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.

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.

**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

|  |  |
| --- | --- |
| **Keyword** | **Description** |
| try | The try block contains set of statements where an exception can occur. |
| catch | Catch block is used to handle the uncertain condition of try block. A try block is always followed by a catch block, which handles the exception that occurs in associated try block. |
| finally | It is executed after catch block. We basically use it to put some common code when there are multiple catch blocks. |
| throw | The "throw" keyword is used to throw an exception. Throw keyword is used to transfer control from try block to catch block. |
| throws | 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. How to use try-catch clause? try {  // block of code to monitor for errors  // the code you think can raise an exception  }  catch (ExceptionType1 exOb) {  // exception handler for ExceptionType1  }  catch (ExceptionType2 exOb) {  // exception handler for ExceptionType2  }  // optional  finally {  // block of code to be executed after try block ends  }   1. public class JavaExceptionExample{ 2. public static void main(String args[]){ 3. try{ 4. //code that may raise exception 5. int data=100/0; 6. }catch(ArithmeticException e){System.out.println(e);} 7. //rest code of the program 8. System.out.println("rest of the code..."); 9. } 10. }   **Output:**  Exception in thread main java.lang.ArithmeticException:/ by zero  rest of the code...   String s=null;   System.out.println(s.length());//NullPointerException   1. String s="abc"; 2. int i=Integer.parseInt(s);//NumberFormatException 3. int a[]=new int[5]; 4. a[10]=50; //ArrayIndexOutOfBoundsException |

class Division {

    public static void main(String[] args)

    {

        int a = 10, b = 5, c = 5, result;

        try {

            result = a / (b - c);

            System.out.println("result" + result);

        }

        catch (ArithmeticException e) {

            System.out.println("Exception caught:Division by zero");

        }

        finally {

            System.out.println("I am in final block");

        }

    }

}

**Output:**

Exception caught:Division by zero

I am in final block

**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](https://www.geeksforgeeks.org/checked-vs-unchecked-exceptions-in-java/). The throw keyword is mainly used to throw custom exceptions.

**Syntax:**

**throw *Instance***

Example:

**throw new ArithmeticException("/ by zero");**

But this exception i.e, *Instance* must be of type **Throwable** or a subclass of **Throwable**. For example Exception is a sub-class of Throwable and [user defined exceptions typically extend Exception class](https://www.geeksforgeeks.org/g-fact-32-user-defined-custom-exception-in-java/). Unlike C++, data types such as int, char, floats or non-throwable classes cannot be used as exceptions.

The flow of execution of the program stops immediately after the throw statement is executed and the nearest enclosing **try** block is checked to see if it has a **catch** statement that matches the type of exception. If it finds a match, controlled is transferred to that statement otherwise next enclosing **try** block is checked and so on. If no matching **catch** is found then the default exception handler will halt the program.

**An example of throw keyword:**

|  |
| --- |
| // Java program to demonstrate working of throws  class ThrowsExecp {       // This is a caller function      public static void main(String args[])      {          try {              int i=1;  int j=5;  int k=i+j;  if(k<10)  {  throw new ArithmeticException();  }          }          catch (NullPointerException e) {  System.out.println("NullEx called.");  }  catch (ArithmeticException e) {              System.out.println("ArithEx It is less than 10.");          }      }  } |

**Output:**

ArithEx It is less than 10.

|  |
| --- |
| **USING THROWS**  class ThrowsExecp2 {    public static int getdiv(int a,int b)  throws ArithmeticException  {  int res=a/b;  return res;  }  // This is a caller function  public static void main(String args[])  {  try {  int i=getdiv(6,2);    System.out.println(i);  }  catch (Exception e) {  System.out.println("NullEx  called.");  }    }  } |

**Output:**3

# Lifecycle and States of a Thread in Java

A [thread](https://www.geeksforgeeks.org/multithreading-in-java/) in Java at any point of time exists in any one of the following states. A thread lies only in one of the shown states at any instant:

1. New
2. Runnable
3. Blocked
4. Waiting
5. Timed Waiting
6. Terminated

#### Life Cycle of a thread

1. **New Thread:** When a new thread is created, it is in the new state. The thread has not yet started to run when the thread is in this state. When a thread lies in the new state, its code is yet to be run and hasn’t started to execute.
2. **Runnable State:** A thread that is ready to run is moved to a runnable state. In this state, a thread might actually be running or it might be ready to run at any instant of time. It is the responsibility of the thread scheduler to give the thread, time to run.   
   A multi-threaded program allocates a fixed amount of time to each individual thread. Each and every thread runs for a short while and then pauses and relinquishes the CPU to another thread so that other threads can get a chance to run. When this happens, all such threads that are ready to run, waiting for the CPU and the currently running thread lie in a runnable state.
3. **Blocked/Waiting state:** When a thread is temporarily inactive, then it’s in one of the following states:
   * Blocked
   * Waiting
4. **Timed Waiting:** A thread lies in a timed waiting state when it calls a method with a time-out parameter. A thread lies in this state until the timeout is completed or until a notification is received. For example, when a thread calls sleep or a conditional wait, it is moved to a timed waiting state.
5. **Terminated State:** A thread terminates because of either of the following reasons:
   * Because it exits normally. This happens when the code of the thread has been entirely executed by the program.
   * Because there occurred some unusual erroneous event, like segmentation fault or an unhandled exception.

|  |  |
| --- | --- |
| **Method** | **Meaning** |
| getName | Obtain thread’s name |
| getPriority | Obtain thread’s priority |
| isAlive | Determine if a thread is still running |
| Join | Wait for a thread to terminate |
| Run | Entry point for the thread |
| Sleep | Suspend a thread for a period of time |
| Start | Start a thread by calling its run method |

// ABC class implements the interface Runnable

class ABC implements Runnable

{

public void run()

{

// try-catch block

try

{

// moving thread t2 to the state timed waiting

Thread.sleep(100);

}

catch (InterruptedException ie)

{

ie.printStackTrace();

}

System.out.println("The state of thread t1 while it invoked the method join() on thread t2 -"+ ThreadState.t1.getState());

// try-catch block

try

{

Thread.sleep(200);

}

catch (InterruptedException ie)

{

ie.printStackTrace();

}

}

}

// ThreadState class implements the interface Runnable

public class ThreadState implements Runnable

{

public static Thread t1;

public static ThreadState obj;

// main method

public static void main(String argvs[])

{

// creating an object of the class ThreadState

obj = new ThreadState();

t1 = new Thread(obj);

// thread t1 is spawned

// The thread t1 is currently in the NEW state.

System.out.println("The state of thread t1 after spawning it - " + t1.getState());

// invoking the start() method on

// the thread t1

t1.start();

// thread t1 is moved to the Runnable state

System.out.println("The state of thread t1 after invoking the method start() on it - " + t1.getState());

}

public void run()

{

ABC myObj = new ABC();

Thread t2 = new Thread(myObj);

// thread t2 is created and is currently in the NEW state.

System.out.println("The state of thread t2 after spawning it - "+ t2.getState());

t2.start();

// thread t2 is moved to the runnable state

System.out.println("the state of thread t2 after calling the method start() on it - " + t2.getState());

// try-catch block for the smooth flow of the  program

try

{

// moving the thread t1 to the state timed waiting

Thread.sleep(200);

}

catch (InterruptedException ie)

{

ie.printStackTrace();

}

System.out.println("The state of thread t2 after invoking the method sleep() on it - "+ t2.getState() );

// try-catch block for the smooth flow of the  program

try

{

// waiting for thread t2 to complete its execution

t2.join();

}

catch (InterruptedException ie)

{

ie.printStackTrace();

}

System.out.println("The state of thread t2 when it has completed it's execution - " + t2.getState());

}

}

**Output:**

The state of thread t1 after spawning it - NEW

The state of thread t1 after invoking the method start() on it - RUNNABLE

The state of thread t2 after spawning it - NEW

the state of thread t2 after calling the method start() on it - RUNNABLE

The state of thread t1 while it invoked the method join() on thread t2 -TIMED\_WAITING

The state of thread t2 after invoking the method sleep() on it - TIMED\_WAITING

The state of thread t2 when it has completed it's execution – TERMINATED

# Multithreading in Java

Multithreading is a Java feature that allows concurrent execution of two or more parts of a program for maximum utilization of CPU. Each part of such program is called a thread. So, threads are light-weight processes within a process.

