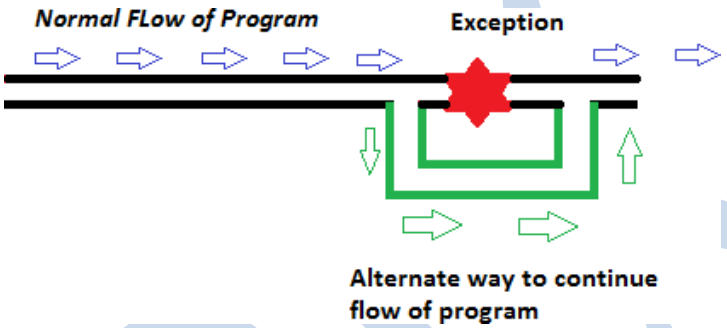
UNIT III

EXCEPTION HANDLING AND I/O

EXCEPTION

The exception handling in java is one of the powerful mechanisms to handle the runtime errors so that normal flow of the application can be maintained. Exception is the unexpected event which disturbs the normal flow of the program.

An alternate way to continue the program is called Exception Handling. The process of converting system error messages into user friendly error message is known as Exception handling.



An exception can occur for many different reasons. Following are some scenarios where an exception occurs.

- A user has entered an invalid data.
- A file that needs to be opened cannot be found.
- A network connection has been lost in the middle of communications or the JVM has run out of memory.

Categories of Exception:

Exceptions are categorized into 3 categories.

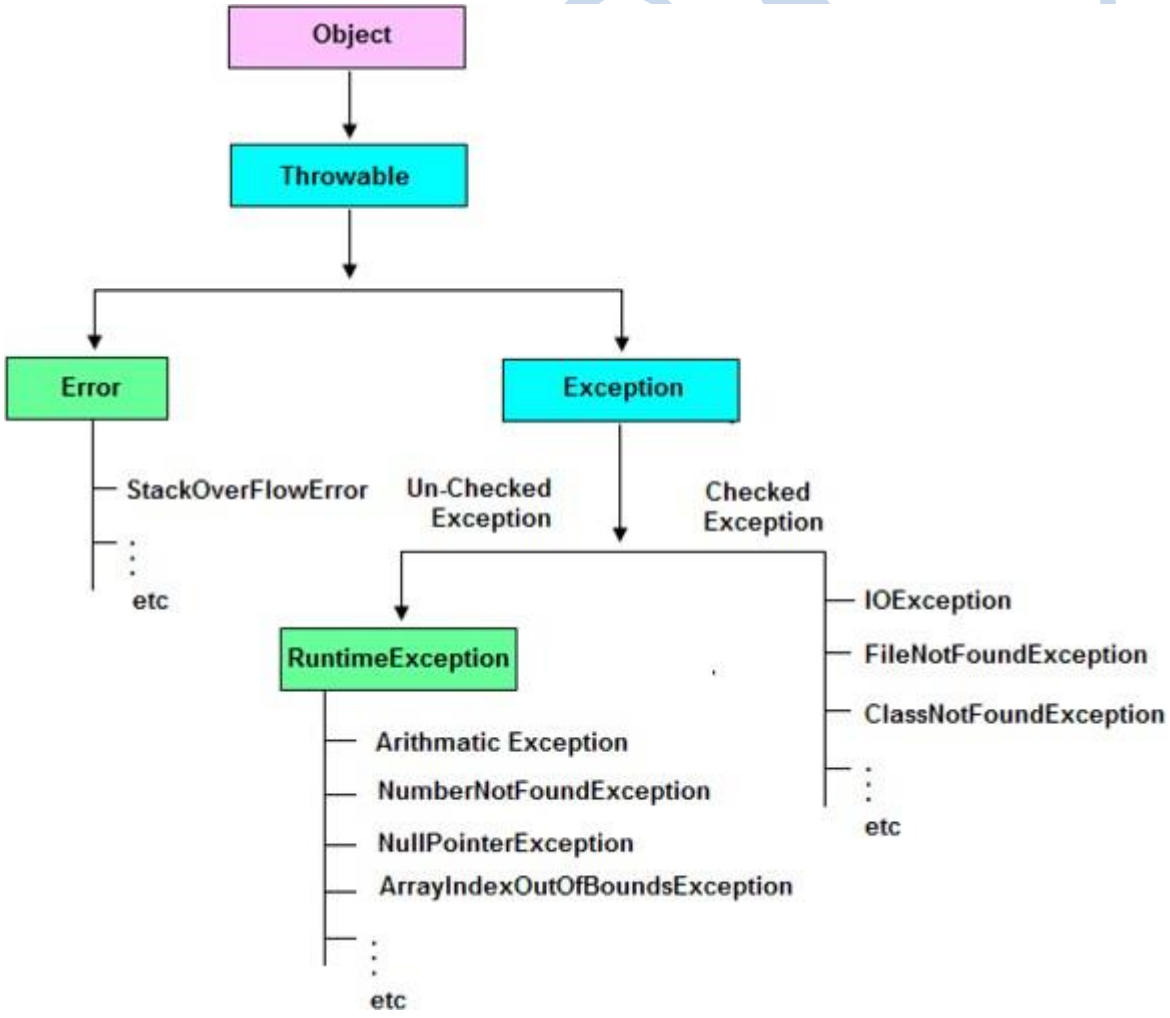
- Checked Exception
    - The exception that can be predicted by the programmer at the compile time.
    - These type of exceptions must be checked at compile time.
    - Example : File that need to be opened is not found.
- Unchecked Exception
    - Unchecked exceptions are the class that extends RuntimeException. Unchecked exceptions are ignored at compile time.

- Example: ArithmeticException, NullPointerException, ArrayIndexoutof Bound exception.
- Unchecked exceptions are checked at runtime.

- Error
  - Errors are typically ignored in code because you can rarely do anything about an error.
  - This type of error cannot be handled in the code.
  - Example: if stack overflow occurs, an error will arise.

Hierarchy of Exception classes:

All exception types are subclasses of class *Throwable*, which is at the top of exception class hierarchy.



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

Example 1:

- int a=50/0; → $ArithmeticException$
    - If we divide any number by zero, it raise an ArithmeticException.

Example 2:

- String s=null;
- System.out.println(s.length()); → $NullPointerException$
    - If we have null value in any variable, performing any operation by the variable occurs an NullPointerException.

Example 3:

- String s="abc";
- int i=Integer.parseInt(s); → $NumberFormatException$
    - Converting string into digit raise NumberFormatException.

