**Exception Handling in Java**

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

In java, exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.

Exception Handling is a mechanism to handle runtime errors such as ClassNotFound, IO, SQL, Remote etc.

The core advantage of exception handling is to maintain the normal flow of the application. Exception normally disrupts the normal flow of the application that is why we use exception handling. Let's take a scenario:

statement 1;

statement 2;

statement 3;

statement 4;

statement 5;//exception occurs

statement 6;

statement 7;

statement 8;

statement 9;

statement 10;

Suppose there are 10 statements in your program and there occurs an exception at statement 5, rest of the code will not be executed i.e. statement 6 to 10 will not run. If we perform exception handling, rest of the statement will be executed. That is why we use exception handling in java.

**Hierarchy of Java Exception classes**



**Types of Exception**

There are mainly two types of exceptions: checked and unchecked where error is considered as unchecked exception.

* Checked Exception (marked in yellow)
* Unchecked Exception -> Error (marked in blue)

**Difference between checked and unchecked exceptions**

**1) Checked Exception**

The classes that extend Throwable class except RuntimeException and Error are known as checked exceptions e.g.IOException, SQLException etc (Marked yellow in above diagram) Checked exceptions are checked at compile-time.

**2) Unchecked Exception**

The classes that extend RuntimeException are known as unchecked exceptions e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc. Unchecked exceptions are not checked at compile-time rather they are checked at runtime.

**3) Error**

Error is irrecoverable e.g. OutOfMemoryError, VirtualMachineError, AssertionError etc.

Common scenarios where exceptions may occur

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

1) Scenario where ArithmeticException occurs

If we divide any number by zero, there occurs an ArithmeticException.

int a=50/0;//ArithmeticException

2) Scenario where NullPointerException occurs

If we have null value in any variable, performing any operation by the variable occurs an NullPointerException.

String s=null;

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

3) Scenario where NumberFormatException occurs

The wrong formatting of any value, may occur NumberFormatException. Suppose I have a string variable that have characters, converting this variable into digit will occur NumberFormatException.

String s="abc";

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

4) Scenario where ArrayIndexOutOfBoundsException occurs

If you are inserting any value in the wrong index, it would result ArrayIndexOutOfBoundsException as shown below:

int a[]=new int[5];

a[10]=50; //ArrayIndexOutOfBoundsException

**Java try-catch**

Java try block is used to enclose the code that might throw an exception. It must be used within the method. Java try block must be followed by either catch or finally block.

Syntax of java try-catch

**try** {

} **catch** (Exception e) {

// **TODO**: handle exception

}

Syntax of try-finally block

**try** {

} **finally** {

// **TODO**: handle finally clause

}

**Java catch block**

Java catch block is used to handle the Exception. It must be used after the try block only. You can use multiple catch block with a single try.

Problem without exception handling

Let's try to understand the problem if we don't use try-catch block.

**public** **class** Testtrycatch1 {

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

**int** data = 50 / 0;// may throw exception

System.***out***.println("rest of the code...");

}

}

**Output:**

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

at Testtrycatch1.main(Testtrycatch1.java:3)

As displayed in the above example, rest of the code is not executed (in such case, rest of the code... statement is not printed).

Solution by exception handling

Let's see the solution of above problem by java try-catch block.

**public** **class** Testtrycatch2 {

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

**try** {

**int** data = 50 / 0;

} **catch** (ArithmeticException e) {

System.***out***.println(e);

}

System.***out***.println("rest of the code...");

}

}

**Output:**

java.lang.ArithmeticException: / by zero

rest of the code...

Now, as displayed in the above example, rest of the code is executed i.e. rest of the code... statement is printed.

The JVM firstly checks whether the exception is handled or not. If exception is not handled, JVM provides a default exception handler that performs the following tasks:

* Prints out exception description.
* Prints the stack trace (Hierarchy of methods where the exception occurred).
* Causes the program to terminate.

But if exception is handled by the application programmer, normal flow of the application is maintained i.e. rest of the code is executed.

**Java Multi catch block**

If you have to perform different tasks at the occurrence of different Exceptions, use java multi catch block.

