Exceptions Handling in Java

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

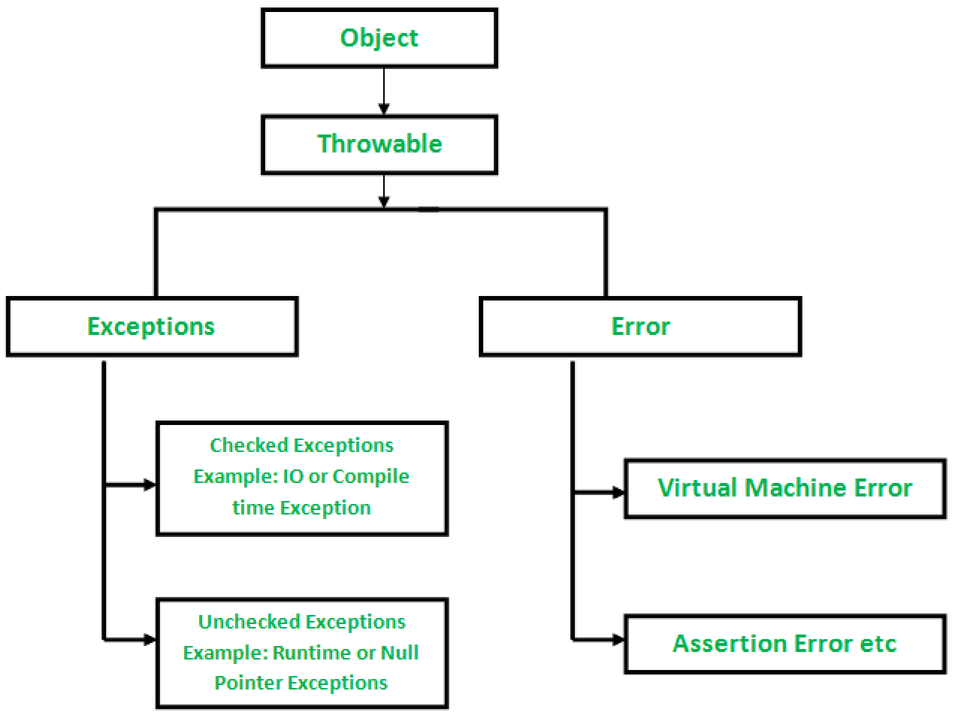
Errors represent irrecoverable conditions such as Java virtual machine (JVM) running out of memory, memory leaks, stack overflow errors, library incompatibility, infinite recursion, etc. Errors are usually beyond the control of the programmer and we should not try to handle errors.

Let us discuss the most important part which is the **differences between Error and Exception** that is as follows:

- **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.

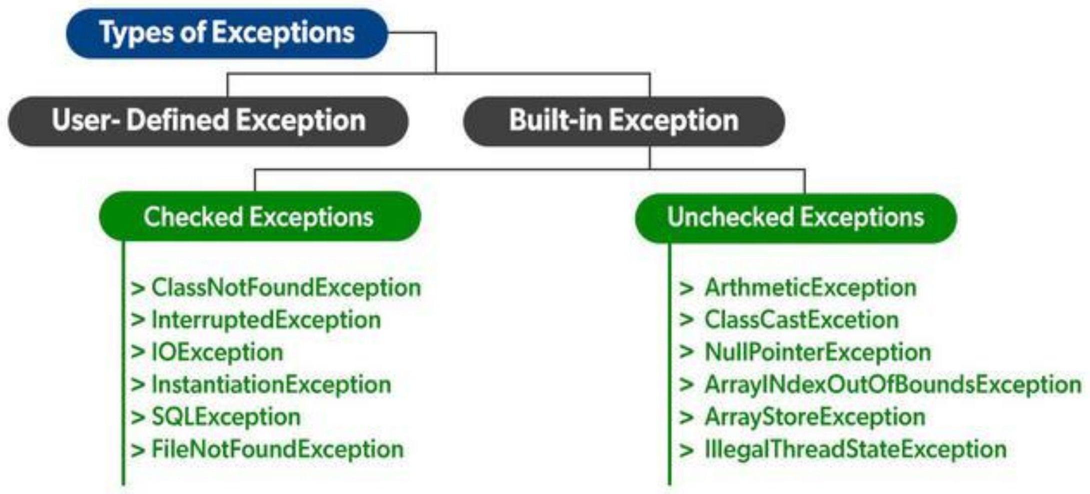
Exception Hierarchy

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.


Object
Throwable
Exceptions
Error
Checked Exceptions Example: IO or Compile time Exception
Unchecked Exceptions Example: Runtime or Null Pointer Exceptions
Virtual Machine Error
Assertion Error etc

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**

Let us discuss the above-defined listed exception that is as follows:

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

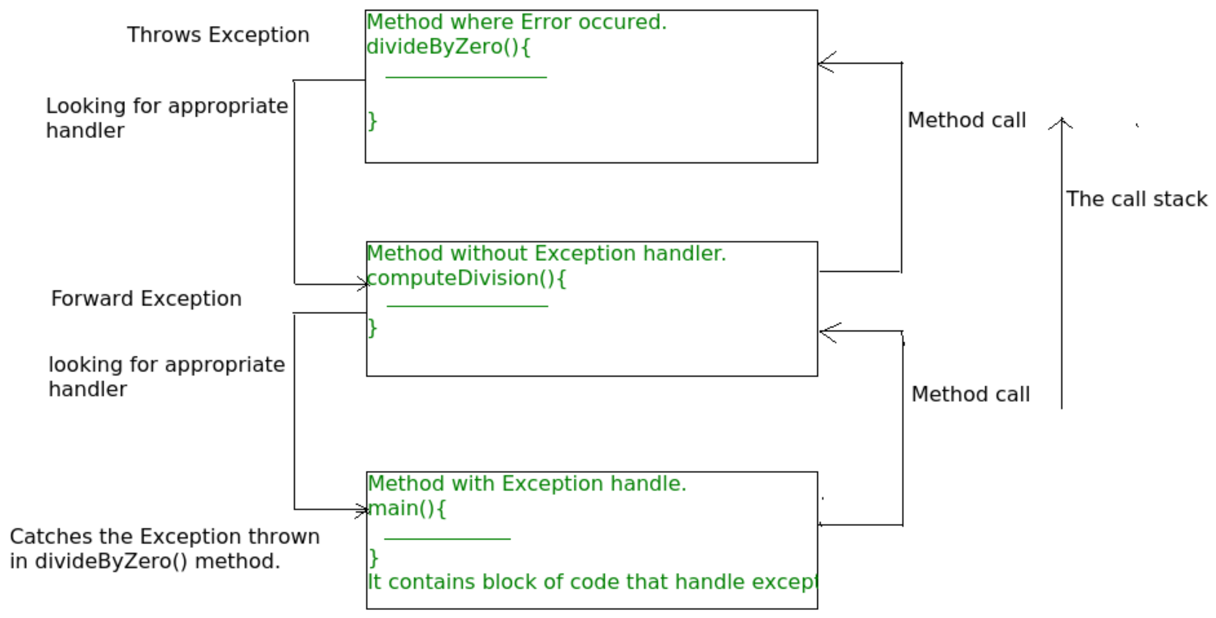
How Does JVM handle an Exception?

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**.

```
Exception in thread "xxx" Name of Exception : Description
... ...... ..  // Call Stack
```

Look at the below diagram to understand the flow of the call stack.

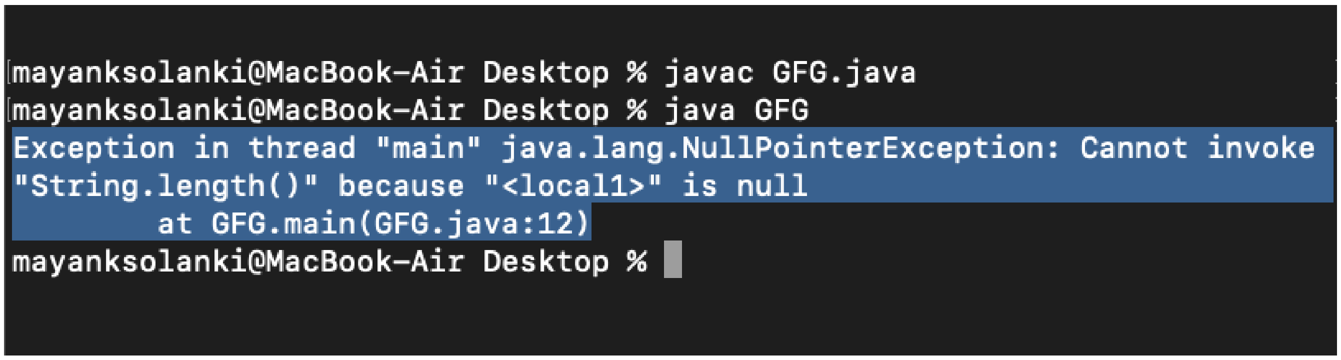

Throws Exception
Method where Error occured.
divideByZero(){
}
Looking for appropriate handler
Method call
The call stack
Method without Exception handler.
computeDivision(){
}
Forward Exception
looking for appropriate handler
Method call
Method with Exception handle.
main(){
}
It contains block of code that handle except
Catches the Exception thrown in divideByZero() method.

The call stack and searching the call stack for exception handler.

Illustration:

```
// 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());
    }
}
```

Output:



```
mayanksolanki@MacBook-Air Desktop % javac GFG.java
mayanksolanki@MacBook-Air Desktop % java GFG
Exception in thread "main" java.lang.NullPointerException: Cannot invoke
"String.length()" because "<local1>" is null
        at GFG.main(GFG.java:12)
mayanksolanki@MacBook-Air Desktop %
```

Let us see an example that illustrates how a run-time system searches for appropriate exception handling code on the call stack.

Example:

```
// Java Program to Demonstrate Exception is Thrown
// How the runTime System Searches Call-Stack
// to Find Appropriate Exception Handler

// Class
// ExceptionThrown
class GFG {

    // Method 1
    // It throws the Exception(ArithmeticException).
    // Appropriate Exception handler is not found
    // within this method.
    static int divideByZero(int a, int b)
    {

        // this statement will cause ArithmeticException
        // (/by zero)
        int i = a / b;

        return i;
    }

    // The runTime System searches the appropriate
    // Exception handler in method also but couldn't have
    // found. So looking forward on the call stack
    static int computeDivision(int a, int b)
    {

        int res = 0;

        // Try block to check for exceptions
        try {

            res = divideByZero(a, b);
        }

        // Catch block to handle NumberFormatException
        // exception Doesn't matches with
        // ArithmeticException
        catch (NumberFormatException ex) {
            // Display message when exception occurs
            System.out.println(
                "NumberFormatException is occurred");
        }
        return res;
    }

    // Method 2
    // Found appropriate Exception handler.
    // i.e. matching catch block.
    public static void main(String args[])
    {
        int a = 1;
        int b = 0;

        // Try block to check for exceptions
        try {
            int i = computeDivision(a, b);
        }

        // Catch block to handle ArithmeticException
        // exceptions
        catch (ArithmeticException ex) {

            // getMessage() will print description
            // of exception(here / by zero)
            System.out.println(ex.getMessage());
        }
    }
}
```

Output

```
/ by zero
```

How Programmer Handles an Exception?

