Exception Handling

Exception means an abnormal condition, Rare Case, in Hindi “**Apwad**”.

Exception is an error event that can happen during the execution of a program and disrupts its normal flow. Exception can arise from different kind of situations such as wrong data entered by user, hardware failure, network connection failure etc.

Exception is mainly came to avoid run-time error. **Runtime Errors** are more dangerous than other errors because it comes at run-time. If the client gets this error and program terminate then it sends bad impression on the company. We don't know how different a user will use the program and what exceptions can come. So to maintain the program flow "exception" concept came. It handles the situation and prevents program termination.

The exception handling in Java is one of the powerful mechanism to handle the runtime errors such as such as (ClassNotFound, IO, SQL, Remote etc.) so that normal flow of the application can be maintained.

Difference between Exception and Error?

A condition that can handle is called Exception and the condition that can't handle become an error.

If we take real word example, suppose a biker is riding with speed 120 km per hour in street then in this case accident chances are higher and this is a normal condition. Because streets are also for walking.

But suppose a biker riding a bike at the highway with 120 km PH and somehow a child comes to the road then this will call abnormal condition because no one expects a child at the highway.

If somehow a biker saves the child then this will call an exception. If biker does not save then it will be called error.

In Java 10/0 is the exception that compiler handles and don't terminate the flow of the program. Even java rarely terminates the program that's why we call java a robust programing language.

Errors are divided into three type.

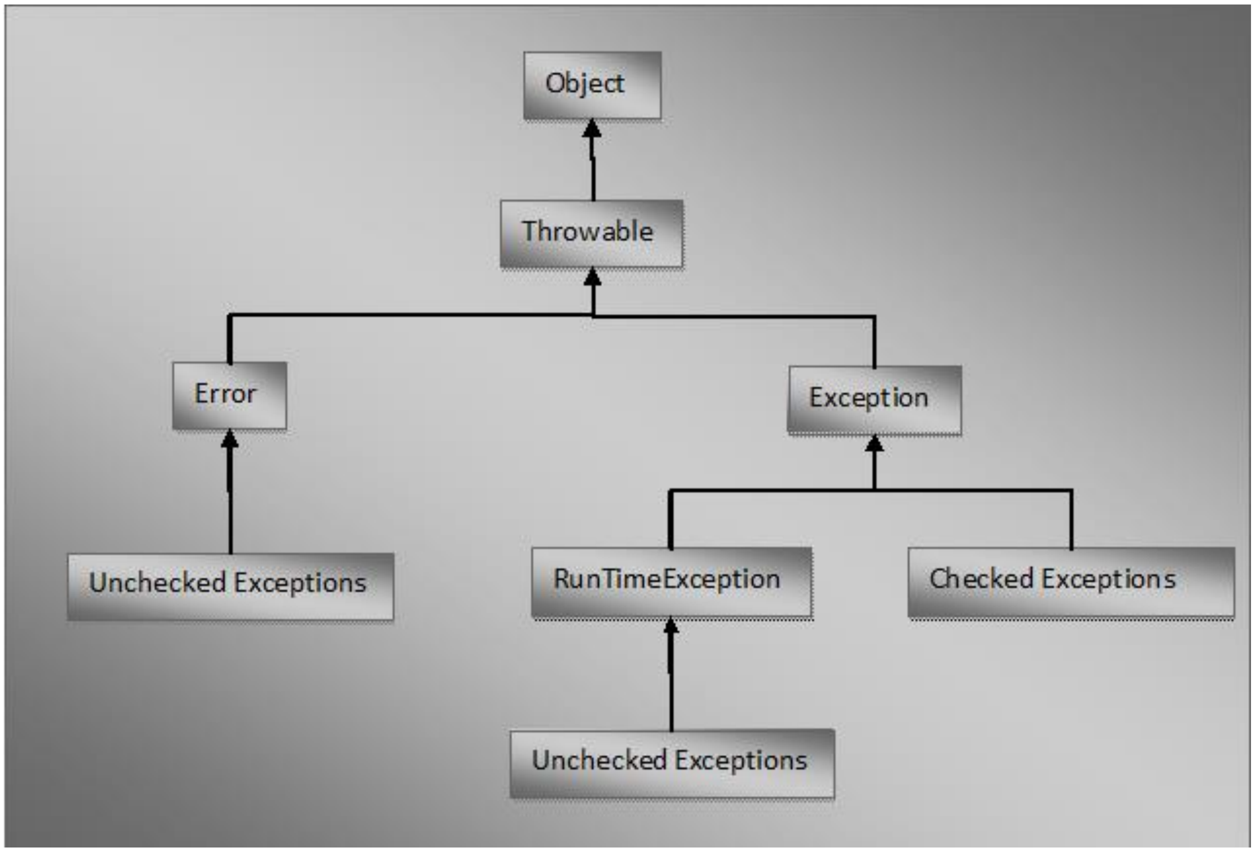
1. Syntax Error / Compilation Error
2. Logical Error
3. Runtime Error