Example 4:

- int a[]=new int[5];
- a[10]=50; → $ArrayIndexOutOfBoundsException$
    - Inserting any value in the wrong index, it results in ArrayIndexOutOfBoundsException

Java Exception Handling Keywords



Before understanding the use of these keywords lets us define some basic terms-

- Exception: Exception is a representation of error condition.
- Throwing an Exception: When we want to detect an abnormal condition then it is called throwing an exception.
- Catching an Exception: The Exception which is thrown must be caught. this process is called catching of exception.
- Handling an Exception: The Exception must be processed properly this process is called handling of exception.

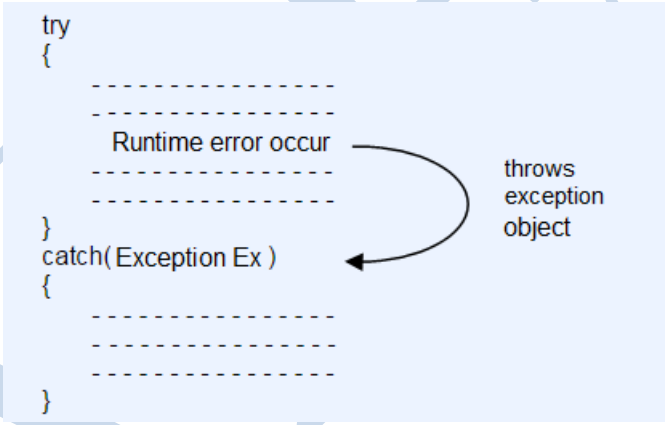
Java exception handling is managed via five keywords: try, catch, throw, throws, and finally

- Try : The try block contains statements which may generate exceptions.
- $Catch$:The catch block defines the action to be taken, when an exception occur.
- $Throw$: When an exception occurs in try block, it is thrown to the catch block using throw keyword.
- $Throws$: Throws keyword is used in situation, when we need a method to throw an exception.
- $Finally$: If exception occur or not, finally block will always execute.

Catching Exceptions

A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch is:

Syntax:



Filename : Test.java

```
public class Test
{
  public static void main(String args[])
  {
    int data = 50/0;
    System.out.println("Without Exception Handling...");
  }
}
```

→ **Division by Zero raise an ERROR**

→ **WITHOUT Exception handling**

Output:

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

Filename : Test.java

USING EXCEPTION HANDLING

```
public class Test
{
  public static void main(String args[])
  {
   try
   {
    int data = 50/0;
   }
```

→ **TRY BLOCK**

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

→ **CATCH BLOCK**

```
   System.out.println("Exception handling using TRY-CATCH block...");
  }
}
```

Output:

```
java.lang.ArithmeticException: / by zero
Exception handling using TRY-CATCH block...
```

Multiple Catch Blocks

A try block can be followed by multiple catch blocks. The syntax for multiple catch blocks is:

Syntax:

try {

// Protected code }

catch (*ExceptionType*1 *e*1) {

//Catch block }

catch (*ExceptionType*2 *e*2) {

// Catch block }

catch (*ExceptionType*3 *e*3) {

// Catch block }

Filename : Test.java

MULTIPLE CATCH STATEMENTS

```
public class Test
{
 public static void main(String[] args)
 {
  try
  {
    int arr[]={1,2};
    arr[10] = 3/0;
  }
  catch(ArithmeticException e)
  {
   System.out.println("Divide by Zero Exception");   }
  catch(ArrayIndexOutOfBoundsException e)
  {
   System.out.println("Array Index out of bound Exception");   }
 } }
```

TRY block

Accessing 9th array value, but the array limit is 2 elements only.
Dividing by zero too

Output:

```
Divide by Zero Exception
```

Note: Although both *ArrayIndexOutOfBoundsException* and *ArithmeticException* occured, but since first catch is of Arithmetic Exception, it will be caught there and program control will be continued after the catch block.

Finally block

Java finally block is used to execute important code such as closing connection, stream etc. Finally block is always executed whether exception is handled or not. It follows try or catch block.

```
try {
      // block of code to monitor for errors
}
catch (ExceptionType1  e1) {
      // exception handler for ExceptionType1
}
catch (ExceptionType2 e2) {
      // exception handler for ExceptionType2
}
// ...
finally {
      // block of code to be executed before try block
ends
```

Filename : Test.java

JAVA FINALLY OCCUR EVEN WHEN EXCEPTION DOESN'T OCCUR

```
public class Test
{
  public static void main(String args[]) {
  try
  {
   int data = 100/5;
   System.out.println("Given data is :" + data);   }

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

  finally
  {
    System.out.println("Finally block executed always");
  }
  System.out.println("End of Program...");
  }
}
```

Output:

```
Given data is :20
Finally block executed always
End of Program...
```

Filename : Test.java

JAVA FINALLY OCCUR DURING EXCEPTION

```
public class Test
{
  public static void main(String args[]) {
  try
  {
  int data = 100/0;
   System.out.println("Given data is :" + data);   }

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

  finally
  {
    System.out.println("Finally block executed always");
  }
  System.out.println("End of Program...");
  } }
```

Division by Zero

Output:

```
java.lang.ArithmeticException: / by zero
Finally block executed always
End of Program..
```

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.

Syntax:

$throw\ instance;$

Example: $throw\ new\ ArithmeticException("Division\ by\ Zero");$

But this exception must be of type $Throwable$ or a subclass of $Throwable$.

Filename : Test.java

THROW EXCEPTION when the condition is not satisfied

```
public class Test
{
  public static void main(String args[])
   {
     int age=5, weight=50;
     System.out.println("Welcome to the Registration process!!");

     if (age<18 || weight <48)
      throw new ArithmeticException("Student not eligible");
     else
       System.out.println("Student Entry is Valid!!");

    System.out.println("Have a nice day..");
 }
}
```

Output:

```
Welcome to the Registration process!!
Exception in thread "main" java.lang.ArithmeticException:
Student not eligible
```

If the student age<18 or weight <48 then the program should return that the student is not eligible for registration.

Throws

```
class SomeClass {

    public void method(int i) throws IllegalArgumentException {
        if (i < 0)
            throw new IllegalArgumentException();

        // ...
    }

}
```

Says "this method throws the checked exception IllegalArgumentException".

Throws an IllegalArgumentException.