Customized Exception Handling: Java exception handling is managed via five keywords: **try**, **catch**, **throw**, **throws**, and **finally**. Briefly, here is how they work. Program statements that you think can raise exceptions are contained within a try block. If an exception occurs within the try block, it is thrown. Your code can catch this exception (using catch block) and handle it in some rational manner. System-generated exceptions are automatically thrown by the Java run-time system. To manually throw an exception, use the keyword throw. Any exception that is thrown out of a method must be specified as such by a throws clause. Any code that absolutely must be executed after a try block completes is put in a finally block.

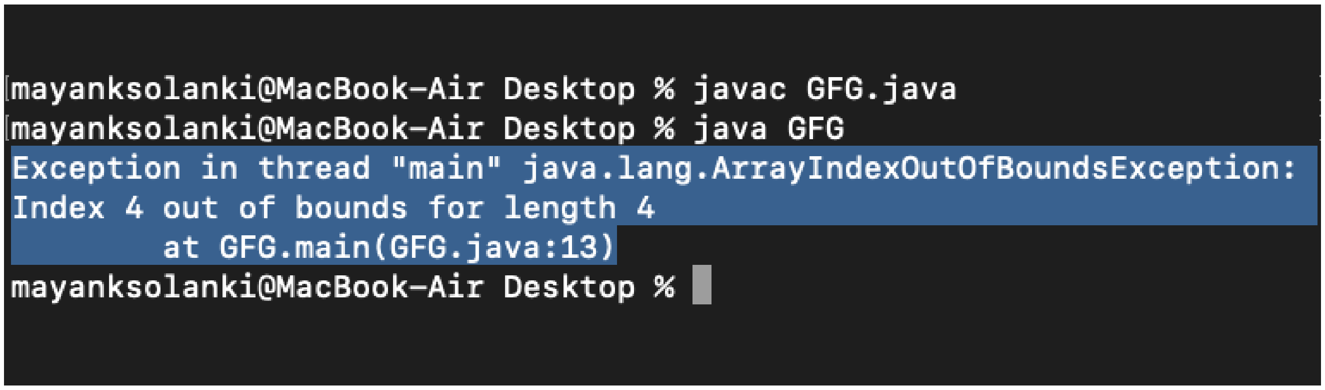
Need for try-catch clause(Customized Exception Handling)

Consider the below program in order to get a better understanding of the try-catch clause.

Example:

```
// Java Program to Demonstrate
// Need of try-catch Clause

// Class
class GFG {

    // Main driver method
    public static void main(String[] args)
    {
        // Taking an array of size 4
        int[] arr = new int[4];

        // Now this statement will cause an exception
        int i = arr[4];

        // This statement will never execute
        // as above we caught with an exception
        System.out.println("Hi, I want to execute");
    }
}
```

Output:



```
mayanksolanki@MacBook-Air Desktop % javac GFG.java
mayanksolanki@MacBook-Air Desktop % java GFG
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException:
Index 4 out of bounds for length 4
        at GFG.main(GFG.java:13)
mayanksolanki@MacBook-Air Desktop %
```

Output explanation: In the above example, an array is defined with size i.e. you can access elements only from index 0 to 3. But you trying to access the elements at index 4(by mistake) that's why it is throwing an exception. In this case, JVM terminates the program **abnormally**. The statement System.out.println("Hi, I want to execute"); will never execute. To execute it, we must handle the exception using try-catch. Hence to continue the normal flow of the program, we need a try-catch clause.

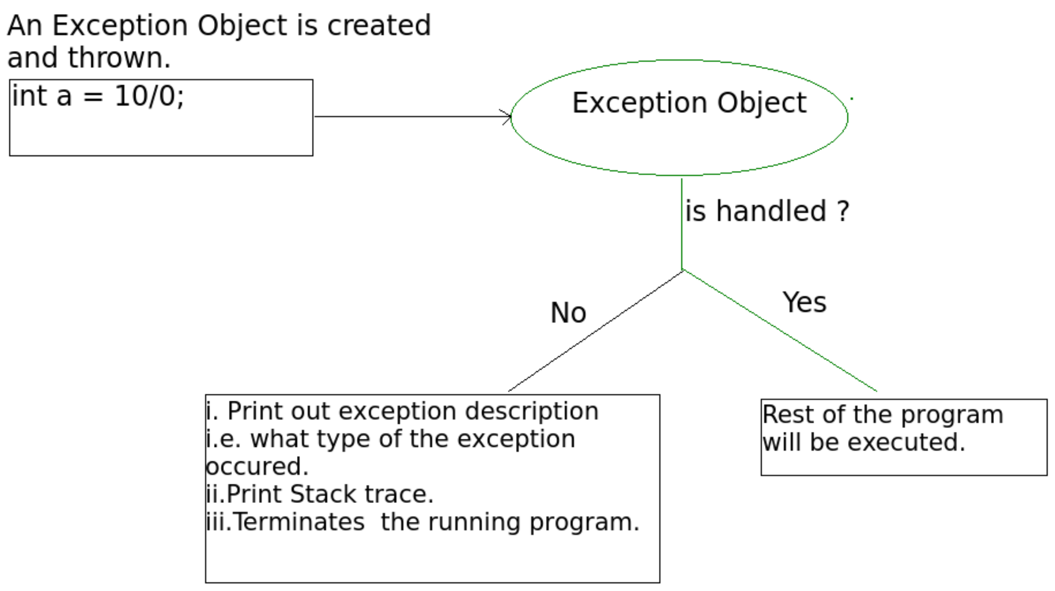
How to Use the try-catch Clause?

```
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
}
```