# **Logical errors** are that does not fulfill the requirement. Suppose everything is working fine, there is no error in the program, but it does not fulfill the client requirement then it will be called Logical Error.

#**Runtime Errors** are more dangerous than other errors because it comes at run-time. If the client gets this error and program terminate then it sends bad impression on the company.

To avoid runtime error testing module came where we test the program again and again in different situations. But still, we don't know how different a user will use the program and what exceptions can come.

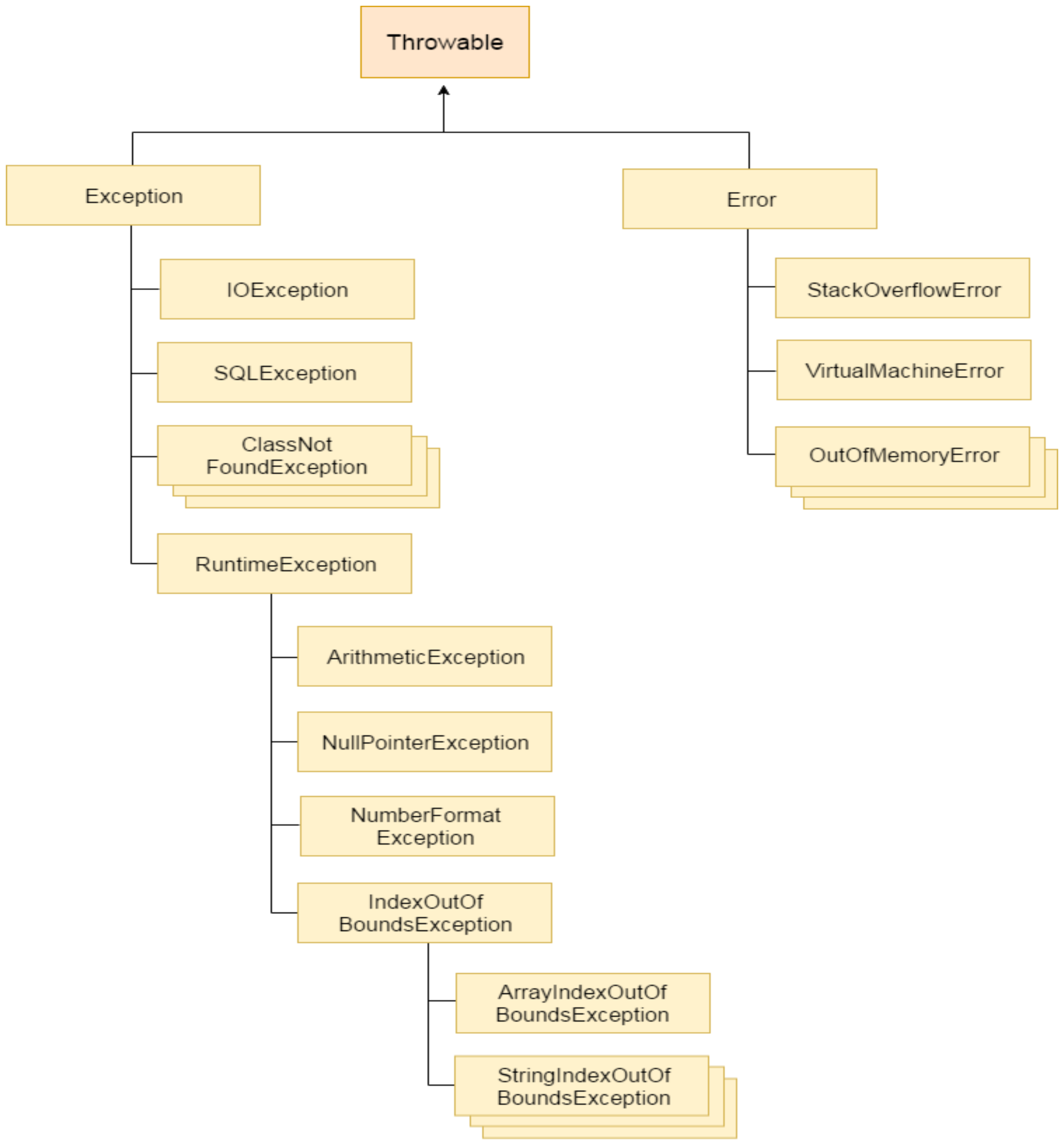
So to maintain the program flow "exception" concept came. It handles the situation and prevents program termination.



