**Exception Handling**

**Exception:-**

Exception handling is one of the most important feature of java programming that allows us to handle the runtime errors caused by exceptions.

**What is an exception?**

An Exception is an unwanted event that interrupts the normal flow of the program. When an exception occurs program execution gets terminated. In such cases we get a system generated error message. The good thing about exceptions is that they can be handled in Java. By handling the exceptions we can provide a meaningful message to the user about the issue rather than a system generated message, which may not be understandable to a user.

**Why an exception occurs?**

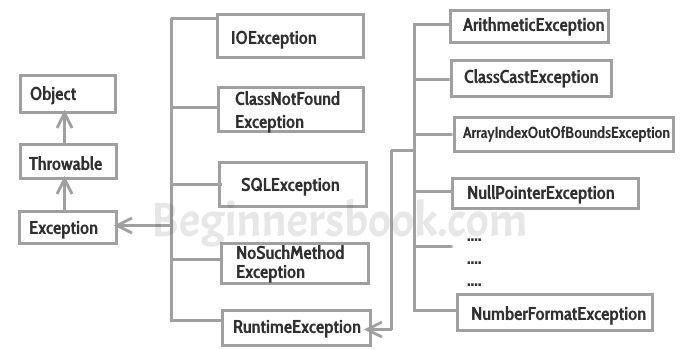
There can be several reasons that can cause a program to throw exception. For example: Opening a non-existing file in your program, Network connection problem, bad input data provided by user etc.

**Exception Handling**

If an exception occurs, which has not been handled by programmer then program execution gets terminated and a system generated error message is shown to the user. For example look at the system generated exception below:

**Errors** indicate that something severe enough has gone wrong, the application should crash rather than try to handle the error.

**Exceptions** are events that occurs in the code. A programmer can handle such conditions and take necessary corrective actions. Few examples:  
**a) NullPointerException –** When you try to use a reference that points to null.  
**b) ArithmeticException –** When bad data is provided by user, for example, when you try to divide a number by zero this exception occurs because dividing a number by zero is undefined.  
**c) ArrayIndexOutOfBoundsException –** When you try to access the elements of an array out of its bounds, for example array size is 5 (which means it has five elements) and you are trying to access the 10th element.



**Types of exceptions**

There are two types of exceptions in Java:  
1)Checked exceptions  
2)Unchecked exceptions

**Checked exceptions**

All exceptions other than Runtime Exceptions are known as Checked exceptions as the compiler checks them during compilation to see whether the programmer has handled them or not. If these exceptions are not handled/declared in the program, you will get compilation error. For example, SQLException, IOException, ClassNotFoundException etc.

**Unchecked Exceptions**

Runtime Exceptions are also known as Unchecked Exceptions. These exceptions are not checked at compile-time so compiler does not check whether the programmer has handled them or not but it’s the responsibility of the programmer to handle these exceptions and provide a safe exit. For example, ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc.

**How to handle Exceptions:-**

Try, catch and finally blocks are used to handle exceptions.

**Try block**

The try block contains set of statements where an exception can occur. A try block is always followed by a catch block, which handles the exception that occurs in associated try block. A try block must be followed by catch blocks or finally block or both.

**Syntax of try block**

try {

//statements that may cause an exception

}

**Catch block**

A catch block is where you handle the exceptions, this block must follow the try block. A single try block can have several catch blocks associated with it. You can catch different exceptions in different catch blocks. When an exception occurs in try block, the corresponding catch block that handles that particular exception executes. For example if an arithmetic exception occurs in try block then the statements enclosed in catch block for arithmetic exception executes.

**Syntax of try catch in java**

try

{

//statements that may cause an exception

}

catch (exception(type) e(object))‏

{

//error handling code

**Example:-**

class Example1 {

public static void main(String args[]) {

int num1, num2;

try {

num1 = 0;

num2 = 62 / num1;

System.out.println(num2);

System.out.println("Hey I'm at the end of try block");

}

  catch (Exception e) {

System.out.println("Exception occurred");

}

catch (ArithmeticException e) {

System.out.println("You should not divide a number by zero");

}

System.out.println("I'm out of try-catch block in Java.");

}}

**Output:**

You should not divide a number by zero

I'm out of try-catch block in Java.

**Multiple catch blocks**

**Example**

class Example2{

public static void main(String args[]){

try{

int a[]=new int[7];

a[4]=30/0;

System.out.println("First print statement in try block");

}

catch(ArithmeticException e){

System.out.println("Warning: ArithmeticException");

}

catch(ArrayIndexOutOfBoundsException e){

System.out.println("Warning: ArrayIndexOutOfBoundsException");

}

catch(Exception e){

System.out.println("Warning: Some Other exception");

}

System.out.println("Out of try-catch block...");

}

}

Output:

Warning: ArithmeticException

Out of try-catch block...

**Nested try catch block**

When a try catch block is present in another try block then it is called the nested try catch block. Each time a try block does not have a catch handler for a particular exception, then the catch blocks of parent try block are inspected for that exception, if match is found that catch block executes.

If neither catch block nor parent catch block handles exception then the system generated message would be shown for the exception, similar to what we see when we don’t handle exception.