Certain below key points are needed to be remembered that are as follows:

- In a method, there can be more than one statement that might throw an exception, So put all these statements within their own **try** block and provide a separate exception handler within their own **catch** block for each of them.
- If an exception occurs within the **try** block, that exception is handled by the exception handler associated with it. To associate the exception handler, we must put a **catch** block after it. There can be more than one exception handlers. Each **catch** block is an exception handler that handles the exception of the type indicated by its argument. The argument, ExceptionType declares the type of exception that it can handle and must be the name of the class that inherits from the **Throwable** class.
- For each try block there can be zero or more catch blocks, but **only one** final block.
- The finally block is optional. It always gets executed whether an exception occurred in try block or not. If an exception occurs, then it will be executed after **try and catch blocks.** And if an exception does not occur then it will be executed after the **try** block. The finally block in java is used to put important codes such as clean up code e.g. closing the file or closing the connection.

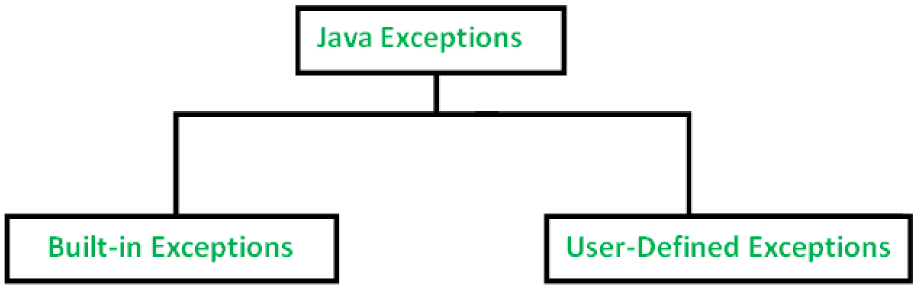
The summary is depicted via visual aid below as follows:


An Exception Object is created and thrown.
int a = 10/0;
Exception Object
is handled ?
No
Yes
i. Print out exception description i.e. what type of the exception occured.
ii.Print Stack trace.
iii.Terminates the running program.
Rest of the program will be executed.

Types of Exception in Java with Examples

- Difficulty Level : Medium
- Last Updated : 16 Jun, 2022

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



Built-in Exceptions:

Built-in exceptions are the exceptions that are available in Java libraries. These exceptions are suitable to explain certain error situations. Below is the list of important built-in exceptions in Java.

1. **ArithmeticException:** It is thrown when an exceptional condition has occurred in an arithmetic operation.

2. **ArrayIndexOutOfBoundsException:** It is thrown to indicate that an array has been accessed with an illegal index. The index is either negative or greater than or equal to the size of the array.
3. **ClassNotFoundException:** This Exception is raised when we try to access a class whose definition is not found
4. **FileNotFoundException:** This Exception is raised when a file is not accessible or does not open.
5. **IOException:** It is thrown when an input-output operation failed or interrupted
6. **InterruptedException:** It is thrown when a thread is waiting, sleeping, or doing some processing, and it is interrupted.
7. **NoSuchFieldException:** It is thrown when a class does not contain the field (or variable) specified
8. **NoSuchMethodException:** It is thrown when accessing a method that is not found.
9. **NullPointerException:** This exception is raised when referring to the members of a null object. Null represents nothing
10. **NumberFormatException:** This exception is raised when a method could not convert a string into a numeric format.
11. **RuntimeException:** This represents an exception that occurs during runtime.
12. **StringIndexOutOfBoundsException:** It is thrown by String class methods to indicate that an index is either negative or greater than the size of the string
13. **IllegalArgumentException :** This exception will throw the error or error statement when the method receives an argument which is not accurately fit to the given relation or condition. It comes under the unchecked exception.
14. **IllegalStateException :** This exception will throw an error or error message when the method is not accessed for the particular operation in the application. It comes under the unchecked exception.

Examples of 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

```
// Java program to demonstrate NoSuchElementException
public class NoSuchElementException_Demo {


    public static void main(String[] args)
    {


        Set exampleleSet = new HashSet();


        Hashtable exampleTable = new Hashtable();


        exampleleSet.iterator().next();
          //accessing Set


        exampleTable.elements().nextElement();
          //accessing Hashtable
          // This throws a NoSuchElementException as there are
        // no elements in Set and HashTable and we are
        // trying to access elements
    }
}
```

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’.

The following steps are followed for the creation of a user-defined Exception.

- The user should create an exception class as a subclass of the Exception class. Since all the exceptions are subclasses of the Exception class, the user should also make his class a subclass of it. This is done as:

```
class MyException extends Exception
```

- We can write a default constructor in his own exception class.

```
MyException(){}
```

- We can also create a parameterized constructor with a string as a parameter.
  We can use this to store exception details. We can call the superclass(Exception) constructor from this and send the string there.

```
MyException(String str)
{
   super(str);
}
```

- To raise an exception of a user-defined type, we need to create an object to his exception class and throw it using the throw clause, as:

```
MyException me = new MyException("Exception details");
throw me;
```

- The following program illustrates how to create your own exception class MyException.
- Details of account numbers, customer names, and balance amounts are taken in the form of three arrays.
- In main() method, the details are displayed using a for-loop. At this time, a check is done if in any account the balance amount is less than the minimum balance amount to be apt in the account.
- If it is so, then MyException is raised and a message is displayed "Balance amount is less".