Exception Class Hierarchy

#. More than 250 predefined exception classes are available in java.

Exception classes inherit other exception classes because they re-use common features. Parent class is "**Throwable**". There are two classes in parallel "**Exception**" and "**Error**".



#. Types of Exception

There are mainly two types of exceptions: “**checked**” and “**unchecked**” where the **error** is considered as an unchecked exception. The sun microsystem says there are three types of exceptions.

1. Checked Exception
2. Unchecked Exception
3. Error

Checked Exceptions are known to the compiler but unchecked is not known to the compiler.

In case of a checked exception, **compiler forces the programmer** to put the handler but in case of unchecked exceptions **compiler does not force** to put the handler.

How to know if any exception is checked or unchecked?

#. Java programs run inside JVM boundary and when the program goes outside the JVM boundary like in case of Database, File or Networking then compiler forces to put handler.

#. If the program runs inside JVM boundary then it won't force to put handler but if we don't add handler then the program will terminate. That’s why it is recommended to put that cade inside try block that can occur exception.

Try-Catch-Finally

#. All those statements and function which can generate some kind of an exception at runtime must be written in a particular block called try block, otherwise, we won't be able to catch the exception or we can say we won't know the reason of abnormal condition and program will be terminate.

#. If we want to catch the exception then immediately we need to add catch block.

We have to predict which exception can occur so the same exception class will be defined inside catch parentheses. But it is hard to remember all exception classes. So to solve this issue we can define “Exception” class that will catch exception object. As we know parent class can hold the child reference id, the same happening here.

If an exception occurs then all lines of try block will be skipped and immediately catch block will run after that remaining program will run. The most important thing is program will not terminate.

What Task should be done in catch block?

- Print appropriate message.
- Call any function according to requirement but never write those kind of statement that can further generate exception.
- Close all those resources which have opened in a try block.

Rule: "**Try**" and "**catch**" are made for each other however catch can't be without try but try is allowed with finally block.

Rule: "**try catch**" block can be define inside any block except class block.

Rule: We can have more than one "**catch**" block with a single "**try**" block. However, if we are having more than one catch block with a single try then at a time only one catch block will be executed. We also need to set catch block from child to parent class order. It will be compile error if we don't set catch block in child to parent order.

If we are having more than one catch block with single try block then they must be in child to parent order. Because if we set Exception class first and other child classes later then all exception will be caught at Exception class and other catch block will never run.

In a single try, Java never occurs more than one exception because at first exception "try" block will skip all codes and only one catch block will execute.

#Note: We can put try catch block inside catch block we should not do this because we should avoid writing those kind of codes inside catch block that can further generate any exception.

#Note: We can also do different task on different exception in single exception.

Program...

Instance of is also another way to do different-different task in single exception.

Rule: We can also have more than one exception class in a single catch block. If we are having more than one exception class name in a single catch block then those exception classes must be parallel to each other.

Rule: We can't write and statements between try and catch block, only spaces are allowed.

#Note: If in try block first line occurs exception/error then all code will be skip. Just because of one error all work will be skip. That's why we should not put all works inside a single try block. We should make different-different try block for different-different tasks.

“If we understand this with a real word example: Ek ladke ne murder Kar Diya police wale ko pata chal gya. Police wale Ghar gye puchh tachh kiya, koi nahi bataya ladka kha hai kyunki kisi ko pata hi mahi Tha. To police wale need sabko peet Diya aur jail me daal Diya.

Fir police wale ladke ke Nani ke Ghar gye aur waha bhi puchh tachh ki. Koi nahi bataya to sabko pakad ke peet Diya.

Ye to galat hua na ek ki wajah se sabko peet Diya. That's why to perform different task we should make different try catch block. It is also good practice to put simmilar type of task code inside try block “.

#. It is recommended to write the statement inside try block that may occur same exception. We should not take the risk.

If an exception occurs then it never go back to try block implicitly.