Threads can be created by using two mechanisms :

1. Extending the Thread class
2. Implementing the Runnable Interface

**Thread creation by extending the Thread class**  
We create a class that extends the **java.lang.Thread** class. This class overrides the run() method available in the Thread class. A thread begins its life inside run() method. We create an object of our new class and call start() method to start the execution of a thread. Start() invokes the run() method on the Thread object.

// Java code for thread creation by extending

// the Thread class

class MultithreadingDemo extends Thread {

public void run()

{

try {

// Displaying the thread that is running

System.out.println(

"Thread " + Thread.currentThread().getId()

+ " is running");

}

catch (Exception e) {

// Throwing an exception

System.out.println("Exception is caught");

}

}

}

// Main Class

public class Multithread {

public static void main(String[] args)

{

int n = 8; // Number of threads

for (int i = 0; i < n; i++) {

MultithreadingDemo object

= new MultithreadingDemo();

object.start();

}

}

}

**Output**

Thread 15 is running

Thread 14 is running

Thread 16 is running

Thread 12 is running

Thread 11 is running

Thread 13 is running

Thread 18 is running

Thread 17 is running

**Thread creation by implementing the Runnable Interface**  
We create a new class which implements java.lang.Runnable interface and override run() method. Then we instantiate a Thread object and call start() method on this object. 

|  |
| --- |
| // Java code for thread creation by implementing  // the Runnable Interface  class MultithreadingDemo implements Runnable {      public void run()      {          try {              // Displaying the thread that is running              System.out.println(                  "Thread " + Thread.currentThread().getId()                  + " is running");          }          catch (Exception e) {              // Throwing an exception              System.out.println("Exception is caught");          }      }  }    // Main Class  class Multithread {      public static void main(String[] args)      {          int n = 8; // Number of threads          for (int i = 0; i < n; i++) {              Thread object                  = new Thread(new MultithreadingDemo());              object.start();          }      }  } |

**Output**

Thread 13 is running

Thread 11 is running

Thread 12 is running

Thread 15 is running

Thread 14 is running

Thread 18 is running

Thread 17 is running

Thread 16 is running

|  |  |
| --- | --- |
| **User Thread** | **Daemon Thread** |
| JVM wait until user threads to finish their work. It never exit until all user threads finish their work. | The JVM will’t wait for daemon threads to finish their work. The JVM will exit as soon as all user threads finish their work. |
| JVM will not force to user threads for terminating, so JVM will wait for user threads to terminate themselves. | If all user threads have finished their work JVM will force the daemon threads to terminate |
| User threads are created by the application. | Mostly Daemon threads created by the JVM. |
| Mainly user threads are designed to do some specific task. | Daemon threads are design as to support the user threads. |
| User threads are foreground threads. | Daemon threads are background threads. |
| User threads are high priority threads. | Daemon threads are low priority threads. |
| Its life independent. | Its life depends on user threads. |

**Example: Check Thread is Daemon or not**

One can make a user thread as daemon thread by using setDaemon(boolean) method. In this example, thread type is being checked (User thread or Daemon thread) by using isDaemon() method. It returns true if it is daemon otherwise it returns false. for performing background tasks like Garbage collection and other housekeeping tasks.

// Java program check thread is Daemon or not

class MyThread extends Thread {

@Override

public void run()

{

System.out.println("User Thread or Non-Daemon Thread");

}

}

class MainThread {

public static void main(String[] args)

{

MyThread mt = new MyThread();

mt.start();

System.out.println("Main Thread");

System.out.println("Is " + mt.getName()

+ " a Daemon Thread: "

+ mt.isDaemon());

System.out.println("Is " + Thread.currentThread().getName()

+ " a Daemon Thread: "

+ Thread.currentThread().isDaemon());

}

}

**Output:**

Main Thread

Is Thread-0 a Daemon Thread: false

Is main a Daemon Thread: false

User Thread or Non-Daemon Thread

**Example: Make Non-Daemon Thread as a Daemon Thread:**  
In this example, a non-daemon thread is made a daemon using setDeamon(boolean).