Example

```
// Java program to demonstrate user defined exception


// This program throws an exception whenever balance

// amount is below Rs 1000

class MyException extends Exception

{

    //store account information

    private static int accno[] = {1001, 1002, 1003, 1004};


    private static String name[] =

                 {"Nish", "Shubh", "Sush", "Abhi", "Akash"};


    private static double bal[] =

         {10000.00, 12000.00, 5600.0, 999.00, 1100.55};


    // default constructor

    MyException() {    }


    // parameterized constructor

    MyException(String str) { super(str); }


    // write main()

    public static void main(String[] args)

    {

        try {
            // display the heading for the table
            System.out.println("ACCNO" + "\t" + "CUSTOMER" +
                                                "\t" + "BALANCE");

            // display the actual account information
            for (int i = 0; i < 5 ; i++)
            {
                System.out.println(accno[i] + "\t" + name[i] +
                                                    "\t" + bal[i]);

                // display own exception if balance < 1000
                if (bal[i] < 1000)
                {
                    MyException me =
                       new MyException("Balance is less than 1000");
                    throw me;
                }
            }
        } //end of try

        catch (MyException e) {
            e.printStackTrace();
        }
    }
}
```

Runtime Error

```
 MyException: Balance is less than 1000
    at MyException.main(fileProperty.java:36)
```

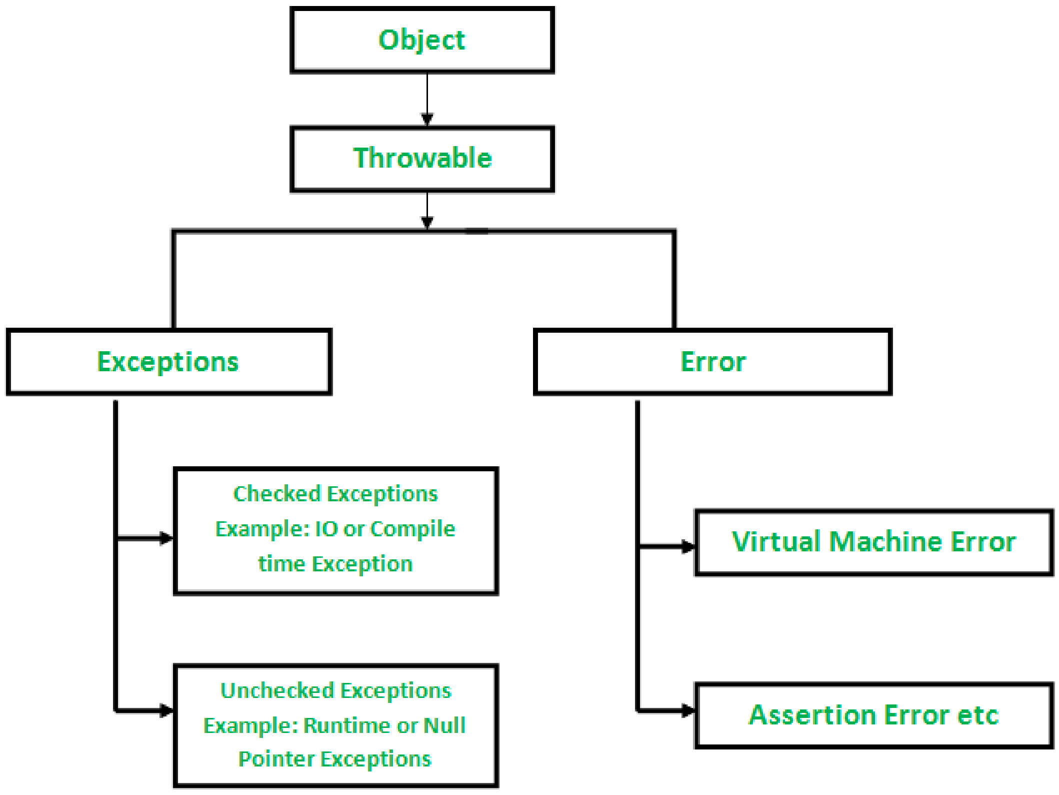
Output:

```
ACCNO    CUSTOMER    BALANCE
1001    Nish    10000.0
1002    Shubh    12000.0
1003    Sush    5600.0
1004    Abhi    999.0
```

Checked vs Unchecked Exceptions in Java

An 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. In Java, there are two types of exceptions:

1. Checked exceptions
2. Unchecked exceptions


Object
Throwable
Exceptions
Error
Checked Exceptions Example: IO or Compile time Exception
Unchecked Exceptions Example: Runtime or Null Pointer Exceptions
Virtual Machine Error
Assertion Error etc

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.