" Jese Mithai wala apni mithai khud nahi khata,
Wese hi programmer apna software khud istemal Nahi karat ".

Program

```
// If we having more that one catch block with a single try
block them only a one catch block will execute.

class ExceptionDemo{

    //Can't write try catch in class block

    public static void main(String...a){
        // Tutorial 1 - (Separate Catch for Each Exception)
        //Checked Exception: Compiler demands to add handler
        try{

            //ArithmeticException
// Exception - It will jump to the ArithmeticException
            int x = 10/0;

//It won't jump to the ArrayIndexOutOfBoundsException because
exception already occurred and try program is already skipped.
So Only ArithmeticException Catch block will execute and program
will continue

          int y[] = new int[5];
          y[6] = 10;

            System.out.println("I Won't Execute");

            //close

        }
        catch(ArithmeticException e){
            System.out.println("Please Contact Developer");
            System.out.println(e);
            //Close the resource
        }
        catch(ArrayIndexOutOfBoundsException ie){
            System.out.println("Inside
ArrayIndexOutOfBoundsException");
            System.out.println(ie);
            System.out.println(ie.getMessage());
        }

        System.out.println("Program is Continue 1");


        /*
```

```
If we are having more than one catch block in a single try block
then they must be in child to parent order
            Because If the Exception (Parent class) is in first
catch block then
            It will give error that exception has been caught
and we won't be able to perform task especially for some
Exception
        */

        try {
            String name = null;
            System.out.println(name.length());
        }catch(Exception e){
            System.out.println("Inside Exception Catch: " + e);
        }/*catch(ArithmeticException ae){
            System.out.println("
     If StringIndexOutOfBoundsException Comes Do this " + ae);
        }*/

        System.out.println("Program Is Continue 2");


    // Tutorial 2 - (Single Catch for Multiple Exception )
            try {
                //ArithmeticException
                int x = 10/0;

                // NullPointerException
                String name = null;
                System.out.println(name.length());

                //ArrayIndexOutOfBoundsException
                int y[] = new int[5];
                y[6] = 10;

        }catch(Exception e){
            System.out.println("Inside Exception Catch: " + e);
        }

        System.out.println("Program Is Continue 3");


// Tutorial 3 (We can do different task on different exception
in single Exception)

            try {
                //ArithmeticException
                int x = 10/0;

                // NullPointerException
                String name = null;
                System.out.println(name.length());

                //ArrayIndexOutOfBoundsException
                int y[] = new int[5];
                y[6] = 10;

        }catch(ArithmeticException |
ArrayIndexOutOfBoundsException | StringIndexOutOfBoundsException
e){
            System.out.println("javaJavaJava " + e);
        }catch(Exception e){
          if(e instanceof ArithmeticException){
                System.out.println("Inside ArithmeticException:
" + e);
            }
          else if(e instanceof ArrayIndexOutOfBoundsException){
                System.out.println("Inside
ArrayIndexOutOfBoundsException: " + e);
            }
          else if(e instanceof NullPointerException){
                System.out.println("Inside NullPointerException:
" + e);
            }
        }

        System.out.println("Program Is Continue 4");

    }
}
```

Finally Block

"**finally**" block is always executed.

Even if an error comes then try will skip but definitely finally will run and then program will terminate. If exception does not comes then try will successfully run then finally block will run.

Finally block is never used to catch any exception. Inside finally block we write those kind of statements that we must want to execute, Like database

back up command, closing the opened resources etcetera.
Now we can also have “**try**” block with only “**finally**” block without catch block.

Rule: We can only have one finally block with single try block, we can't have more than one finally block.

Rule: If we are having finally block with try and catch block then it must be the last block.

Tips: Every resources that opens must be closed.

Program:

```
/* Finally Block Always Executes.............. Except in Only Case if
before finally block has System.exit(0);   */

class FinallyBlock{

    public static void main(String...a){

/* If there is no exception, try will executes and then finally will
execute. Program will continue.....*/

      try{
            System.out.println("Try Condition 1 ");
        }catch(Exception e){
            System.out.println("Catch Condition 1: " + e);
        }finally{
            System.out.println("Finally Condition 1: ");
        }
        System.out.println("");
        System.out.println("");

/* If Exception came and Caught then Try will skip rest of the code.
Catch and then Finally Will execute. Program will continue.....*/
        try{
            int x=10/a.length;
            System.out.println("Try Condition 2 ");
        }catch(Exception e){
            System.out.println("Catch Condition 2: " + e);
        }finally{
            System.out.println("Finally Condition 2: ");
        }

/* If Exception comes and does not catch then that will become Error.
Try will skip rest of the code. finally will execute. Program will
terminate. */
```

```
        try{
            int x= 10/0; // creating error
            System.out.println("Try Condition 3 ");
        }catch(NullPointerException e){
            System.out.println("Catch Condition 3: " + e);
        }finally{
            System.out.println("Finally Condition 3: ");
        }

        System.out.println("This Line Won't Execute");
    }
}
```