**public** **class** TestMultipleCatchBlock {

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

**try** {

**int** a[] = **new** **int**[5];

a[5] = 30 / 0;

} **catch** (ArithmeticException e) {

System.***out***.println("task1 is completed");

} **catch** (ArrayIndexOutOfBoundsException e) {

System.***out***.println("task 2 completed");

} **catch** (Exception e) {

System.***out***.println("common task completed");

}

System.***out***.println("rest of the code...");

}

}

Output:

task1 is completed

rest of the code...

**Rule: At a time only one Exception is occurred and at a time only one catch block is executed.**

**Rule: All catch blocks must be ordered from most specific to most general i.e. catch for ArithmeticException must come before catch for Exception .**

**class** TestMultipleCatchBlock1 {

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

**try** {

**int** a[] = **new** **int**[5];

a[5] = 30 / 0;

} **catch** (Exception e) {

System.***out***.println("common task completed");

} **catch** (ArithmeticException e) {

System.***out***.println("task1 is completed");

} **catch** (ArrayIndexOutOfBoundsException e) {

System.***out***.println("task 2 completed");

}

System.***out***.println("rest of the code...");

}

}

Output:

Unreachable catch block for ArithmeticException. It is already handled by the catch block for Exception

**Java Nested try block**

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

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** {

**try** {

} **catch** (Exception e) {

// **TODO**: handle exception

}

} **catch** (Exception e) {

// **TODO**: handle exception

}

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("handled"); //handled is not printed

}

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

}

}

Output:

going to divide

java.lang.ArithmeticException: / by zero

java.lang.ArrayIndexOutOfBoundsException: 5

other statement

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.



**Java throw keyword**

The Java throw keyword is used to explicitly throw an exception.

We can throw either checked or uncheked exception in java by throw keyword. The throw keyword is mainly used to throw custom exception.

The syntax of java throw keyword is given below.

throw exception;

In this example, we have created the validate method that takes integer value as a parameter. If the age is less than 18, we are throwing the ArithmeticException otherwise print a message welcome to vote.

**public** **class** TestThrow1{

**static** **void** validate(**int** age){

**if**(age<18)

**throw** **new** ArithmeticException("not valid");

**else**

System.***out***.println("welcome to vote");

}

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

*validate*(13);

System.***out***.println("rest of the code...");

}

}

Output:

Exception in thread "main" java.lang.ArithmeticException: not valid

at TestThrow1.validate(TestThrow1.java:4)

at TestThrow1.main(TestThrow1.java:9)

**Java Exception propagation**

An exception is first thrown from the top of the stack and if it is not caught, it drops down the call stack to the previous method,If not caught there, the exception again drops down to the previous method, and so on until they are caught or until they reach the very bottom of the call stack. This is called exception propagation.

Rule: By default Unchecked Exceptions are forwarded in calling chain (propagated).

Program of Exception Propagation

**package** pack1;

**class** TestExceptionPropagation1 {

**void** m() {

**int** data = 50 / 0;

}

**void** n() {

System.***out***.println("n1");

m();

System.***out***.println("n2");

}

**void** p() {

**try** {

System.***out***.println("P1");

n();

System.***out***.println("P2");

} **catch** (Exception e) {

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

}

}

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

TestExceptionPropagation1 obj = **new** TestExceptionPropagation1();

obj.p();

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

}

}

Output:

P1

n1

exception handled

normal flow...

s

In the above example exception occurs in m() method where it is not handled, so it is propagated to previous n() method where it is not handled, again it is propagated to p() method where exception is handled.

Exception can be handled in any method in call stack either in main() method, p() method, n() method or m() method.

Rule: By default, Checked Exceptions are not forwarded in calling chain (propagated).

Program which describes that checked exceptions are not propagated

**class** TestExceptionPropagation2 {

**void** m() {

**throw** **new** java.io.IOException("device error");// compilation error

}

**void** n() {

m();

}

**void** p() {

**try** {

n();

} **catch** (Exception e) {

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

}

}

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

TestExceptionPropagation2 obj = **new** TestExceptionPropagation2();

obj.p();

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

}

}

Output:

Unhandled exception type IOException

**Java throws keyword**

The Java throws keyword is used to declare an exception. It gives information to the programmer that an exception may occur so it is better for the programmer to provide the exception handling code so that normal flow can be maintained.