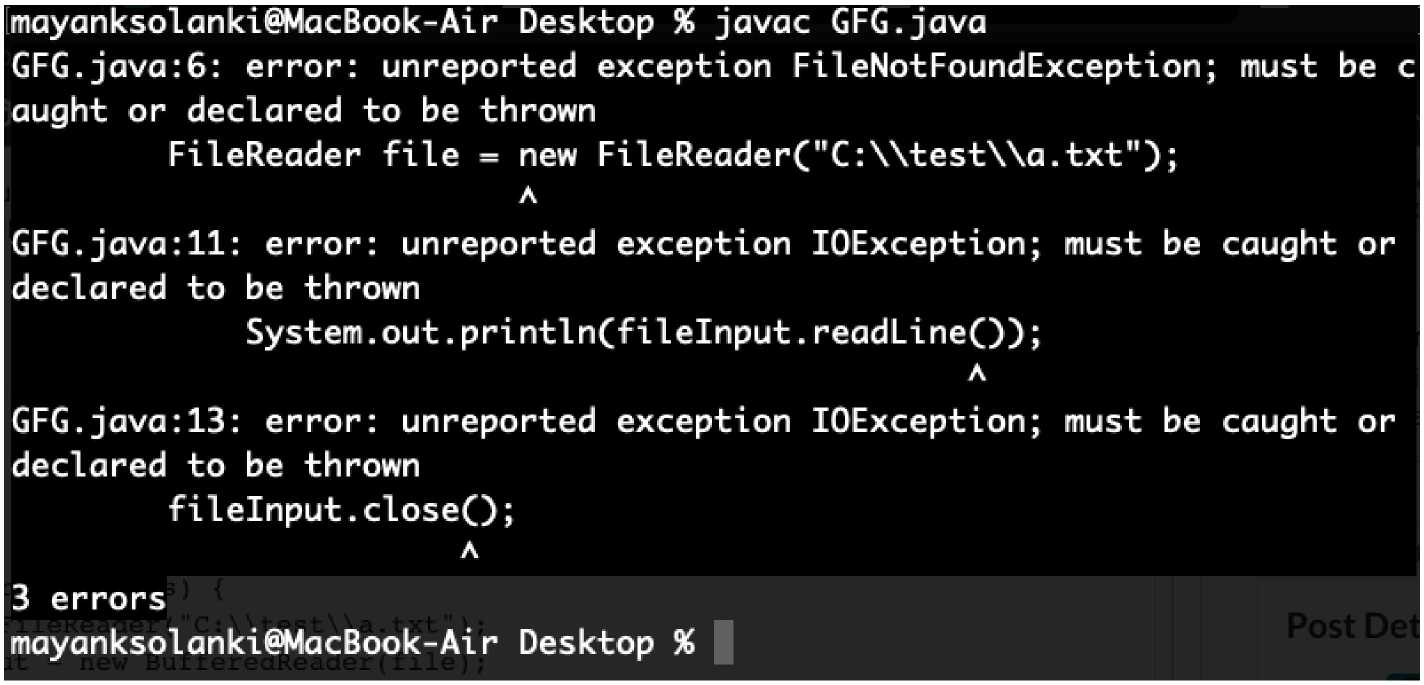
For example, consider the following Java program that opens the file at location "C:\test\a.txt" and prints the first three lines of it. The program doesn't compile, because the function main() uses FileReader() and FileReader() throws a checked exception *FileNotFoundException*. It also uses readLine()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

Example:

```
// 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:



```
mayanksolanki@MacBook-Air Desktop % javac GFG.java
GFG.java:6: error: unreported exception FileNotFoundException; must be c
aught or declared to be thrown
        FileReader file = new FileReader("C:\\test\\a.txt");
                          ^
GFG.java:11: error: unreported exception IOException; must be caught or
declared to be thrown
            System.out.println(fileInput.readLine());
                                                 ^
GFG.java:13: error: unreported exception IOException; must be caught or
declared to be thrown
        fileInput.close();
                       ^
3 errors
mayanksolanki@MacBook-Air Desktop %
```

To fix the above program, we either need to specify a list of exceptions using throws, or we need to use a try-catch block. We have used throws in the below program. Since *FileNotFoundException* is a subclass of *IOException*, we can just specify *IOException* in the throws list and make the above program compiler-error-free.

Example:

```
// 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.

Example:

```
// 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
```

Try, catch, throw and throws in Java

What is an Exception?

An exception is an “unwanted or unexpected event”, which occurs during the execution of the program i.e, at run-time, that disrupts the normal flow of the program’s instructions. When an exception occurs, the execution of the program gets terminated.

Why does an Exception occur?

An exception can occur due to several reasons like a Network connection problem, Bad input provided by a user, Opening a non-existing file in your program, etc

Blocks & Keywords used for exception handling

1. **try**: The try block contains a set of statements where an exception can occur.

```
try
{
    // statement(s) that might cause exception
}
```

2. **catch**: The catch block is used to handle the uncertain condition of a try block. A try block is always followed by a catch block, which handles the exception that occurs in the associated try block.

```
catch
{
   // statement(s) that handle an exception
   // examples, closing a connection, closing
   // file, exiting the process after writing
   // details to a log file.
}
```

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

4. **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.

5. **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.

Example of an exception generated by the system is given below :

```
Exception in thread "main"
java.lang.ArithmeticException: divide
by zero at ExceptionDemo.main(ExceptionDemo.java:5)
ExceptionDemo: The class name
main:The method name
ExceptionDemo.java:The file name
java:5:line number
// Java program to demonstrate working of try,

// catch and finally



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
```

An example of throws keyword:

```
// Java program to demonstrate working of throws

class ThrowsExecp {


    // This method throws an exception

    // to be handled

    // by caller or caller

    // of caller and so on.

    static void fun() throws IllegalAccessException

    {

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

        throw new IllegalAccessException("demo");

    }


    // This is a caller function

    public static void main(String args[])

    {

        try {

            fun();

        }

        catch (IllegalAccessException e) {

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

        }

    }

}
```

Output:

```
Inside fun().
caught in main.
```

Flow control in try catch finally in Java

In this article, we'll explore all the possible combinations of try-catch-finally which may happen whenever an exception is raised and how the control flow occurs in each of the given cases.