Need of finally:

```
class NeedOfFinally{

    public static void main(String...a){

        // Case 1
        try {
            boolean con;
            con = true;
            System.out.println("Case 1 Inserting");
            System.out.println("Case 1 Updating");
            System.out.println("Case 1 Fetching");

/* But Somehow Error Comes and Try Will Skip the Rest Code
   Backup task won't perform and resource will be opened. */

/* Make Database Backup (Most Important - Required To Perform)
   // Close Connection
           con false;
        }catch(Exception e){

// somehow catch also did not run
 // Make Database Backup (Most Important - Required To Perform)
   // Close Connection
            con = false;
            System.out.println("Case 1 Catch");
        }finally{
            // finally will run in every case except System.exit(0 OR
1) statement is not before finally
        }
    }
}
```

Program when finally does not executes.

Only in one case finally does not executes, if there is **System.exit(0);** before finally block.

```
try{

    System.exit(0);
}catch(Exception e){
    System.out.println(e);
}finally{
    System.out.println("Finally block will not run");
}
```

What is the Difference between System.exit(0) and System.exit(1 or other values)?

“0” means normal termination. In this case all the opened resources will be close then program will terminate.

“1 other integer value” means abnormal termination. In this case program will terminate without closing the opened resources.

In case abnormal condition there are chances to corrupt data that’s why it is not recommend to terminate the program with other values.

Real word example: Turn of the all home’s electronic devices the switch off the main switch, it is the normal condition. But if we directly switch off the main switch then it will be called abnormal condition.

Automatic Resources Handling

From JDK 1.7 we don't need to close the resources explicitly, the JVM closed the all resources automatically.

From JDK 9 we have parentheses after try where we can write syntax to open resources and JVM will automatically close all those resources when cursor reach at end of try block. This process is known as “**try-with-resources**”.

Rule: However, not all the classes can have object inside try parentheses. Only the objects of those classes can be created within a parentheses of try block which are implementing "**AutoCloseable**" interface.

Rule: Whatever resources we create within a parentheses of a try block that has to be local. If we define Reference variable outside the try resource then connection will be close but object's Reference Id will be available in the program.

The declaration statement appears within parentheses immediately after the try keyword. The class **BufferedReader** in Java SE 7 and later implements the interface "**java.lang.AutoCloseable**". It is the BufferedReader instant is declared in a try with resources. It will be closed regardless of whether the try statement completes normally or abruptly as a result of the method B**ufferReader.readLine()** throwing an IOException.

We can also use multiple resources inside **try-with-resource** block and have them all automatically close here is an example.

This example creates two resources inside the parentheses after the **try** keyword. An **BufferedInputStream** and a **FileInputStream**. Both of these resources will be closed automatically when execution leaves the **try** block.

These resources will be closed in reverse order of the order in which they are created/listed inside the parenthesis. First the **BufferedInputStream** will be closed then the **FileInputSpring** will be closed.

```
class ARE{

    private static void printFileJava7() throws IOException{
        try(FileInputStream input = new
FileInputStream("abc.txt"); BufferedInputStream fin = new
BufferedInputStream(input)){
            int data = input.read();
            while(data != -1){
                System.out.println((char) data);
                data = input.read();
            }
        }
    }

    public static void main(String...a){

    }
}
```

Can we create our own close-able class?

Yes we can make our own close-able class. All we need to do is implement the “**AutoCloseable**” interface and override the close() method.

```
class ClassOne implements AutoCloseable{
    void show(){
        System.out.println("Show From ClassOne");
    }

/* Overriding the close method that inside AutoCloseable interface */
    public void close(){
        System.out.println("Close From ClassOne");
    }
}

class ClassTwo implements AutoCloseable{
    void display(){
        System.out.println("Display From ClassTwo");
    }

/* Overriding the close method that inside AutoCloseable interface */
    public void close(){
        System.out.println("Close From ClassTwo");
    }
}


/* Automatic Resource Handling (ARE): Don't need to close the resource
explicitly */
class ClassThree{

    public static void main(String...a){

/* Automatic Resource Handling: notice we have not call close() but it
will call by JVM

The resources will be close in reverse order. ClassTwo resources will
be close the ClassOne Resources will be close.*/

        try(ClassOne co = new ClassOne();
                ClassTwo ct = new ClassTwo()){

            co.show();
            ct.display();
        }
        catch(ArithmeticException e){
            System.out.println(e);
        }
    }
}
```

Throw Keyword

The Java “**throw**” keyword is used to explicitly throw an exception. We can throw either **checked or unchecked exception** in java by throw keyword. The throw keyword is mainly used to throw custom exception. I will see custom exceptions later.