Difference between throw and throws:

throw	throws
Java throw keyword is used to explicitly throw an exception.	Java throws keyword is used to declare an exception.
Checked exception cannot be propagated using throw only.	Checked exception can be propagated with throws.
Throw is followed by an instance.	Throws is followed by class.
Throw is used inside the method.	Throws is used in the method signature.
Multiple exceptions cannot be thrown	Multiple exceptions can be declared. e.g. public void method()**throws** IOException,SQLException.

Nested try catch block in Java

When a try catch block is present in another try block then it is called the nested try catch block.

Syntax of Nested try Catch

```
try {
  statement 1;
  statement 2;
    try {
     statement 3;
     statement 4;
       try {
       statement 5;
       statement 6;
         }
       catch(Exception e2) {
         …
         }
     }
     catch(Exception e1) {
       …
       }
   }
catch(Exception e3) {
   …
  }
```

Filename : Test.java

```
public class Test
{
 public static void main(String args[])
 {
  try                                          Outer try catch
  {
    try
    {
      System.out.println("Going to Divide");
      int b =39/0;
      }
    catch(ArithmeticException e)
    {
      System.out.println(e);
    }
```

→ try catch Block 1

```
    try
    {
      int a[]=new int[5];
      a[10] = 4;
      }
    catch(ArrayIndexOutOfBoundsException e)
    {
      System.out.println(e);
    }
```

→ try catch Block 2

```
    System.out.println("Other statement");
  }
  catch(Exception e)
   {
    System.out.println("Handled");
    }
  System.out.println("Normal flow..");
 }
}
```

Output:

```
Going to Divide
java.lang.ArithmeticException: / by zero
java.lang.ArrayIndexOutOfBoundsException: 10
Other statement
Normal flow..
```

Built-in Exceptions in Java

Java defines several exception classes inside the standard package **java.lang**. Following is the list of Java Unchecked *RuntimeException*.

<table>
<tr><th>S.No</th><th>Description of the Exception</th></tr>
<tr><td>1</td><td>

Arithmetic Exception

It is thrown when an exceptional condition has occurred in an arithmetic operation.

```
// Java program to demonstrate ArithmeticException
public class Test {
public static void main(String args[])
 {
   try {
      System.out.println("Result = " + (20/0));   // cannot divide by zero
      }

   catch (ArithmeticException e) {
      System.out.println("Can't divide 20 by 0");
      }    } }
```

```
Output:
Can't divide 20 by 0
```

</td></tr>
<tr><td>2</td><td>

ArrayIndexOutOfBoundException

It is thrown to indicate that an array has been accessed with an illegal index.

```
// Java program to demonstrate ArrayIndexOutOfBoundException
public 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 B unds
```

</td></tr>
<tr><td>3</td><td>

ClassNotFoundException

This Exception is raised when we try to access a class whose definition is not found.

```
// Java program to demonstrate ClassNotFoundException
public class Student
{
  public static void main(String args[])
  {
    try
    {
      Class.forName("Test");
    }
    catch(ClassNotFoundException e)
    {
      System.out.println("Sorry, class is not existing. " + e);
    }
  }
}
```

```
Output:
Sorry, class is not existing.
java.lang.ClassNotFoundException: Test
```

</td></tr>
</table>

<table>
<tr><th>S.No</th><th>Description of the Exception</th></tr>
<tr><td>4</td><td>

FileNotFoundException

This Exception is raised when a file is not accessible or does not open.

```
// Java program to demonstrate FileNotFoundException
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
public class File_notFound_Demo {

public static void main(String args[])
   {
      try {
         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
```

</td></tr>
<tr><td>5</td><td>

NullPointerException

It is thrown when referring to the members of a null object. Null represents nothing

```
// Java program to demonstrate NullPointerException
public 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..
```

</td></tr>
<tr><td>6</td><td>

NumberFormatException

It is thrown when a method could not convert a string into a numeric format.

```
// Java program to demonstrate NumberFormatException
public class Test {
public static void main(String args[])
   {
      try {
         // "CSE" is not a number
         int num = Integer.parseInt("CSE");
      }
      catch (NumberFormatException e) {
         System.out.println("Number format exception");
      } } }
```

```
Output:
Number format exception
```

</td></tr>
<tr><td>7</td><td>

RuntimeException

This represents any exception which occurs during runtime.

</td></tr>
</table>

S.No	Description of the Exception

8	***StringIndexOutOfBoundsException***
	It is thrown by String class methods to indicate that an index is either negative than the size of the string
	`// Java program to demonstrate StringIndexOutOfBoundsException`
	`public class Test {`
	`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:`
	`StringIndexOutOfB undsExc`
	`eption`

9	***IOException*** It is thrown when an input-output operation failed or interrupted
10	***InterruptedException*** It is thrown when a thread is waiting, sleeping, or doing some processing, and it is interrupted.
11	***NoSuchFieldException*** It is thrown when a class does not contain the field (or variable) specified
12	***NoSuchMethodException*** It is thrown when accessing a method which is not found.
13	***IOException*** It is thrown when an input-output operation failed or interrupted

User-defined Custom Exception in Java

Java provides us facility to create our own exceptions which are basically derived classes of Exception. Apart from the built-in exception classes we can define your own exception classes.

This is very useful when we want to define exception types that are having a different behaviour from the standard exception type.

Filename : Test.java

```
class StudentAgeException extends Exception
{
   StudentAgeException(String s)
   {
     System.out.println(s);
   }
}
```

Creating own Exceptions

```
public class Test
{
   public static void main(String args[])
   {
      int age = 16;
      try
      {
        if (age<21)
          throw new StudentAgeException("At age 21, you can register");
      }
      catch (StudentAgeException ex)
      {
        System.out.println("Caught the Exception");
      }
   }
}
```

try catch Block

Output:

```
At age 21, you can register
Caught the Exception
```

Filename : Test.java

```
class StudentAgeException extends Exception
{
   StudentAgeException(String s)
   {
     super(s);
   }
}

public class Test
{
   public static void main(String args[])
   {
      int age = 16;
      try
      {
       if (age<21)
         throw new StudentAgeException("At age 21, you can register");
      }
      catch (StudentAgeException ex)
      {
        System.out.println("Caught the Exception");
      }
   }
}
```

Output:

```
Caught the Exception
```

APEC

A new exception called $\boldsymbol{StudentAgeException}$ is created in the above program. The constructor of $\boldsymbol{StudentAgeException}$ takes a string as its argument in the first example. The string is passed to parent class Exception's constructor using super() in the second example. The constructor of Exception class can also be called without a parameter and call to super is not mandatory.

