**Exception handling in java**

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:

An exception generated by the system is given below

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

*ExceptionDemo : The class name*

*main : The method name*

*ExceptionDemo.java : The filename*

*java:5 : Line number*

This message is not user friendly so a user will not be able to understand what went wrong. In order to let them know the reason in simple language, we handle exceptions. We handle such conditions and then prints a user friendly warning message to user, which lets them correct the error as most of the time exception occurs due to bad data provided by user.

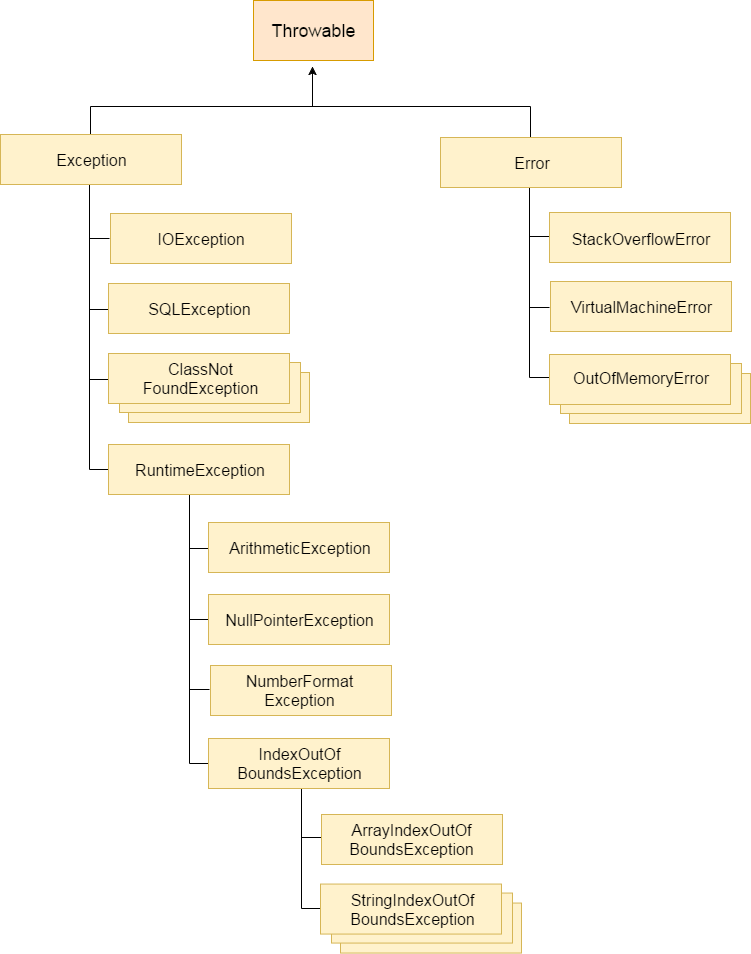
### Advantage of exception handling

Exception handling ensures that the flow of the program doesn’t break when an exception occurs. For example, if a program has bunch of statements and an exception occurs mid way after executing certain statements then the statements after the exception will not execute and the program will terminate abruptly.  
By handling we make sure that all the statements execute and the flow of program doesn’t break.

## Difference between error and exception

**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.



**IOException:**

This exception occurs when an IO operation has failed for some reason.

**SQLException:**

An SQLException can occur both in the driver and the database.

**ClassNotFoundException** :

 The **ClassNotFoundException** is thrown when the **Java**Virtual Machine (JVM) tries to load a particular class and the specified class cannot be found in the classpath.

**ArithmeticException:**

 Thrown when an exceptional arithmetic condition has occurred. For example, an integer "divide by zero" throws an instance of this class.

Example:

int a=50/0;//ArithmeticException

**NullPointerException**:

NullPointerException is thrown when program attempts to use an object reference that has the null value.

Example:

String s=null;

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

**NumberFormatException:**

A Java NumberFormatException usually occurs when you try to do something like convert a String to a numeric value, like an int, float, double, long, etc.