The syntax of java throw keyword is given below.

```
throw new IOException("sorry device error);
```

Program:

```
/* Class Bhi Meri, Object Bhi Meri, Throw Bhi Hum Hi Karenge Aur
Catch Bhi Hum Hi Karenge */

// Custom Exception :
class AgeException extends Exception{
    // Constructor
    AgeException(String s){
        super(s); // calling parent constructor
    }
}

/* Employee Call That Will Use Custom Exception Class */
class Employee{
    int age;

    public static void main(String...arg){
        Employee emp = new Employee();
/* Putting value through Command Line Argument(String to Int) */
        emp.setAge(Integer.parseInt(arg[0]));

        System.out.println("\n Program is continue.....");
    }

    void setAge(int age){
        if(age < 18){

            try{

throw new AgeException("Invalid Age, Employee age must me
greater than 18");
```

```
                }catch(AgeException a){
                    System.out.println(a);
                }
            }else{
                this.age = age;
                System.out.println("Age is Set: " + this.age);
            }
        }

        void onlyThrow(){

        }
}
```

We can also throw without try catch block but we won’t be able to catch the exception and program will terminate.

Exception Handling With Calling Chain

```
class ClassOne{


    public void show(){
      /* 1. Exception occurs so JVM will look for try catch (handler)
here.
       If try-catch is not here then will check from this method is
called. In this from case display() */
        int x = 10/0;
    }

    public void display(){
        /*  2. As we see inside show() as well as display(), there is
no handler then it will look from display() called. */
        show();
    }

    public void xyz(){
        /* 3. As we see inside xyz(), there is no handler then it will
look from xyz is called */
        display();
    }
}

class ClassTwo{

    public static void main(String...a){
        ClassOne co = new ClassOne();
/* 4. At last it will look inside main(). if it does not get here then
the JVM will catch the object and terminate the program
          co.xyz();  It will terminate the program.
        main() function is also inside handler tryCatch implicitly */

        try{
            co.xyz();
        }catch(ArithmeticException e){
            System.out.println("Problem is: " + e);

            // To understand the Calling Chain, we can print the stack
            e.printStackTrace();
        }

        System.out.println("Program Continue");
    }
}
```

1. Exception occurs so JVM will look for try catch (handler) here.
   If try-catch is not here then will check from this method is called. In this from case display ().

2. As we see inside show () as well as display (), there is no handler then it will look from display () called.

3. As we see inside xyz (), there is no handler then it will look from xyz is called.

4. At last it will look inside main (). if it does not get here then the JVM will catch the object and terminate the program co.xyz(); It will terminate the program.

main () function is also inside handler try-Catch implicitly.

Throws Keyword

“**throws”** keyword is used to make our program or function handler free.

“**throws**” keyword is used to give indication about an exception to the end-user of program that there may occur an exception so it is better for the programmer to provide the exception handling code so that normal flow can be maintained.

“**throws**” keyword is also used for forwarding the only checked exception in calling chain because Unchecked Exception are automatically forwarded in a calling chain.

Rule: If you are calling a method that declares an exception, you must either caught or declare the exception.

Rule: In case you declare the exception, if exception does not occur, the code will be executed fine.

Rule: In case you declare the exception if exception occurs, an exception will be thrown at runtime because throws does not handle the exception.

Program:

```
import java.io.*;
class ClassOne{

    int div() throws IOException, ArithmeticException{
        System.out.println("Enter First Number");

/* this is checked exception but it won't force to put handler
because we have declared throws keyword after function name */
        int x = System.in.read();

        System.out.println("Enter Second Number");
        int y = System.in.read();
        return x/y;
    }
}

class ClassTwo{
    public static void main(String...a) {
        ClassOne co = new ClassOne();
// We have added indication at div() function using throws
keyword. // That will force to put handler at the time of
calling div()

 co.div();          // It will force to put handler.

        /* The correct way is this….
          try{
            int z = co.div();
            System.out.println(z);
        }catch(IOException e){
            System.out.println("Exception is : " + e);
        }*/

        System.out.println("Program is continue.....");

    }
}
```

However we can also make our program handler free by adding throws keyword after "main" method. But this is not recommended because after that JVM will catch the exception and terminate the program.