// Java program make user thread as a daemon thread

class MyThread extends Thread {

@Override

public void run()

{

System.out.println("Non-Daemon thread");

}

}

class MainThread {

public static void main(String[] args)

{

MyThread mt = new MyThread();

System.out.println("Before using setDaemon() method: ");

System.out.println("Is " + mt.getName()

+ " a Daemon Thread: "

+ mt.isDaemon());

mt.setDaemon(true);

System.out.println("After using setDaemon() method: ");

System.out.println("Is " + mt.getName()

+ " a Daemon Thread: "

+ mt.isDaemon());

}

}

**Output:**

Before using setDaemon() method:

Is Thread-0 a Daemon Thread: false

After using setDaemon() method:

Is Thread-0 a Daemon Thread: true

**Stream and its types**

Java provides I/O Streams to read and write data where, a Stream represents an input source or an output destination which could be a file, i/o device, other program etc.

In general, a Stream will be an input stream or, an output stream.

* **InputStream** − This is used to read data from a source.
* **OutputStream** − This is used to write data to a destination.

Based on the data they handle there are two types of streams −

* **Byte Streams** − These handle data in bytes (8 bits) i.e., the byte stream classes read/write data of 8 bits. Using these you can store characters, videos, audios, images etc.
* **Character Streams** − These handle data in 16 bit Unicode. Using these you can read and write text data only.

Following diagram illustrates all the input and output Streams (classes) in Java.

## Standard Streams

In addition to above mentioned classes Java provides 3 standard streams representing the input and, output devices.

* **Standard Input** − This is used to read data from user through input devices. keyboard is used as standard input stream and represented as System.in.
* **Standard Output** − This is used to project data (results) to the user through output devices. A computer screen is used for standard output stream and represented as System.out.
* **Standard Error** − This is used to output the error data produced by the user's program and usually a computer screen is used for standard error stream and represented as System.err.

#### Byte stream classes

|  |  |
| --- | --- |
| **Stream class** | **Description** |
| **BufferedInputStream** | Used for Buffered Input Stream. |
| **BufferedOutputStream** | Used for Buffered Output Stream. |
|  |  |
|  |  |
| **FileInputStream** | Input stream that reads from a file |
| **FileOutputStream** | Output stream that write to a file. |
| **InputStream** | Abstract class that describe stream input. |
| **OutputStream** | Abstract class that describe stream output. |
| **PrintStream** | Output Stream that contain print() and println() method |

These classes define several key methods. Two most important are

1. read() : reads byte of data.
2. write() : Writes byte of data.

## Java Character Stream Classes

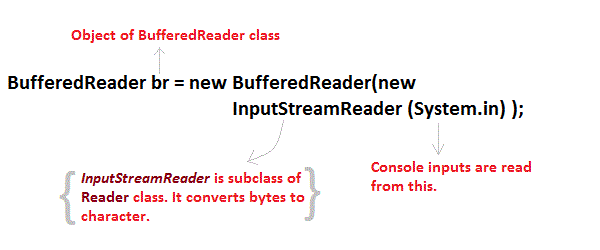
Character stream is also defined by using two abstract class at the top of hierarchy, they are Reader and Writer.

**Charcter stream classes**

|  |  |
| --- | --- |
| **Stream class** | **Description** |
| **BufferedReader** | Handles buffered input stream. |
| **BufferedWriter** | Handles buffered output stream. |
| **FileReader** | Input stream that reads from file. |
| **FileWriter** | Output stream that writes to file. |
| **InputStreamReader** | Input stream that translate byte to character |
| **OutputStreamReader** | Output stream that translate character to byte. |
| **PrintWriter** | Output Stream that contain print() and println() method. |
| **Reader** | Abstract class that define character stream input |
| **Writer** | Abstract class that define character stream output |

**Reading Console Input**

We use the object of BufferedReader class to take inputs from the keyboard



#### Reading Characters

read() method is used with BufferedReader object to read characters. As this function returns integer type value has we need to use typecasting to convert it into **char** type.