1. **Control flow in try-catch clause OR try-catch-finally clause**
   - **Case 1:** Exception occurs in try block and handled in catch block
   - **Case 2:** Exception occurs in try-block is not handled in catch block
   - **Case 3:** Exception doesn't occur in try-block
2. **try-finally clause**
   - **Case 1:** Exception occurs in try block
   - **Case 2:** Exception doesn't occur in try-block

Control flow in try-catch OR try-catch-finally

1. Exception occurs in try block and handled in catch block: If a statement in try block raised an exception, then the rest of the try block doesn't execute and control passes to the **corresponding** catch block. After executing the catch block, the control will be transferred to finally block(if present) and then the rest program will be executed.

- **Control flow in try-catch:**

```
// Java program to demonstrate

// control flow of try-catch clause

// when exception occur in try block

// and handled in catch block

class GFG

{

    public static void main (String[] args)

    {



        // array of size 4.

        int[] arr = new int[4];

        try

        {

            int i = arr[4];



            // this statement will never execute

            // as exception is raised by above statement

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

        }

        catch(ArrayIndexOutOfBoundsException ex)

        {
            System.out.println("Exception caught in Catch block");
        }


        // rest program will be executed
        System.out.println("Outside try-catch clause");
    }
}
```

Output:

```
Exception caught in Catch block
Outside try-catch clause
```

- **Control flow in try-catch-finally clause :**

```
// Java program to demonstrate
// control flow of try-catch-finally clause
// when exception occur in try block
// and handled in catch block
class GFG
{
    public static void main (String[] args)
    {


        // array of size 4.
        int[] arr = new int[4];


        try
        {
            int i = arr[4];


            // this statement will never execute
            // as exception is raised by above statement
            System.out.println("Inside try block");
        }


        catch(ArrayIndexOutOfBoundsException ex)
        {
            System.out.println("Exception caught in catch block");
        }

        finally
        {
            System.out.println("finally block executed");
        }

        // rest program will be executed
        System.out.println("Outside try-catch-finally clause");
    }
}
```

Output:

```
Exception caught in catch block
finally block executed
Outside try-catch-finally clause
```

2. Exception occurred in try-block is not handled in catch block: In this case, default handling mechanism is followed. If finally block is present, it will be executed followed by default handling mechanism.

- **try-catch clause :**

```
// Java program to demonstrate
// control flow of try-catch clause
// when exception occurs in try block
// but not handled in catch block
class GFG
{
    public static void main (String[] args)
    {

        // array of size 4.
        int[] arr = new int[4];
        try
        {
            int i = arr[4];

            // this statement will never execute
            // as exception is raised by above statement
            System.out.println("Inside try block");
        }

        // not a appropriate handler
        catch(NullPointerException ex)
        {
            System.out.println("Exception has been caught");
        }

        // rest program will not execute
        System.out.println("Outside try-catch clause");
    }
}
```

Run Time Error:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 4
    at GFG.main(GFG.java:12)
```

- **try-catch-finally clause :**

```
// Java program to demonstrate
// control flow of try-catch-finally clause
// when exception occur in try block
// but not handled in catch block
class GFG
{
    public static void main (String[] args)
    {

        // array of size 4.
        int[] arr = new int[4];
        try
        {
            int i = arr[4];

            // this statement will never execute
            // as exception is raised by above statement
            System.out.println("Inside try block");
        }

        // not a appropriate handler
        catch(NullPointerException ex)
        {
            System.out.println("Exception has been caught");
        }

        finally
        {
            System.out.println("finally block executed");
        }

        // rest program will not execute
        System.out.println("Outside try-catch-finally clause");
    }
}
```

Output :

```
finally block executed
```

Run Time error:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 4
    at GFG.main(GFG.java:12)
```

3. Exception doesn't occur in try-block: In this case catch block never runs as they are only meant to be run when an exception occurs. finally block(if present) will be executed followed by rest of the program.

- **try-catch clause :**

```
// Java program to demonstrate try-catch
// when an exception doesn't occurred in try block
class GFG
{
    public static void main (String[] args)
    {

        try
        {

            String str = "123";

            int num = Integer.parseInt(str);

            // this statement will execute
            // as no any exception is raised by above statement
            System.out.println("Inside try block");

        }

        catch(NumberFormatException ex)
        {

            System.out.println("catch block executed...");

        }

        System.out.println("Outside try-catch clause");
    }
}
```

Output :

```
Inside try block
Outside try-catch clause
```

- **try-catch-finally clause**

```
// Java program to demonstrate try-catch-finally
// when exception doesn't occurred in try block
class GFG
{
    public static void main (String[] args)
    {

    try
    {

        String str = "123";

        int num = Integer.parseInt(str);

        // this statement will execute
        // as no any exception is raised by above statement
        System.out.println("try block fully executed");

    }

    catch(NumberFormatException ex)
    {

        System.out.println("catch block executed...");

    }

    finally
    {
        System.out.println("finally block executed");
    }
    System.out.println("Outside try-catch-finally clause");

    }

}
```

Output :

```
try block fully executed
finally block executed
Outside try-catch clause
```

Control flow in try-finally

In this case, no matter whether an exception occur in try-block or not, **finally will always be executed.** But control flow will depend on whether exception has occurred in try block or not.