Points to be remembered:

- All exceptions must be a child of $\boldsymbol{Throwable}$.
- If you want to write a checked exception, then extend the Exception class.
- If you want to write a runtime exception, extend the RuntimeException class.

Chained Exceptions

Chained Exception helps to identify a situation in which one exception causes another Exception in an application.

For instance, consider a method which throws an $\boldsymbol{ArithmeticException}$ because of an attempt to divide by zero but the actual cause of exception was an $\boldsymbol{I/O\ error}$ which caused the divisor to be zero.The method will throw the ArithmeticException to the caller. The caller would not know about the actual cause of an Exception. Chained Exception is used in such situations. This concept was introduced in **JDK 1.4.**

Constructors Of $\boldsymbol{Throwable\ class}$ that support chained exceptions in java

- $\boldsymbol{Throwable(Throwable\ cause)}$:- Where cause is the exception that causes the current exception.
- $\boldsymbol{Throwable(String\ msg, Throwable\ cause)}$:- Where msg is the exception message and cause is the exception that causes the current exception.

Methods Of $\boldsymbol{Throwable\ class}$ Which support chained exceptions in java

- $\boldsymbol{getCause()\ method}$: It returns actual cause of an exception.
- $\boldsymbol{initCause(Throwable\ cause)\ method}$: It sets the cause for the calling exception.

Filename : Test.java

```
public class Test
{
   public static void main(String[] args)
   {
      try
      {
       NumberFormatException ex=new NumberFormatException("Exception");
      ex.initCause(new NullPointerException("This is actual reason for exception"));
      throw ex;
      }

      catch(NumberFormatException ex)
      {
         System.out.println(ex);

         System.out.println(ex.getCause());
      }
   }
}
```

Creating an exception

Setting the cause for the exception

Throwing an exception with cause

Displaying the exception

Getting the actual cause of the exception

Output:

```
java.lang.NumberFormatException: Exception
java.lang.NullPointerException: This is actual reason for the exception
```

StackTraceElement

When a Java virtual machine runs a program, it needs memory to store many things. The memory is organized based on the following need:

- Loading classes - Extracts information from class files and stores them in method area.
- Creating Objects - As the program runs, JVM creates new objects and places them in the heap area
- Calling Methods - As each new thread comes into existence, it gets its own program counter (pc) and Java stack. The value of the pc register indicates the next instruction to execute. If the thread is executing a Java method, the thread's Java stack stores the method's local variables, the parameters with which it was invoked, its return value (if any), and intermediate calculations. All this is put onto the stack

Java Stack

The Java stack is composed of stack frames where each frame contains the state of one Java method invocation. When a thread invokes a method, the JVM pushes a new frame onto that thread's Java stack. When the method completes, the JVM pops and discards the frame for that method.

StackTraceElement Structure

Each stack frame is represented by $StackTraceElement$. The frame at the top of the stack at the zeroth element represents the execution point at which the stack trace was generated. The last element of the array represents the bottom of the stack, which is the first method invocation in the sequence. $Throwable.\ getStackTrace()$ provides programmatic access to the stack trace information.

java.lang.StackTraceElement class	
Class Declaration	
$\boldsymbol{public\ final\ class\ StackTraceElement\ extends\ Object\ implements\ Serializable}$	
Class Constructor & Description	
$\boldsymbol{StackTraceElement}$	$\boldsymbol{StackTraceElement(String\ declaringClass, String\ methodName, String\ fileName, int\ lineNumber)}$ This creates a stack trace element representing the specified execution point.
Class methods	
boolean **equals(Object obj)**	Returns true if the specified object is another StackTraceElement instance representing the same execution point as this instance.

Example

Syntax: boolean **equals(Object obj)**

Returns: Returns true if the specified object is another StackTraceElement instance representing the same execution point as this instance.

Exception: NA

```
import java.lang.*;
import java.io.*;
import java.util.*;
public class StackTraceElementDemo
{
   public static void main(String[] arg)
   {
      StackTraceElement st1 = new StackTraceElement("foo", "fuction1",
                                 "StackTrace.java", 1);
```

st1 & st2 are StackTraceElements

```
        StackTraceElement st2 = new StackTraceElement("bar", "function2",
                                        "StackTrace.java", 1);

        Object ob = st1.getFileName();

        // checking whether file names are same or not
        System.out.println(st2.getFileName().equals(ob));
    }
}
```

Output:
```
true
```

String **getClassName()**	Returns the fully qualified name of the class containing the execution point represented by this stack trace element.

Example

Syntax: String **getClassName()**

Returns: Returns the fully qualified name of the class containing the execution point represented by this stack trace element.

Exception: NA.

```
import java.lang.*;
import java.io.*;
import java.util.*;
public class StackTraceElementDemo
{
   public static void main(String[] arg)
   {
      System.out.println("Class name of each thread involved:");
      for(int i = 0; i<2; i++)
      {
        System.out.println(Thread.currentThread().getStackTrace()[i].getClassName());
      }
   }
}
```

Output:
```
Class name of each thread involved:
java.lang.Thread
StackTraceElementDemo
```

String **getFileName()**	Returns the name of the source file containing the execution point represented by this stack trace element.

Example

Syntax: String **getFileName()**

Returns: Returns the name of the source file containing the execution point represented by this stack trace element.

Exception: NA.

```
import java.lang.*;
import java.io.*;
import java.util.*;
public class StackTraceElementDemo
{     public static void main(String[] arg)
  {
      System.out.println("File name: ");
      for(int i = 0; i<2; i++)
      System.out.println(Thread.currentThread().getStackTrace()[i].getFileName());
  }
}
```

Output:
```
File name:
Thread.java
StackTraceElementDemo.java
```

int **getLineNumber()**	Returns the line number of the source line containing the execution point represented by this stack trace element.

Example
Syntax: int **getLineNumber()**
Returns: Returns the line number of the source line containing the execution point represented by this stack trace element.
Exception: NA.

```
import java.lang.*;
import java.io.*;
import java.util.*;
public class StackTraceElementDemo
{
  public static void main(String[] arg)
  {
     System.out.println("Line number: ");
     for(int i = 0; i<2; i++)
     System.out.println(Thread.currentThread().getStackTrace()[i].getLineNumber());
  }
}
```

Output:
```
Line Number :
1559
10
```

String **getMethodName()**	Returns the name of the method containing the execution point represented by this stack trace element.