**Below is a simple example explaining character input.**

## import java.io.\*;

## class CharRead

## {

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

## {

## try{

## BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

## char c = (char)br.read(); //Reading character

## System.out.println(c);

## }

## catch(Exception e)

## {

## }

## }

## }

**String input from Keyboard in Java**

import java.io.\*;

class MyInput

{

public static void main(String[] args)

{

try{

String text;

InputStreamReader isr = new InputStreamReader(System.in);

BufferedReader br = new BufferedReader(isr);

text = br.readLine(); //Reading String

System.out.println(text);

}

catch(Exception e)

{

}

}

}

**to read from a file using BufferedReader class**

import java.io.\*;

class Tests

{

public static void main(String[] args)

{

File file = new File("d:/myfile.txt");

try (BufferedReader br = new BufferedReader(new FileReader(file)))

{

String line;

while ((line = br.readLine()) != null) {

System.out.println(line);

}

} catch (IOException e) {

e.printStackTrace();

}

}

}

**to write to a File using FileWriter class**

import java.io.\*;

class WriteTest

{

public static void main(String[] args)

{

try

{

File fl = new File("d:/myfile.txt");

String str="Write this string to my file";

FileWriter fw = new FileWriter(fl) ;

fw.write(str);

fw.close();

}

catch (IOException e)

{ e.printStackTrace(); }

}

}

# FileInputStream Class

Java FileInputStream class obtains input bytes from a [file](https://www.javatpoint.com/java-file-class). It is used for reading byte-oriented data (streams of raw bytes) such as image data, audio, video etc. You can also read character-stream data. But, for reading streams of characters, it is recommended to use [FileReader](https://www.javatpoint.com/java-filereader-class) class.

## FileInputStream class methods

|  |  |
| --- | --- |
| **Method** | **Description** |
| int available() | It is used to return the estimated number of bytes that can be read from the input stream. |
| int read() | It is used to read the byte of data from the input stream. |
| int read(byte[] b) | It is used to read up to **b.length** bytes of data from the input stream. |
| void close() | It is used to closes the [stream](https://www.javatpoint.com/java-8-stream). |

import java.io.FileInputStream;

public class DataStreamExample {

     public static void main(String args[]){

          try{

            FileInputStream fin=new FileInputStream("D:\\testout.txt");

            int i=fin.read();

            System.out.print((char)i);

            fin.close();

          }catch(Exception e){System.out.println(e);}

         }

        }

# DataInputStream Class

Java DataInputStream [class](https://www.javatpoint.com/object-and-class-in-java) allows an application to read primitive data from the input stream in a machine-independent way.

Java application generally uses the data output stream to write data that can later be read by a data input stream.

|  |  |
| --- | --- |
| **Method** | **Description** |
| int read(byte[] b) | It is used to read the number of bytes from the input stream. |
| int readInt() | It is used to read input bytes and return an int value. |
| byte readByte() | It is used to read and return the one input byte. |
| char readChar() | It is used to read two input bytes and returns a char value. |
| double readDouble() | It is used to read eight input bytes and returns a double value. |
| boolean readBoolean() | It is used to read one input byte and return true if byte is non zero, false if byte is zero. |

import java.io.\*;

public class DataStreamExamples {

  public static void main(String[] args) throws IOException {

    InputStream input = new FileInputStream("D:\\testout.txt");

    DataInputStream inst = new DataInputStream(input);

    int count = input.available();

    byte[] ary = new byte[count];

    inst.read(ary);

    for (byte bt : ary) {

      char k = (char) bt;

      System.out.print(k+"-");

    }

  }

}

Here, we are assuming that you have following data in **"testout.txt"** file:

JAVA

Output:

J-A-V-A

# InputStream Class

The InputStream class of the java.io package is an abstract superclass that represents an input stream of bytes.

Since InputStream is an abstract class, it is not useful by itself. However, its subclasses can be used to read data.