1. Exception raised: If an exception has occurred in try block then control flow will be finally block followed by default exception handling mechanism.

```
// Java program to demonstrate

// control flow of try-finally clause

// when exception occur in try block

class GFG

{

    public static void main (String[] args)

    {


        // array of size 4.

        int[] arr = new int[4];

        try

        {

            int i = arr[4];


            // this statement will never execute

            // as exception is raised by above statement

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

        }


        finally

        {
            System.out.println("finally block executed");
        }


        // rest program will not execute
        System.out.println("Outside try-finally clause");
    }
}
```

Output :

```
finally block executed
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 4
    at GFG.main(GFG.java:11)
```

2. Exception not raised: If an exception does not occur in try block then control flow will be finally block followed by rest of the program

```
// Java program to demonstrate
// control flow of try-finally clause
// when exception doesn't occur in try block
class GFG
{
    public static void main (String[] args)
    {


        try
        {
            String str = "123";


            int num = Integer.parseInt(str);


            // this statement will execute
            // as no any exception is raised by above statement
            System.out.println("Inside try block");
        }


        finally
        {
            System.out.println("finally block executed");

        }



        // rest program will be executed

        System.out.println("Outside try-finally clause");

    }

}
```

Output :

```
Inside try block
finally block executed
Outside try-finally clause
```

throw and throws in Java

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("/ 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. 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.

```
// Java program that demonstrates the use of throw

class ThrowExcep

{

    static void fun()

    {

        try

        {
            throw new NullPointerException("demo");
        }
        catch(NullPointerException e)
        {
            System.out.println("Caught inside fun().");
            throw e; // rethrowing the exception
        }
    }

    public static void main(String args[])
    {
        try
        {
            fun();
        }
        catch(NullPointerException e)
        {
            System.out.println("Caught in main.");
        }
    }
}
```

Output:

```
Caught inside fun().
Caught in main.
```

Another Example:

```
// Java program that demonstrates the use of throw
class Test
{
    public static void main(String[] args)
    {
        System.out.println(1/0);
    }
}
```

Output:

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

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.

Syntax:

```
type method_name(parameters) throws exception_list
exception_list is a comma separated list of all the
exceptions which a method might throw.
```

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

We can use throws keyword to delegate the responsibility of exception handling to the caller (It may be a method or JVM) then caller method is responsible to handle that exception.

```
// Java program to illustrate error in case

// of unhandled exception

class tst

{

    public static void main(String[] args)

    {

        Thread.sleep(10000);

        System.out.println("Hello Geeks");

    }

}
```

Output:

```
error: unreported exception InterruptedException; must be caught or declared to be
thrown
```

Explanation: In the above program, we are getting compile time error because there is a chance of exception if the main thread is going to sleep, other threads get the chance to execute main() method which will cause InterruptedException.

```
// Java program to illustrate throws

class tst

{

    public static void main(String[] args)throws InterruptedException

    {

        Thread.sleep(10000);

        System.out.println("Hello Geeks");

    }

}
```

Output:

```
Hello Geeks
```

Explanation: In the above program, by using throws keyword we handled the InterruptedException and we will get the output as **Hello Geeks**

Another Example:

```
// 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.

User-defined Custom Exception in Java

An exception is an issue (run time error) that occurred during the execution of a program. When an exception occurred the program gets terminated abruptly and, the code past the line that generated the exception never gets executed.

Java provides us the facility to create our own exceptions which are basically derived classes of Exception. Creating our own Exception is known as a custom exception or user-defined exception. Basically, Java custom exceptions are used to customize the exception according to user needs. In simple words, we can say that a User-Defined Exception or custom exception is creating your own exception class and throwing that exception using the 'throw' keyword.

For example, MyException in the below code extends the Exception class.

Why use custom exceptions?

Java exceptions cover almost all the general types of exceptions that may occur in the programming. However, we sometimes need to create custom exceptions.

Following are a few of the reasons to use custom exceptions:

- To catch and provide specific treatment to a subset of existing Java exceptions.
- Business logic exceptions: These are the exceptions related to business logic and workflow. It is useful for the application users or the developers to understand the exact problem.

In order to create a custom exception, we need to extend the Exception class that belongs to **java.lang package.**

Example: We pass the string to the constructor of the superclass- Exception which is obtained using the "getMessage()" function on the object created.

```
// A Class that represents use-defined exception


class MyException extends Exception {
    public MyException(String s)
    {
        // Call constructor of parent Exception
        super(s);
    }
}


// A Class that uses above MyException
public class Main {
    // Driver Program
    public static void main(String args[])
    {
        try {
            // Throw an object of user defined exception
            throw new MyException("NM-AIST");
        }
        catch (MyException ex) {
            System.out.println("Caught");

            // Print the message from MyException object
            System.out.println(ex.getMessage());
        }
    }
}
```

Output

```
Caught
NM-AIST
```

In the above code, the constructor of MyException requires a string as its argument. The string is passed to the parent class Exception's constructor using super(). The constructor of the Exception class can also be called without a parameter and the call to super is not mandatory.

```
 // A Class that represents use-defined exception


class MyException extends Exception {
}
```

```
// A Class that uses above MyException

public class setText {

    // Driver Program

    public static void main(String args[])

    {

        try {

            // Throw an object of user defined exception

            throw new MyException();

        }

        catch (MyException ex) {

            System.out.println("Caught");

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

        }

    }

}
```

Output

```
Caught
null
```