Exception Handling With Method Overriding

There are many rules if we talk about method overriding with exception handling. The Rules are as follows:

```
/* 1) Rule: If the superclass method declares an exception,
subclass overridden method can declare same, subclass exception
or no exception but cannot declare parent exception. */

import java.io.*;
class Parent{

    void show() throws IOException{
        System.out.println("Parent Show");
    }

    void display() throws IOException{
        System.out.println("Parent Display");
    }
```

```java
    void drawing() throws IOException{
        System.out.println("Parent Display");
    }

    void watch()  throws IOException{
        System.out.println("Parent Watch");
    }
}

public class CaseTwo extends Parent{

    public static void main(String...a){
        try{

            new CaseTwo().show();
            new CaseTwo().display();
            new CaseTwo().drawing();
        }catch(Exception e){
            System.out.println(e);
        }
    }

/* Overriding the show() with no exception. It Can Happen. */
    void show(){
        System.out.println("Child Show");
    }

/* Overriding the display() with same exception. It Can Happen.
*/

    void display()  throws IOException{
        System.out.println("Child Display");
    }

/* Overriding the drawing() with class that is child of parent's
drawing() throwing exception class. FileNotFoundException is the
Child of IOException class. It Can Happen. */

    void drawing() throws FileNotFoundException{
        System.out.println("Child Drawing");
    }

/* Overriding the watch() with class that is parent exception of
my parent's watch() throwing exception class Exception is the
parent of IOException. It Can't Happen */

    /*  void watch() throws Exception{
            System.out.println("Watch From Child");
        }
    */
}
```

Custom Exception

If we are creating our own Exception that is known as custom exception or user-defined exception. Java custom exceptions are used to customize the exception according to user need.

Advantage of Custom Exception:

By the help custom exception, we can print a customize message to end-user. It also makes the debugging easy.

Generally custom exception are made either by inheriting an exception class or child of it.

IQ: As we know there are two types of exception. If we make custom exception then it will be checked or unchecked exception.

Ans: It totally depends on parent class. If parent class is unchecked then our class will be unchecked and if parent class is checked exception then our custom exception will be also checked exception.

Custom Exception Program:

```
class AgeException extends Exception{

    String s;

    AgeException(String s){
        this.s = s;
    }

    public String toString(){
        return s;
    }
}


class Employee{
    int age;

    public static void main(String...a){
        Employee emp = new Employee();
        try
{
            emp.setAge(2);
        }catch(AgeException e){
            System.out.println(e);
        }

        System.out.println("Program is continue....");
    }

    public void setAge(int age)  throws AgeException
{
        if(age < 18){
                throw new AgeException("Age Must Be Greater Than
18");

        }else{
            this.age = age;
            System.out.println("Age Is Set: " + this.age);
        }
    }
}
```

// Proper Way to Make Custom Exception

```
// Proper Way to Make an Exception

import java.io.*;

class InSufficientFundsException extends Exception{
private double amount;

public InSufficientFundsException(double amount){
    this.amount = amount;
}

public double getAmount(){
    return amount;
}
}

class BankTransaction{
private double balance;
private int accountNumber;

public BankTransaction(int accountNumber){
    this.accountNumber = accountNumber;
}

public void deposit(double amount){
    balance += amount;
}

public void withdraw(double amount) throws
InSufficientFundsException{
    if(amount <= balance){
        balance -= amount;
    }else{
        double needs = (amount - balance);
        throw new InSufficientFundsException(needs);
    }
}

public double getBalance(){
    return balance;
}

public int getAccountNumber(){
    return accountNumber;
}
}

public class BankAccount{
public static void main(String...a){
    BankTransaction bt = new BankTransaction(1402003464);

    System.out.println("Depositing $500");
    bt.deposit(500.00);


    try{
        //for(int i=0; i< )
        System.out.println("\n Withdrawing $100");
        bt.withdraw(100.00);
        System.out.println("\n Withdrawing $200");
        bt.withdraw(200.00);
        System.out.println("\n Withdrawing $200");
        bt.withdraw(400.00);

    }catch(InSufficientFundsException e){
        System.out.println("Sorry! But you are shot " +
e.getAmount());
        e.printStackTrace();
    }

    System.out.println("Program is continue...\nProgram is
continue...\nProgram is continue...\n");
}
}
```

Java Nested try block

The try block within a try block is known as nested try block in java.