Exception Handling is mainly used to handle the checked exceptions. If any unchecked exception such as NullPointerException occurs it is programmers fault that he is not performing check up before the code being used.

Syntax of java throws

return\_type method\_name() **throws** exception\_class\_name{

//method code

}

**Advantage of Java throws keyword**

It provides information to the caller of the method about the exception.

Let's see the example of java throws clause which describes that checked exceptions can be propagated by throws keyword.

**import** java.io.IOException;

**class** Testthrows1 {

**void** m() **throws** IOException {

**throw** **new** IOException("device error");// checked exception

}

**void** n() **throws** IOException {

m();

}

**void** p() {

**try** {

n();

} **catch** (Exception e) {

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

}

}

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

Testthrows1 obj = **new** Testthrows1();

obj.p();

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

}

}

Output:

exception handled

normal flow...

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

There are two cases:

Case1:You caught the exception i.e. handle the exception using try/catch.

Case2:You declare the exception i.e. specifying throws with the method.

Case1: You handle the exception

In case you handle the exception, the code will be executed fine whether exception occurs during the program or not.

**import** java.io.\*;

**class** M {

**void** method() **throws** IOException {

**throw** **new** IOException("device error");

}

}

**public** **class** Testthrows2 {

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

**try** {

M m = **new** M();

m.method();

} **catch** (Exception e) {

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

}

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

}

}

**Output:**

exception handled

normal flow...

Case2: You declare the exception

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

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

Program if exception does not occur

**import** java.io.\*;

**class** M {

**void** method() **throws** IOException {

System.***out***.println("device operation performed");

}

}

**class** Testthrows3 {

**public** **static** **void** main(String args[]) **throws** IOException {// declare exception

M m = **new** M();

m.method();

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

}

}

**Output:**

device operation performed

normal flow...

B)Program if exception occurs

**import** java.io.\*;

**class** M {

**void** method() **throws** IOException {

**throw** **new** IOException("device error");

}

}

**class** Testthrows4 {

**public** **static** **void** main(String args[]) **throws** IOException {// declare exception

M m = **new** M();

m.method();

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

}

}

Exception in thread "main" java.io.IOException: device error

at M.method(Testthrows4.java:5)

at Testthrows4.main(Testthrows4.java:12)

**Difference between throw and throws in Java**

|  |  |  |
| --- | --- | --- |
| **No.** | **Throw** | **throws** |
| 1) | Java throw keyword is used to explicitly throw an exception. | Java throws keyword is used to declare an exception. |
| 2) | Checked exception cannot be propagated using throw only. | Checked exception can be propagated with throws. |
| 3) | Throw is followed by an instance. | Throws is followed by class. |
| 4) | Throw is used within the method. | Throws is used with the method signature. |
| 5) | You cannot throw multiple exceptions. | You can declare multiple exceptions e.g. public void method()throws IOException,SQLException. |

**Java throw example**

**void** m(){

**throw** **new** ArithmeticException("sorry");

}

Java throws example

**void** m()**throws** ArithmeticException{

//method code

}

Java throw and throws example

**void** m()**throws** ArithmeticException{

**throw** **new** ArithmeticException("sorry");

}

**Difference between final, finally and finalize**

|  |  |  |  |
| --- | --- | --- | --- |
| **No.** | **Final** | **finally** | **finalize** |
| 1) | Final is used to apply restrictions on class, method and variable. Final class can't be inherited, final method can't be overridden and final variable value can't be changed. | Finally is used to place important code, it will be executed whether exception is handled or not. | Finalize is used to perform clean up processing just before object is garbage collected. |
| 2) | Final is a keyword. | Finally is a block. | Finalize is a method. |

**Java final example**

**class** FinalExample {

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

**final** **int** x = 100;

x = 200;// Compile Time Error

}

}

**Java finally example**

**class** FinallyExample {

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

**try** {

**int** x = 300;

} **catch** (Exception e) {

System.***out***.println(e);

} **finally** {

System.***out***.println("finally block is executed");

}

}

}

**Java finalize example**

**class** FinalizeExample {

**public** **void** finalize() {

System.***out***.println("finalize called");

}

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

FinalizeExample f1 = **new** FinalizeExample();

FinalizeExample f2 = **new** FinalizeExample();

f1 = **null**;

f2 = **null**;

System.*gc*();

}

}

ExceptionHandling with MethodOverriding in Java

* If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception but it can declare unchecked exception.
* If the superclass method declares an exception, subclass overridden method can declare same, subclass exception or no exception but cannot declare parent exception.