Example:

String s="abc";

int i=Integer.parseInt(s);//NumberFormatException

**ArrayIndexOutOfBoundsException:**

* **ArrayIndexOutOfBoundsException** is thrown to indicate that we are trying to access array element with an illegal index.
* This exception is thrown when the index is either negative or greater than or equal to the size of the array.

Example:

int a[]=new int[5];

a[10]=50; //ArrayIndexOutOfBoundsException

**StringIndexOutOfBoundsException :**

This is thrown by String methods to indicate that an index is either negative or greater than the size of the string. For some methods such as the charAt method, this exception also is thrown when the index is equal to the size of the string.

Example:

String str = "foobar"; // length = 6

char error = str.c**harA**t(7); // index input should be less than or equal to length-1

char error = str.charAt(-1); // cannot use negative indexes

**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.

**Java Exception Keywords**

try

catch

finally

throw

throws

## 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:

**public** **class** packageEx12 {

**public** **static** **void** main(String[] args) {

**try** {

**int** a=10/0;

System.***out***.println("value= " +a);

}

**catch**(Exception e) {

System.***out***.println("exception handled");

}

}

}

Output:

Exception Handled

**Multiple catch blocks in Java**

The example we seen above is having multiple catch blocks, lets see few rules about multiple catch blocks with the help of examples.

1. As I mentioned above, a single try block can have any number of catch blocks.  
2. A generic catch block can handle all the exceptions. Whether it is ArrayIndexOutOfBoundsException or ArithmeticException or NullPointerException or any other type of exception, this handles all of them.

catch(Exception e){

  //This catch block catches all the exceptions

}

If you are wondering why we need other catch handlers when we have a generic that can handle all.

This is because in generic exception handler you can display a message but you are not sure for which type of exception it may trigger so it will display the same message for all the exceptions and user may not be able to understand which exception occurred.

Thats the reason you should place is at the end of all the specific exception catch blocks

3. If no exception occurs in try block then the catch blocks are completely ignored.

4. Corresponding catch blocks execute for that specific type of exception:  
catch(ArithmeticException e) is a catch block that can hanlde ArithmeticException

catch(NullPointerException e) is a catch block that can handle NullPointerException

5. You can also throw exception, which is an advanced topic and I have covered it in separate tutorials: user defined exception, throws keyword, throw vs throws.

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...

**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.

Syntax:

....

try

{

statement 1;

statement 2;

try

{

statement 1;

statement 2;

}

catch(Exception e)

{

}

}

catch(Exception e)

{

}

....

**Java nested try example**

Let's see a simple example of java nested try block.

class Excep6{

public static void main(String args[]){

try{

try{

System.out.println("going to divide");

int b =39/0;

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

try{

int a[]=new int[5];

a[5]=4;

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

System.out.println("other statement);

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

System.out.println("normal flow..");

}

}

**Java finally block:**

**Java finally block** is a block that is used *to execute important code* such as closing connection, stream etc.

Java finally block is always executed whether exception is handled or not.

Java finally block follows try or catch block.

**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 block will always execute

\* even if there is no exception in try block

\*/

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

**Few Important points regarding finally block**

1. A finally block must be associated with a try block, you cannot use finally without a try block. You should place those statements in this block that must be executed always.

2. Finally block is optional, as we have seen in previous tutorials that a try-catch block is sufficient for [exception handling](https://beginnersbook.com/2013/04/java-exception-handling/), however if you place a finally block then it will always run after the execution of try block.

3. In normal case when there is no exception in try block then the finally block is executed after try block. However if an exception occurs then the catch block is executed before finally block.

4. An exception in the finally block, behaves exactly like any other exception.

5. The statements present in the **finally block** execute even if the try block contains control transfer statements like return, break or continue.

**Cases when the finally block doesn’t execute**

The circumstances that prevent execution of the code in a finally block are:  
– The death of a Thread  
– Using of the System. exit() method.  
– Due to an exception arising in the finally block.

**Throw Keyword in java:**

In Java we have already defined exception classes such as ArithmeticException, NullPointerException, ArrayIndexOutOfBounds exception etc.

These exceptions are set to trigger on different-2 conditions.