Example
Syntax: String **getMethodName()**
Returns: Returns the name of the method containing the execution point represented by this stack trace element.
Exception: NA.

```
import java.lang.*;
import java.io.*;
import java.util.*;
public class StackTraceElementDemo
{
  public static void main(String[] arg)
  {      System.out.println("method name: ");
     for(int i = 0; i<2; i++)
       System.out.println(Thread.currentThread().getStackTrace()[i].getMethodName());
  }
}
```

Output:
```
Method Name:
getStackTrace
main
```

int **hashCode()**	Returns a hash code value for this stack trace element.

Example
Syntax: int **hashCode()**
Returns: Returns a hash code value for this stack trace element.
Exception: NA.

```
import java.lang.*;
import java.io.*;
import java.util.*;
public class StackTraceElementDemo {
  public static void main(String[] arg)
  {
```

Output:
```
Hash Code :
-1225537242
-1314176653
```

```
      System.out.println("Hash Code : ");
      for(int i = 0; i<2; i++)
      System.out.println(Thread.currentThread().getStackTrace()[i].hashCode());
   }
}
```

boolean **isNativeMethod()**	Returns true if the method containing the execution point represented by this stack trace element is a native method.

Example

Syntax: boolean **isNativeMethod()**

Returns: Returns true if the method containing the execution point represented by this stack trace element is a native method.

```
import java.lang.*;
import java.io.*;
import java.util.*;
public class StackTraceElementDemo
{
   public static void main(String[] arg)
   {
     System.out.println("Hash Code : ");
     for(int i = 0; i<2; i++)
       System.out.println(Thread.currentThread().getStackTrace()[i].isNativeMethod());
   }
}
```

Output:
```
false
false
```

String **toString()**	Returns a string representation of this stack trace element.

Example

Syntax: String **toString()**

Returns: Returns a string representation of this stack trace element.

```
import java.lang.*;
import java.io.*;
import java.util.*;
public class StackTraceElementDemo
{
   public static void main(String[] arg)
   {
      System.out.println("String equivlaent: ");
      for(int i = 0; i<2; i++)
        System.out.println(Thread.currentThread().getStackTrace()[i].toString());
   }
}
```

Output:
```
tring equivlaent:
java.lang.Thread.getStackTrace(Thread.java:1559)
StackTraceElementDemo.main(StackTraceElementDemo.java:10)
```

State the similarities and differences between Errors and Exception in Java

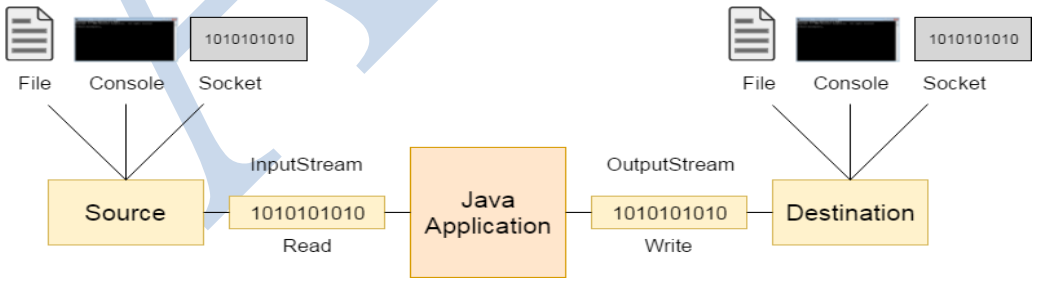
	Error	Exception
Similarities	Both of them are generated in *java.lang.Throwable* package	
	Can be caught by catch block	
Differences	**Cannot be** handled	**Can be** handled
Examples	*java.lang.OutOfMemoryError*	*Java.lang.ArrayIndexOutOfBoundsException*
	java.lang.NoClassDefFoundError	*Java.lang.ArithmeticException*
	java.lang.UnSupportedClassVersionError	*Java.lang.NullPointerException*
	Java.lang.NoSuchMethodError	*Java.lang.NumberFormatException*
		Java.lang.ClassNotFoundException
		Java.lang.NegativeArraySizeException

Input and Output statements in Java

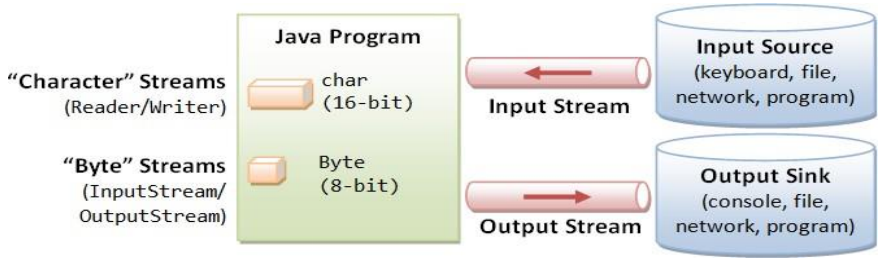
Java I/O (Input and Output) is used to process the input and produce the output. Java performs I/O through Streams. A Stream is used by java I/O system to make input and output operation in java. In general, a stream means continuous flow of data.

Define Stream:

A stream is a sequence of characters or bytes used for program input or output. In general there may be several input streams and output streams. Streams are unidirectional they can transmit bytes in only one direction. The input object that reads the stream of data from the file is called *Input Stream*. The output object that writes the stream of data to a file is called *Output stream*. Java provides many different input and output stream classes in the ***java.io API***.



Java application uses an input stream to read data from a source, it may be a file, an array, peripheral device or socket. An output stream accepts output bytes and sends them to some sink.



Commonly used IO objects are in ***java.io API***

System.in → Standard input stream

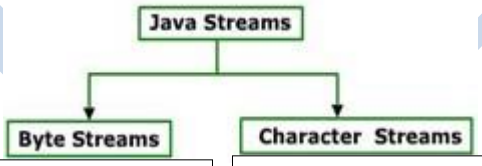
System.out → Standard output stream for printing results

System.err → Standard output stream for printing errors

These objects are used to input data from the user into the application, and output data to the user.

- ***System.in*** → connected to the keyboard and gets character data
- ***System. out*** and ***System. err*** → connected to the monitor and prints character data.

Java encapsulates Stream under ***java. io package***. The java.io package contains a large number of stream classes and is categorized into two groups based on the data type on which they operate.