**Syntax of Nested try Catch**

//Main try block

try {

statement 1;

statement 2;

//try-catch block inside another try block

try {

statement 3;

statement 4;

//try-catch block inside nested try block

try {

statement 5;

statement 6;

}

catch(Exception e2) {

//Exception Message

}

}

catch(Exception e1) {

//Exception Message

}

}

//Catch of Main(parent) try block

catch(Exception e3) {

//Exception Message

}

**Example: Nested try block**

class Nest{

public static void main(String args[]){

//Parent try block

try{

//Child try block1

try{

System.out.println("Inside block1");

int b =45/0;

System.out.println(b);

}

catch(ArithmeticException e1){

System.out.println("Zerodivide: e1");

}

//Child try block2

try{

System.out.println("Inside block2");

String s=null;

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

}

catch(NullPointerException e2){

System.out.println("Nullpointer Exception: e2");

}

System.out.println("Just other statement");

}

catch(ArrayIndexOutOfBoundsException e4){

System.out.println("ArrayIndexOutOfBoundsException");

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

}

catch(Exception e5){

System.out.println("Exception");

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

}

System.out.println("Next statement..");

}}

Output:

Inside block1

Exception: e1

Inside block2

Nullpointer Exception

Inside parent try catch block

Next statement.

**Java Finally block:-**

A finally block contains all the crucial statements that must be executed whether exception occurs or not. The statements present in this block will always execute regardless of whether exception occurs in try block or not such as closing a connection, stream etc.

**Syntax of Finally block**

try {

//Statements that may cause an exception

}

catch {

//Handling exception

}

finally {

//Statements to be executed

}

**A Simple Example of finally block**

class Example

{

public static void main(String args[]) {

try{

int num=121/0;

System.out.println(num);

}

catch(ArithmeticException e){

System.out.println("Number should not be divided by zero");

}

finally{

System.out.println("This is finally block");

}

System.out.println("Out of try-catch-finally");

}

}

**Output:**

Number should not be divided by zero

This is finally block

Out of try-catch-finally

**How to throw exception**

We can define our own set of conditions or rules and throw an exception explicitly using throw keyword. For example, we can throw ArithmeticException when we divide number by 5, or any other numbers, what we need to do is just set the condition and throw any exception using throw keyword. Throw keyword can also be used for throwing custom exceptions.

**Syntax of throw keyword:**

throw new exception\_class("error message");

**Example:**

public class ThrowExample {

static void checkEligibilty(int stuage, int stuweight){

if(stuage<12 && stuweight<40) {

throw new ArithmeticException("Student is not eligible for registration");

}

else {

System.out.println("Student Entry is Valid!!");

}

}

public static void main(String args[]){

System.out.println("Welcome to the Registration process!!");

checkEligibilty(10, 39);

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

}

}

**Throws clause:-**

import java.io.\*;

class ThrowExample {

void myMethod(int num) throws IOException, ClassNotFoundException

{

if(num==1)

throw new IOException("IOException Occurred");

else

throw new ClassNotFoundException ("ClassNotFoundException");

} }

public class Example1{

public static void main(String args[]){

try{

ThrowExample obj=new ThrowExample();

obj.myMethod(1);

}catch(Exception ex){

System.out.println(ex);

} }}

Output:

java.io.IOException: IOException Occurred

**Throw vs Throws**

The throw and throws are the concepts of exception handling in Java where the throw keyword throws the exception explicitly from a method or a block of code, whereas the throws keyword is used in the signature of the method.

**The differences between throw and throws in Java are:**

| **S. No.** | **Key Difference** | **throw** | **throws** |
| --- | --- | --- | --- |
| 1. | Point of Usage | The **throw** keyword is used inside a function. It is used when it is required to throw an Exception logically. | The **throws** keyword is used in the function signature. It is used when the function has some statements that can lead to exceptions. |
| 2. | Exceptions Thrown | The **throw** keyword is used to throw an exception explicitly. It can throw only one exception at a time. | The **throws** keyword can be used to declare multiple exceptions, separated by a comma. Whichever exception occurs, if matched with the declared ones, is thrown automatically then. |
| 3. | Syntax | Syntax of **throw** keyword includes the instance of the Exception to be thrown. Syntax wise throw keyword is followed by the instance variable. | Syntax of **throws** keyword includes the class names of the Exceptions to be thrown. Syntax wise throws keyword is followed by exception class names. |
| 4. | Propagation of Exceptions | **throw** keyword cannot propagate checked exceptions. It is only used to propagate the unchecked Exceptions that are not checked using the throws keyword. | **throws** keyword is used to propagate the checked Exceptions only. |

**User defined exception (or) Custom Exception**

**Example:-**

class MyException extends Exception

{

public MyException(String s)

{

// Call constructor of parent Exception

super(s);

}

}

public class Main

{

public static void main(String args[])

{

try

{

// Throw an object of user defined exception

throw new MyException("GeeksGeeks");

}

catch (MyException ex)

{

System.out.println("Caught");

// Print the message from MyException object

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

}

}

}

**Output:-**

Caught

GeeksGeeks