Why use nested try block

Sometimes a situation may arise where a part of a block may cause one error and the entire block itself may cause another error. In such cases, exception handlers have to be nested.

Object is allowed to throw: - Throwable.

Mine:

"Load Runner" is something that test the application with multiple user.

Interview Questions:

Q. What is a "stacktrace" and how does it relate to an exception?

A stack trace provides the names of the classes and methods that were called, from the start of the application to the point an exception occurred.

It's a very useful debugging tool since it enables us to determine exactly where the exception was thrown in the application and the original causes that led to it.

Q. Is there any way of throwing a checked exception from a method that does not have a *throws* clause?

Yes. We can take advantage of the type erasure performed by the compiler and make it think we are throwing an unchecked exception, when, in fact; we're throwing a checked exception:

```
public <T extends Throwable> T sneakyThrow(Throwable ex) throws
T {
    throw (T) ex;
}

public void methodWithoutThrows() {
    this.<RuntimeException>sneakyThrow(new Exception("Checked
Exception"));
}
```

Q. What is unreachable catch block error?
Ans: If we are having multiple catch block and they are not in child to parent class exception class order then we will get "unreachable catch block error".

What are run time exceptions in java? Give example?

The exceptions which occur at run time are called as run time exceptions. These exceptions are unknown to compiler. All sub classes of "**java.lang.RunTimeException"** and "**java.lang.Error**" are run time exceptions. These exceptions are **unchecked** type of exceptions. For example, NumberFormatException, NullPointerException, ClassCastException, ArrayIndexOutOfBoundException, StackOverflowError etc.

Q. Difference between "**ClassNotFoundException**" Vs "**NoClassDefFoundError**" in Java.

Ans: at instapaper…

What is ClassCastException in java?

ClassCastException is a RunTimeException which occurs when JVM unable to cast an object of one type to another type.

Throwable In Java:

Throwable is a super class for all types of errors and exceptions in java. This class is a member of **java.lang**package. Only instances of this class or its sub classes are thrown by the java virtual machine or by the throw statement. The only argument of catch block must be of this type or its sub classes.

Q. Can we override a super class method which is throwing an unchecked exception with checked exception in the sub class?

No. If a super class method is throwing an unchecked exception, then it can be overridden in the sub class with same exception or any other unchecked exceptions but cannot be overridden with checked exceptions.

Q. Give some examples to checked exceptions?

ClassNotFoundException, SQLException, IOException

Q. Give some examples to unchecked exceptions?

NullPointerException, ArrayIndexOutOfBoundsException, NumberFormatException

What are important methods of Java Exception Class?

Exception and all of its subclasses doesn't provide any specific methods and all of the methods are defined in the base class Throwable.

1. **String getMessage()** – This method returns the message String of Throwable and the message can be provided while creating the exception through it's constructor.
2. **String getLocalizedMessage()** – This method is provided so that subclasses can override it to provide locale specific message to the calling program. Throwable class implementation of this method simply use getMessage() method to return the exception message.
3. **synchronized Throwable getCause()** – This method returns the cause of the exception or null id the cause is unknown.
4. **String toString()** – This method returns the information about Throwable in String format, the returned String contains the name of Throwable class and localized message.
5. **void printStackTrace()** – This method prints the stack trace information to the standard error stream, this method is overloaded

and we can pass PrintStream or PrintWriter as argument to write the stack trace information to the file or stream.

What is difference between final, finally and finalize in Java?

“**final**” and “**finally**” are keywords in java whereas “**finalize**” is a method.

final keyword can be used with class variables so that they can’t be reassigned, with class to avoid extending by classes and with methods to avoid overriding by subclasses, finally keyword is used with try-catch block to provide statements that will always get executed even if some exception arises, usually finally is used to close resources. finalize () method is executed by Garbage Collector before the object is destroyed, its great way to make sure all the global resources are closed.

Out of the three, only finally is related to java exception handling.

Can we have an empty catch block?

We can have an empty catch block but it’s the example of worst programming. We should never have empty catch block because if the exception is caught by that block, we will have no information about the exception and it will be a nightmare to debug it. There should be at least a logging statement to log the exception details in console or log files.