- Handling I/O operations on bytes.
- Used when reading or writing binary data.
- Two kinds of bytes stream classes
  - Input stream class
  - Output stream class

- Handling I/O operations on Characters.
- Character stream uses Unicode and are more efficient
- Two kinds for character stream classes.
  - Reader Stream class
  - Writer Stream class

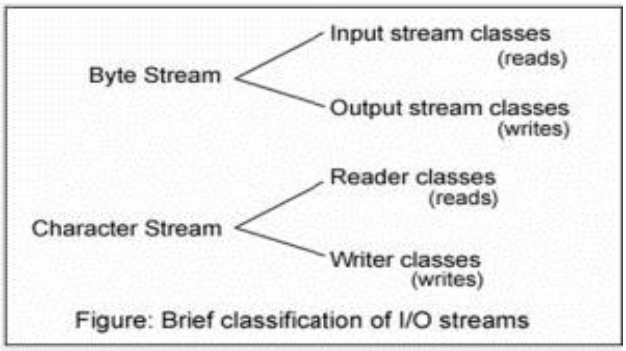


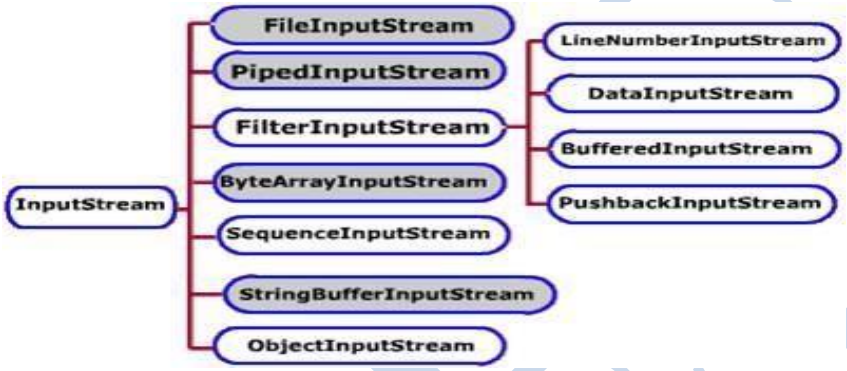
Figure: Brief classification of I/O streams

Byte Stream Classes:

Byte stream classes provide features for creating and manipulating streams and files for reading and writing bytes. Java provides two kinds of bytes stream classes i.e. *input stream class* and *output stream class*.

Input Stream Class

Input stream class is used to read 8 bit bytes, include a super class known as ***InputStream*** and a number of subclasses for supporting various input related functions.



Java.io.InputStream

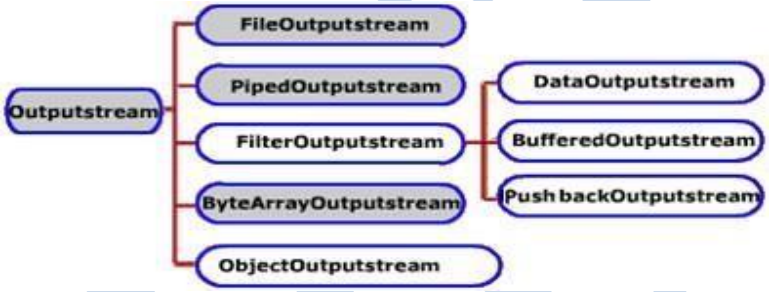
The ***Java.io.InputStream*** class is the superclass of all classes representing an input stream of bytes.

Class declaration
public abstract class InputStream *extends Object* *implements Closeable*
Class constructors
InputStream()
Class methods
int available() **:** It returns an estimate of the number of bytes that can be read (or skipped over) from this input stream
void close() **:** This method closes this input stream and releases any system resources associated with the stream.
void mark(int readlimit) **:** Marks the current position in the input stream.
boolean markSupported() : Tests if this input stream supports the mark and reset methods.
abstract int read() : Reads the next byte of data from the input stream.
int read(byte[] b) : Reads some number of bytes from the input stream and stores them into the buffer array b.

Class methods – Contd...
$int\ read(byte[]\ b, int\ off, int\ len)$: Reads up to **len bytes** of data from the input stream into an array of bytes.
$void\ reset()$: Repositions this stream to the position at the time the mark method was last called on this input stream.
$long\ skip(long\ n)$: Skips over and discards n bytes of data from this input stream.

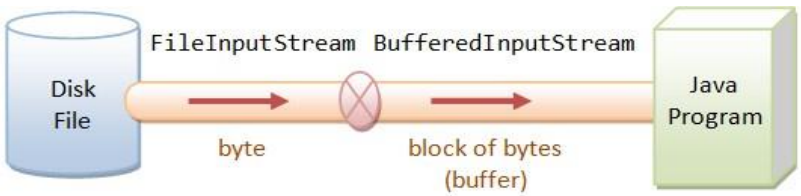
Output Stream Class

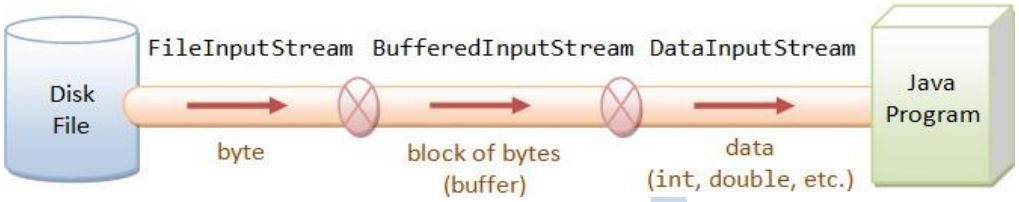
Output stream classes are derived from the base class $OutputStream$. This is an abstract class and has several subclasses and these subclasses can be used for performing output operations.



Outputstream methods
$close()$ It closes the file output stream.
$finalize()$ It cleans up the connection to the file.
$write(int\ w)$ It writes the specified byte to the output stream.
$write(byte[]\ w)$ Writes w.length bytes from the mentioned byte array to the OutputStream.

Working of Byte Streams:



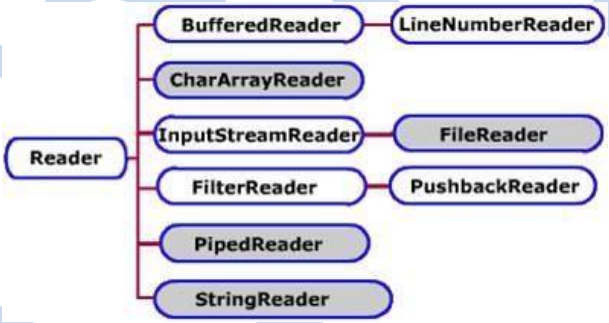


Character Stream class

Character stream classes can be used to read and write 16 bit Unicode characters.