For example when we divide a number by zero, this triggers ArithmeticException, when we try to access the array element out of its bounds then we get ArrayIndexOutOfBoundsException.

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, I have covered that in a separate tutorial, see Custom Exceptions in Java.

**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..");

}

}

**Output:**

Welcome to the Registration process!!Exception in thread "main"

java.lang.ArithmeticException: Student is not eligible for registration

at beginnersbook.com.ThrowExample.checkEligibilty(ThrowExample.java:9)

at beginnersbook.com.ThrowExample.main(ThrowExample.java:18)

**Java throws keyword**

The **Java throws keyword** is used to declare an exception.

**Throws clause in java – Exception handling**

As we know that there are two types of exception checked and unchecked. Checked exception (compile time) force you to handle them, if you don’t handle them then the program will not compile.  
On the other hand unchecked exception (Runtime) doesn’t get checked during compilation.

**Throws keyword** is used for handling checked exceptions . By using throws we can declare multiple exceptions in one go.

**What is the need of having throws keyword when you can handle exception using try-catch?**

Well, thats a valid question. We already know we can handle exceptions using try-catch block.  
The throws does the same thing that try-catch does but there are some cases where you would prefer throws over try-catch.

**Example:**

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

**Difference between throw and throws in java**

**Throw vs Throws in java**

1. **Throws clause** is used to declare an exception, which means it works similar to the try-catch block. On the other hand **throw** keyword is used to throw an exception explicitly.

2. If we see syntax wise than **throw** is followed by an instance of Exception class and **throws** is followed by exception class names.  
For example:

throw new ArithmeticException("Arithmetic Exception");

and

throws ArithmeticException;

3. Throw keyword is used in the method body to throw an exception, while throws is used in method signature to declare the exceptions that can occur in the statements present in the method.

For example:  
**Throw:**

...

void myMethod() {

try {

//throwing arithmetic exception using throw

throw new ArithmeticException("Something went wrong!!");

}

catch (Exception exp) {

System.out.println("Error: "+exp.getMessage());

}

}

...

**Throws:**

...

//Declaring arithmetic exception using throws

void sample() throws ArithmeticException{

//Statements

}

...

4. You can throw one exception at a time but you can handle multiple exceptions by declaring them using throws keyword.

For example:  
**Throw:**

void myMethod() {

//Throwing single exception using throw

throw new ArithmeticException("An integer should not be divided by zero!!");

}

..

**Throws:**

//Declaring multiple exceptions using throws

void myMethod() throws ArithmeticException, NullPointerException{

//Statements where exception might occur

}

**Throw Example**

public class Example1{

void checkAge(int age){

if(age<18)

throw new ArithmeticException("Not Eligible for voting");

else

System.out.println("Eligible for voting");

}

public static void main(String args[]){

Example1 obj = new Example1();

obj.checkAge(13);

System.out.println("End Of Program");

}

}

**Output:**

Exception in thread "main" java.lang.ArithmeticException:

Not Eligible for voting

at Example1.checkAge(Example1.java:4)

at Example1.main(Example1.java:10)

**Throws Example**

public class Example1{

int division(int a, int b) throws ArithmeticException{

int t = a/b;

return t;

}

public static void main(String args[]){

Example1 obj = new Example1();

try{

System.out.println(obj.division(15,0));

}

catch(ArithmeticException e){

System.out.println("You shouldn't divide number by zero");

}

}

}

**Output:**

You shouldn't divide number by zero

**User defined exception in java:**

In java we can create our own exception class and throw that exception using throw keyword. These exceptions are known as user-defined or custom exceptions.

**Example:**

class InvalidProductException extends Exception

{

public InvalidProductException(String s)

{

// Call constructor of parent Exception

super(s);

}

}

public class Example1

{

void productCheck(int weight) throws InvalidProductException{

if(weight<100){

throw new InvalidProductException("Product Invalid");

}

}

public static void main(String args[])

{

Example1 obj = new Example1();

try

{

obj.productCheck(60);

}

catch (InvalidProductException ex)

{

System.out.println("Caught the exception");

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

}

}}

**Output:**

Caught the exception

Product Invalid