalAccessException("demo");

}

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

{

**try**

{

fun();

}

**catch**(IllegalAccessException e)

{

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

}

}

}

**throws**

throws is a keyword in Java which is used in the signature of method to indicate that this method might throw one of the listed type exceptions. The caller to these methods has to handle the exception using a try-catch block.

In a program, if there is a chance of raising an exception then compiler always warn us about it and compulsorily we should handle that checked exception, Otherwise we will get compile time error saying **unreported exception XXX must be caught or declared to be thrown**. To prevent this compile time error we can handle the exception in two ways: 1. By using try catch

2. By using **throws** keyword

// Java program to demonstrate working of throws

**class** ThrowsExecp

{

**static void** fun() **throws** IllegalAccessException

{

System.out.println("Inside fun(). ");

**throw new** IllegalAccessException("demo");

}

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

{

**try**

{

fun();

}

**catch**(IllegalAccessException e)

{

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

}

}

}

**Output:**

Inside fun().

caught in main.

**Important points to remember about throws keyword:** • throws keyword is required only for checked exception and usage of throws keyword for unchecked exception is meaningless.

• throws keyword is required only to convince compiler and usage of throws keyword does not prevent abnormal termination of program. • By the help of throws keyword we can provide information to the caller of the method about the exception.

| S.  No. | Key  Difference | throw | throws |
| --- | --- | --- | --- |
| 1. | Point of  Usage | The throw keyword is used inside a  function. It is used w  hen it is required to throw an Exception logically. | The throws keyword is used in the function signature. It is used  when the function has some statements that can lead to exceptions. |
| 2. | Exceptions Thrown | The throw keyword is used to throw an  exception explicitly. It can throw only one exception at a time. | The throws keyword can be used to declare multiple exceptions, separated by a comma. Whichever exception occurs, if matched with the declared ones, is thrown automatically then. |
| 3. | Syntax | Syntax  of throw keyword  includes the instance of the Exception to be thrown. Syntax wise throw keyword is  followed by the  instance variable. | Syntax  of throws keyword  includes the class  names of the  Exceptions to be  thrown. Syntax wise throws keyword is  followed by exception class names. |

| 4. | Propagation of  Exceptions | throw keyword cannot propagate checked exceptions. It is only used to propagate the unchecked  Exceptions that are not checked using the throws keyword. | throws keyword is used to propagate the  checked Exceptions only. |
| --- | --- | --- | --- |

**User-defined Exceptions:**

You can create your own exceptions in Java by creating your own exception sub class simply by extending java Exception class.

You can define a constructor for your Exception sub class (not compulsory) and you can override the toString() function to display your customized message on catch.

User defined exceptions in java are also known as Custom exceptions. All exceptions must be a child of Throwable.

If you want to write a checked exception that is automatically enforced by the Handle or Declare Rule, you need to extend the Exception class.

If you want to write a runtime exception, you need to extend the RuntimeException class. Program1:

class MyException extends Exception

{

String str1;

MyException(String str2)

{ str1=str2; }

public String toString()

{ return ("Output String = "+str1) ; }

}

class CustomException{

public static void main(String args[]){

try{ throw new MyException("Custom"); // I'm throwing user defined custom exception above } catch(MyException exp)

{ System.out.println("Hi this is my catch block") ; System.out.println(exp) ; } } }

Output: Hi this is my catch block

Output String = Custom