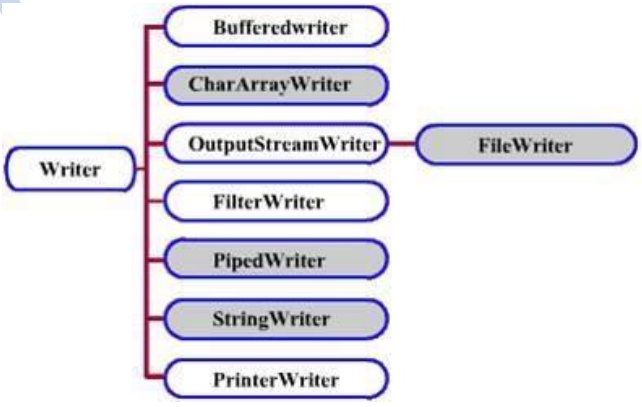
Reader Stream class

It is used to read characters from file. They are functionally same as InputStream classes but these read characters instead of bytes.



Writer Stream class

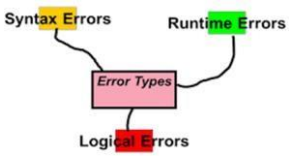
It is used to write characters to file. These are functionally same as OutputStream classes but these write characters instead of bytes.



Errors

Errors are caused by the mistakes in the program. There are three types of errors;

- Syntax errors
- Runtime Errors
- Logical Errors



Syntax errors are the grammatical errors. It prevents the program from running.

Example:

$x = 1000;$

$System.out.println(y)$

```
Syntax error: <identifier> expected
```

Runtime errors occur when we do something illegal

Example:

$int\ x = 1000;$

$int\ y = x + '100';$

```
Runtime error: unclosed character literal
```

Logical errors give the wrong output. Though the program is syntactically correct, it operates incorrectly.

Example:Suppose the program is to find the sum of first 5 numbers.

$int\ x = 1, sum = 0;$

$for\ (int\ i = 1;\ i < 5;\ i++)$

$\{$

$sum = sum + i;$

$\boldsymbol{System.out.println("Sum = " + sum);}$

$\}$

```
This print statement should
come outside the loop
```

Reading Console Input:

In Java, there are 3 ways for reading input from the user in the command line environment (console).

- BufferedReader + InputStreamReader (Classic)
- Scanner (JDK 1.5)
- System.console (JDK 1.6)

Using Buffered Reader Class

This is the Java classical method to take input, introduced in JDK1.0. This method is used by wrapping the System.in (standard input stream) in an ***InputStreamReader*** which is wrapped in a ***BufferedReader***, which can be used to read input from the user in the command line.

Merits

- The input is buffered for efficient reading.

Demerits

- The wrapping code is hard to remember.

Filename : Test.java

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class Test
{
   public static void main(String[] args) throws IOException
   {
BufferedReader r = new BufferedReader(new InputStreamReader (System.in));
        String name = r.readLine();
        System.out.println("My name is : " + name);
   }
}
```

Enter data using BufferReader

Reading data using readLine

Printing the readLine

Output:

```
My name is : Dhaya Chinnathambi
```

Note:
read() function is used to read character and readLine() function is used to read line or string. For other types, functions like Integer.parseInt(), Double.parseDouble() are used.

Filename : Test.java

```
import java.io.*;

public class CalcArea
{
  public static void main ( String args[] ) throws IOException
  {
    System.out.print ( "Enter the radius: " );
BufferedReader i = new BufferedReader (new InputStreamReader (System.in));

    String s = i.readLine();

    double radius = Double.parseDouble(s);

    double area = 3.14159 * radius * radius;

    System.out.println ( "Area of the Circle is: " + area );
  }
}
```

Input:
```
Enter the radius:20
```

Output:
```
Area of the Circle is: 1256.636
```

Using Scanner Class

This is probably the most preferred method to take input. The main purpose of the Scanner class is to parse primitive types and strings using regular expressions.

Filename : Test.java

```
import java.util.Scanner;

public class Test
{
   public static void main(String args[])
   {
      Scanner in = new Scanner(System.in);

      String s = in.nextLine();
      System.out.println("Entered string : "+s);

      int a = in.nextInt();
      System.out.println("Entered integer : "+a);

      float b = in.nextFloat();
      System.out.println("Entered float : "+b);
   }
}
```

→ **Using Scanner to get input from User**

Input:
```
Dhaya
10
23.5
```
Output:
```
Entered string : Dhaya
Entered integer : 10
Entered float : 23.5
```

Merits:

- Convenient methods for parsing primitives (nextInt(), nextFloat(), …) from the tokenized input.
- Regular expressions can be used to find tokens.

Demerits:

- The reading methods are not synchronized

Using Console Class

It is a preferred way for reading user's input from the command line. In addition, it can be used for reading password-like input without echoing the characters entered by the user.

Filename : Test.java

```
public class Sample
{
   public static void main(String[] args)
   {
      String name = System.console().readLine();
      System.out.println(name);
   }
}
```

Using Console to input data from user

Input:
CSE

Output:
CSE

Merits:

- Reading password without echoing the entered characters.
- Reading methods are synchronized.
- Format string syntax can be used.

Demerits:

- Does not work in non-interactive environment (such as in an IDE).

Writing Console output:

- Console output is most easily accomplished with $print()$ and $println()$ methods. These methods are defined by the **class PrintStream** which is the type of object referenced by **System.in**.
- **write()** can be used to write to the console.
- **Syntax:** $\boldsymbol{void\ write(int\ byteval)}$

Filename : Test.java

```
class Test
{
    public static void main(String args[])
    {
       int y ='X';
       System.out.write(y);
       System.out.write('\n');
    }
 }
```

Output:
X

Note:
You will not often use write() to perform console output (although doing so might be useful in some situations) because **print()** and **println()** are substantially easier to use.

Alternate method to print console output **– Using PrintWriter Class**

For real world programs, recommended method to write to the console in java is through a PrintWriter Stream. PrintWriter is one of the character based classes for console output in java.

Syntax: $PrintWriter(OutputStream\ outputStream, boolean\ flushOnNewline)$

Filename : Test.java