## Methods of InputStream

The InputStream class provides different methods that are implemented by its subclasses. Here are some of the commonly used methods:

* read() - reads one byte of data from the input stream
* read(byte[] array) - reads bytes from the stream and stores in the specified array
* available() - returns the number of bytes available in the input stream
* mark() - marks the position in the input stream up to which data has been read
* reset() - returns the control to the point in the stream where the mark was set
* markSupported() - checks if the mark() and reset() method is supported in the stream
* skips() - skips and discards the specified number of bytes from the input stream
* close() - closes the input stream

import java.io.FileInputStream;

import java.io.InputStream;

class InoutDemo {

public static void main(String args[]) {

byte[] array = new byte[100];

try {

InputStream input = new FileInputStream("input.txt");

System.out.println("Available bytes in the file: " + input.available());

// Read byte from the input stream

input.read(array);

System.out.println("Data read from the file: ");

// Convert byte array into string

String data = new String(array);

System.out.println(data);

// Close the input stream

input.close();

} catch (Exception e) {

e.getStackTrace();

}

}

}

**Output**

Available bytes in the file: 39

Data read from the file:This is a line of text inside the file

# DataOutputStream Class

Java DataOutputStream [class](https://www.javatpoint.com/object-and-class-in-java) allows an application to write primitive [Java](https://www.javatpoint.com/java-tutorial) data types to the output stream in a machine-independent way.

Java application generally uses the data output stream to write data that can later be read by a data input stream.

|  |  |
| --- | --- |
| **Method** | **Description** |
| int size() | It is used to return the number of bytes written to the data output stream. |
| void write(int b) | It is used to write the specified byte to the underlying output stream. |
| void write(byte[] b, int off, int len) | It is used to write len bytes of data to the output stream. |
| void writeBoolean(boolean v) | It is used to write Boolean to the output stream as a 1-byte value. |
| void writeChar(int v) | It is used to write char to the output stream as a 2-byte value. |
| void writeChars(String s) | It is used to write [string](https://www.javatpoint.com/java-string) to the output stream as a sequence of characters. |
| void writeByte(int v) | It is used to write a byte to the output stream as a 1-byte value. |
| void writeBytes(String s) | It is used to write string to the output stream as a sequence of bytes. |
| void writeInt(int v) | It is used to write an int to the output stream |
| void writeShort(int v) | It is used to write a short to the output stream. |
| void writeShort(int v) | It is used to write a short to the output stream. |
| void writeLong(long v) | It is used to write a long to the output stream. |

import java.io.\*;

public class OutputExample {

    public static void main(String[] args) throws IOException {

        FileOutputStream file = new FileOutputStream(D:\\testout.txt);

        DataOutputStream data = new DataOutputStream(file);

        data.writeInt(65);

        data.flush();

        data.close();

        System.out.println("Succcess...");

    }

}

Output:

Succcess...

testout.txt:

A

# FileOutputStream Class

Java FileOutputStream is an output stream used for writing data to a [file](https://www.javatpoint.com/java-file-class).

If you have to write primitive values into a file, use FileOutputStream class. You can write byte-oriented as well as character-oriented data through FileOutputStream class. But, for character-oriented data, it is preferred to use [FileWriter](https://www.javatpoint.com/java-filterwriter-class) than FileOutputStream.

|  |  |
| --- | --- |
| void write(byte[] ary) | It is used to write **ary.length** bytes from the byte [array](https://www.javatpoint.com/array-in-java) to the file output stream. |
| void close() | It is used to closes the file output stream. |

import java.io.FileOutputStream;

public class FileOutputStreamExample {

    public static void main(String args[]){

           try{

             FileOutputStream fout=new FileOutputStream("D:\\testout.txt");

        fout.write(65);

//String s="Ram And Bharat.";// byte b[]=s.getBytes();fout.write(b);fout.close();

           System.out.println("success...");

            }catch(Exception e){System.out.println(e);}

      }

}

Success...