**If the superclass method does not declare an exception**

1) Rule: If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception.

**import** java.io.\*;

**class** Parent {

**void** msg() {

System.***out***.println("parent");

}

}

**class** TestExceptionChild **extends** Parent {

**void** msg() **throws** IOException {

System.***out***.println("TestExceptionChild");

}

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

Parent p = **new** TestExceptionChild();

p.msg();

}

}

**Output:**

Exception in thread "main" java.lang.Error: Unresolved compilation problem:

Exception IOException is not compatible with throws clause in Parent.msg()

at TestExceptionChild.msg(TestExceptionChild.java:10)

at TestExceptionChild.main(TestExceptionChild.java:16)

2) Rule: If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception but can declare unchecked exception.

**import** java.io.\*;

**class** Parent {

**void** msg() {

System.***out***.println("parent");

}

}

**class** TestExceptionChild1 **extends** Parent {

**void** msg() **throws** ArithmeticException {

System.***out***.println("child");

}

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

Parent p = **new** TestExceptionChild1();

p.msg();

}

}

**Output:**

child

**If the superclass method declares an exception**

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.

**Example in case subclass overridden method declares parent exception**

**import** java.io.\*;

**class** Parent {

**void** msg() **throws** ArithmeticException {

System.***out***.println("parent");

}

}

**class** TestExceptionChild2 **extends** Parent {

**void** msg() **throws** Exception {

System.***out***.println("child");

}

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

Parent p = **new** TestExceptionChild2();

**try** {

p.msg();

} **catch** (Exception e) {

}

}

}

Output:

Exception in thread "main" java.lang.Error: Unresolved compilation problem:

Exception Exception is not compatible with throws clause in Parent.msg()

at TestExceptionChild2.msg(TestExceptionChild2.java:10)

at TestExceptionChild2.main(TestExceptionChild2.java:17)

**Example in case subclass overridden method declares same exception**

**import** java.io.\*;

**class** Parent {

**void** msg() **throws** Exception {

System.***out***.println("parent");

}

}

**class** TestExceptionChild3 **extends** Parent {

**void** msg() **throws** Exception {

System.***out***.println("child");

}

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

Parent p = **new** TestExceptionChild3();

**try** {

p.msg();

} **catch** (Exception e) {

}

}

}

**Output:**

child

**Example in case subclass overridden method declares subclass exception**

**import** java.io.\*;

**class** Parent {

**void** msg() **throws** Exception {

System.***out***.println("parent");

}

}

**class** TestExceptionChild4 **extends** Parent {

**void** msg() **throws** ArithmeticException {

System.***out***.println("child");

}

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

Parent p = **new** TestExceptionChild4();

**try** {

p.msg();

} **catch** (Exception e) {

}

}

}

**Output:**

child

**Example in case subclass overridden method declares no exception**

**import** java.io.\*;

**class** Parent {

**void** msg() **throws** Exception {

System.***out***.println("parent");

}

}

**class** TestExceptionChild5 **extends** Parent {

**void** msg() {

System.***out***.println("child");

}

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

Parent p = **new** TestExceptionChild5();

**try** {

p.msg();

} **catch** (Exception e) {

}

}

}

**Output:**

child

**Java Custom Exception**

If you are creating your own Exception then it is known as custom exception or user-defined exception. Java custom exceptions are used to customize the exception according to user need.

Let's see a simple example of java custom exception.

**class** InvalidAgeException **extends** Exception {

InvalidAgeException(String s) {

**super**(s);

}

}

**class** TestCustomException1 {

**static** **void** validate(**int** age) **throws** InvalidAgeException {

**if** (age < 18)

InvalidAgeException iae = **new** InvalidAgeException("not valid");

**throw iae**;

**else**

System.***out***.println("welcome to vote");

}

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

**try** {

*validate*(13);

} **catch** (Exception m) {

System.***out***.println("Exception occured: " + m);

}

System.***out***.println("rest of the code...");

}

}

Test it Now

Output:

Exception occured: InvalidAgeException: not valid

rest of the code...