```
class Test
{
 public static void main(String args[])
{
  PrintWriter pw = new PrintWriter(System.out, true);
  pw.println("PrintWriter class demo");
  pw.println(100);
  pw.println(-4.67);
 }
}
```

Output:
PrintWriter class demo
100
-4.67

Note:
The PrintWriter supports the print() and println() methods. Thus, you can use these methods in the same way as you used them with System.out.

Reading and Writing Files

Reading a text file

There are several ways to read a plain text file in Java e.g. you can use FileReader, BufferedReader or Scanner to read a text file. Every utility provides something special e.g. BufferedReader provides buffering of data for fast reading, and Scanner provides parsing ability.

1. **Using BufferedReader:** This method reads text from a character-input stream. It does buffering for efficient reading of characters, arrays, and lines. The buffer size may be specified, or the default size may be used. The default is large enough for most purposes.

In general, each read request made of a Reader causes a corresponding read request to be made of the underlying character or byte stream. It is therefore advisable to wrap a BufferedReader around any Reader whose read() operations may be costly, such as FileReaders and InputStreamReaders. For example,

BufferedReader in = new BufferedReader(Reader in, int size);

Java Program to illustrate reading from FileReader using BufferedReader:

```
Filename : Test.java
import java.io.*;
public class Test
{
  public static void main(String[] args) throws Exception
  {
     File file = new File("C:\\Desktop\\test.txt");
     BufferedReader br = new BufferedReader(new FileReader(file));

    String st;
    while ((st = br.readLine()) != null)
        System.out.println(st);
  }
}
```

2. Using FileReader class: This class is for reading character files. The constructors of this class assume that the default character encoding and the default byte-buffer size are appropriate.

Constructors defined in this class are:

- Creates a new FileReader, given the File to read from.
  Syntax: $FileReader(File\ file)$

- Creates a new FileReader, given the FileDescriptor to read from.
  Syntax: $FileReader(FileDescriptor\ fd)$

- Creates a new FileReader, given the name of the file to read from.
  Syntax: $FileReader(String\ fileName)$

Java Program to illustrate reading from FileReader using FileReader

Filename : Test.java

```
import java.io.*;
public class Test
{
  public static void main(String[] args) throws Exception
  {
    //
    FileReader fr = new FileReader("C:\\Desktop\\test.txt");

    int i;
    while ((i=fr.read()) != -1)
        System.out.print((char) i);
  }
}
```

pass the path to the file as a parameter

3. Using Scanner class: A simple text scanner which can parse primitive types and strings using regular expressions.

A Scanner breaks its input into tokens using a delimiter pattern, which by default matches whitespace. The resulting tokens may then be converted into values of different types using the various next methods.

Java Program to illustrate reading from Text File using Scanner Class

```
Filename : Test.java
import java.io.File;
import java.util.Scanner;
public class Test
{
  public static void main(String[] args) throws Exception
  {
    File file = new File("C:\\Desktop\\test.txt");
    Scanner sc = new Scanner(file);

    while (sc.hasNextLine())
      System.out.println(sc.nextLine());
  }
}
```

Writing to a text file

1. Java **FileWriter class** is used to write character-oriented data to a file.

```
Filename : Test.java

import java.io.FileWriter;
public class Test
{
   public static void main(String args[])
   {
        try
        {
           FileWriter fw=new FileWriter("D:\\testout.txt");
           fw.write("Welcome to java.");
           fw.close();
            }

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

2. Java **BufferedWriter class** is used to provide buffering for Writer instances. It makes the performance fast. It inherits Writer class. The buffering characters are used for providing the efficient writing of single arrays, characters, and strings.

```
Filename : Test.java

import java.io.*;
public class Test
{
public static void main(String[] args) throws Exception
{
    FileWriter writer = new FileWriter("D:\\testout.txt");
    BufferedWriter buffer = new BufferedWriter(writer);
    buffer.write("Welcome to java.");
    buffer.close();
    System.out.println("Success");
    }
}
```

3. **Using FileOutputStream** is an output stream used for writing data to a file.

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 than FileOutputStream.

```
Filename : Test.java

import java.io.FileOutputStream;
public class Test
{
    public static void main(String args[])
    {
       try
       {
       FileOutputStream fout=new FileOutputStream("D:\\testout.txt");
        fout.write(65);
        fout.close();
        System.out.println("success...");
       }

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

2 marks:

1. List the types of errors. P.No.148
2. List the reasons for raising the errors. P.No.123
3. Mention the categories of Exception P.No.123
4. Draw the Hierarchy of Exception P.No.124
5. Explain Nested try catch block in Java P.No.131
6. State the similarities and differences between Errors and Exception in Java 143
7. Define Stream P.No.143
8. What are the types of streams in Java P.No.144

13 marks

1. Explain Java Exception Handling in detail P.No.125 – 129
2. (i) How to throw an exception . Give example P.No.130
   (ii) State the difference between throw and throws. P.No.131
3. Give some Built-in Exceptions in Java. Explain in detail. P.No.133 – 135
4. Create a User-defined Custom Exception in Java P.No.135 - 136
5. Explain the concept of Chained Exceptions in Java P.No.136 – 137
6. What is StackTraceElement? Explain any 5 methods P.No.138 – 142
7. What are the types of streams in Java? Explain in detail P.No.144 – 147
8. How to read and write files in Java P.No.153 – 156
9. How to read and write using console? P.No.149 – 152

Difference between throw, throws and Throwable in java(Interview question)

Throw

- Throw is a keyword in java which is used to throw an exception manually.
- Must be of type **java.lang.Throwable class** or it's sub classes.

```
class Test
 {
    void method()
    {
      throw e;
    }
 }
```

Throws

Throws is also a keyword in java which is used in the method signature to indicate that this method may throw mentioned exceptions.

```
class Test
 {
   void methodOne() throws SQLException
   {
      //This method may throw SQLException
   }

   void methodTwo() throws IOException
   {
      //This method may throw IOException
   }

   void methodThree() throws ClassNotFoundException
   {
      //This method may throw ClassNotFoundException
   }
}
```

Throwable In Java

Throwable is a super class for all types of errors and exceptions in java. This class is member of java.lang package.

```
class MyException extends Throwable
 {
  …
  }

class Test
{
   void method() throws MyException
   {
       throw MyException;
   }
}
```

```
public class StackTraceExample
{
 public static void main(String[] args)
 {
   method1();
 }

 public static void method1()
 {
   method11();
 }

  public static void method11()
  {
    method111();
  }

   public static void method111()
   {
    throw new NullPointerException("Fictitious NullPointerException");
  }
}
```

Output:

```
Exception in thread "main" java.lang.NullPointerException:
Fictitious NullPointerException
        at StackTraceExample.method111(StackTraceExample.java:15)
        at StackTraceExample.method11(StackTraceExample.java:11)
        at StackTraceExample.method1(StackTraceExample.java:7)
        at StackTraceExample.main(StackTraceExample.java:3)
```

AREC