The content of a text file **testout.txt** is set with the data **A**.

testout.txt

A

**OutputStream** :

This class is the superclass of all classes representing an output stream of bytes.

|  |  |
| --- | --- |
| 1 | [void close()](https://www.tutorialspoint.com/java/io/outputstream_close.htm)  This method closes this output stream and releases any system resources associated with this stream. |
| 2 | [void flush()](https://www.tutorialspoint.com/java/io/outputstream_flush.htm)  This method flushes this output stream and forces any buffered output bytes to be written out. |
| 3 | [void write(byte[] b)](https://www.tutorialspoint.com/java/io/outputstream_write_byte.htm)  This method writes *b.length* bytes from the specified byte array to this output stream. |

import java.io.FileOutputStream;

import java.io.OutputStream;

public class OutSDemo {

public static void main(String args[]) {

String data = "ram bharat laxman shatrughan";

try {

OutputStream out = new FileOutputStream("output.txt");

// Converts the string into bytes

byte[] dataBytes = data.getBytes();

// Writes data to the output stream

out.write(dataBytes);

System.out.println("Data is written to the file.");

// Closes the output stream

out.close();

}

catch (Exception e) {

e.getStackTrace();

}

}

}

**StreamTokenizer**

class takes an input stream and parses it into "tokens", allowing the tokens to be read one at a time. The stream tokenizer can recognize identifiers, numbers, quoted strings, and various comment styles.

Now, we need to understand the default configuration. We have the following types of characters:

* *Word characters*: ranges like ‘a' to ‘z' and ‘A' to ‘Z
* *Numeric characters*: 0,1,…,9
* *Whitespace characters*: ASCII values from 0 to 32
* *Comment character*: /
* *String quote characters*: ‘ and “

These fields are :

* **nval** field contains the value of the number.
* **String sval** − If the current token is a word token, this field contains a string giving the characters of the word token.
* **TT\_EOL** is used to determine end of line
* **static int TT\_EOF** − A constant indicating that the end of the stream has been read.
* **ttype** field contains the type of the token just read.
* **TT\_WORD** indicates that the token is a word.
* **TT\_NUMBER** indicates that the token is a number.

|  |  |
| --- | --- |
| **Sr.No.** | **Method & Description** |
| 1 | [void commentChar(int ch)](https://www.tutorialspoint.com/java/io/streamtokenizer_commentchar.htm)  Specified that the character argument starts a single-line comment. |
| 2 | [void eolIsSignificant(boolean flag)](https://www.tutorialspoint.com/java/io/streamtokenizer_eolissignificant.htm)  This method determines whether or not ends of line are treated as tokens. |
| 3 | [int lineno()](https://www.tutorialspoint.com/java/io/streamtokenizer_lineno.htm)  This method returns the current line number. |
| 4 | [void lowerCaseMode(boolean fl)](https://www.tutorialspoint.com/java/io/streamtokenizer_lowercasemode.htm)  This method determines whether or not word token are automatically lowercased. |

[int nextToken()](https://www.tutorialspoint.com/java/io/streamtokenizer_nexttoken.htm)

This method parses the next token from the input stream of this tokenizer.

**Example 1**

import java.io.IOException;

import java.io.Reader;

import java.io.StreamTokenizer;

import java.io.StringReader;

public class JavaStreamTokenizerExample {

public static void main(String[] args) throws IOException {

Reader reader = new StringReader("This is a test string for JCG Stream

Tokenizer Example");

StreamTokenizer tokenizer = new StreamTokenizer(reader);

while(tokenizer.nextToken()!=StreamTokenizer.TT\_EOF){

System.out.println(tokenizer.sval);

}

}

}

**Example 2**

import java.io.FileReader;

import java.io.IOException;

import java.io.StreamTokenizer;

public class JavaStreamTokenizerExamples {

public static void main(String[] args) throws IOException {

//Read from a file

FileReader fileReader = new FileReader("test.txt");

StreamTokenizer filetokenizer = new StreamTokenizer(fileReader);

while(filetokenizer.nextToken()!=StreamTokenizer.TT\_EOF){

if(filetokenizer.ttype==StreamTokenizer.TT\_NUMBER){

System.out.println(filetokenizer.nval);

}else if(filetokenizer.ttype==StreamTokenizer.TT\_WORD){

System.out.println(filetokenizer.sval);

}

}

}

}