Exception Handling in Java

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

**What is exception**

Dictionary Meaning: Exception is an abnormal condition.

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

**What is exception handling**

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

**Advantage of Exception Handling**

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;

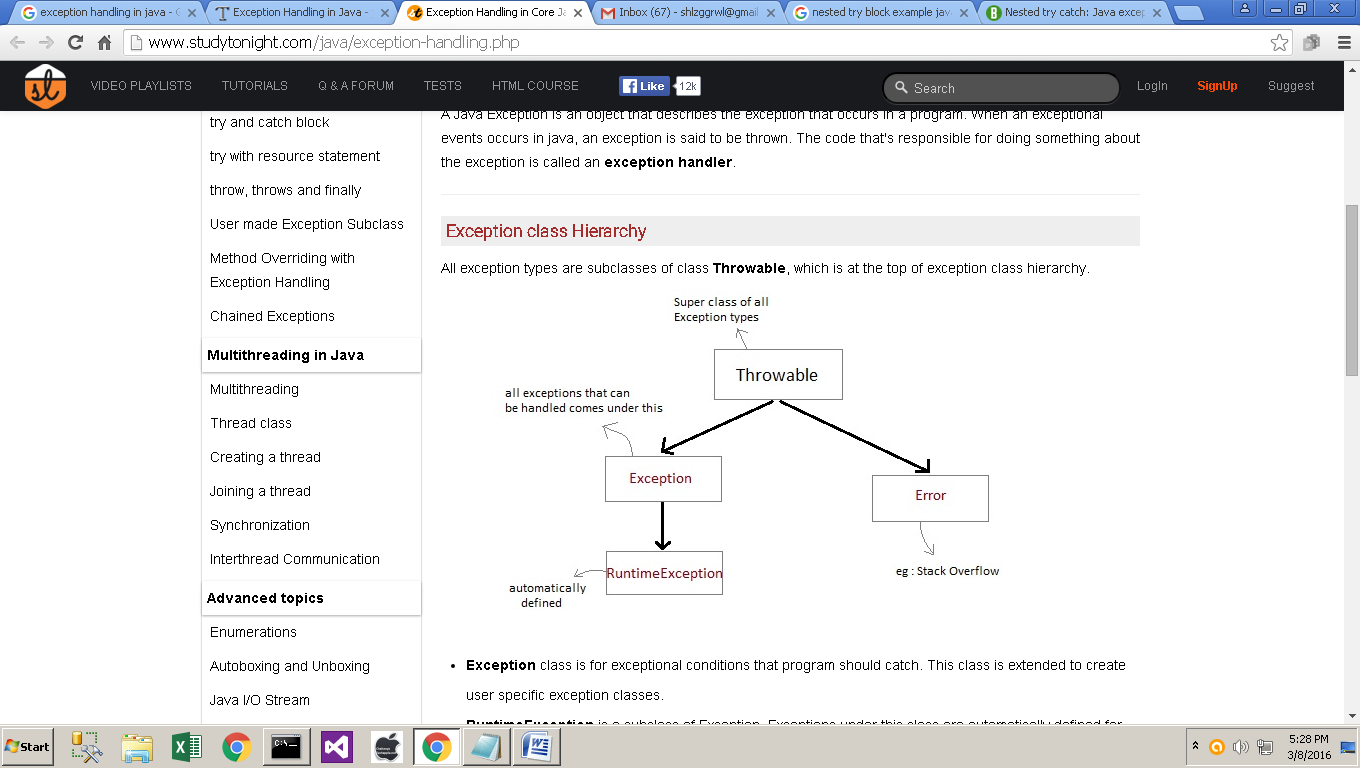
statement 8;

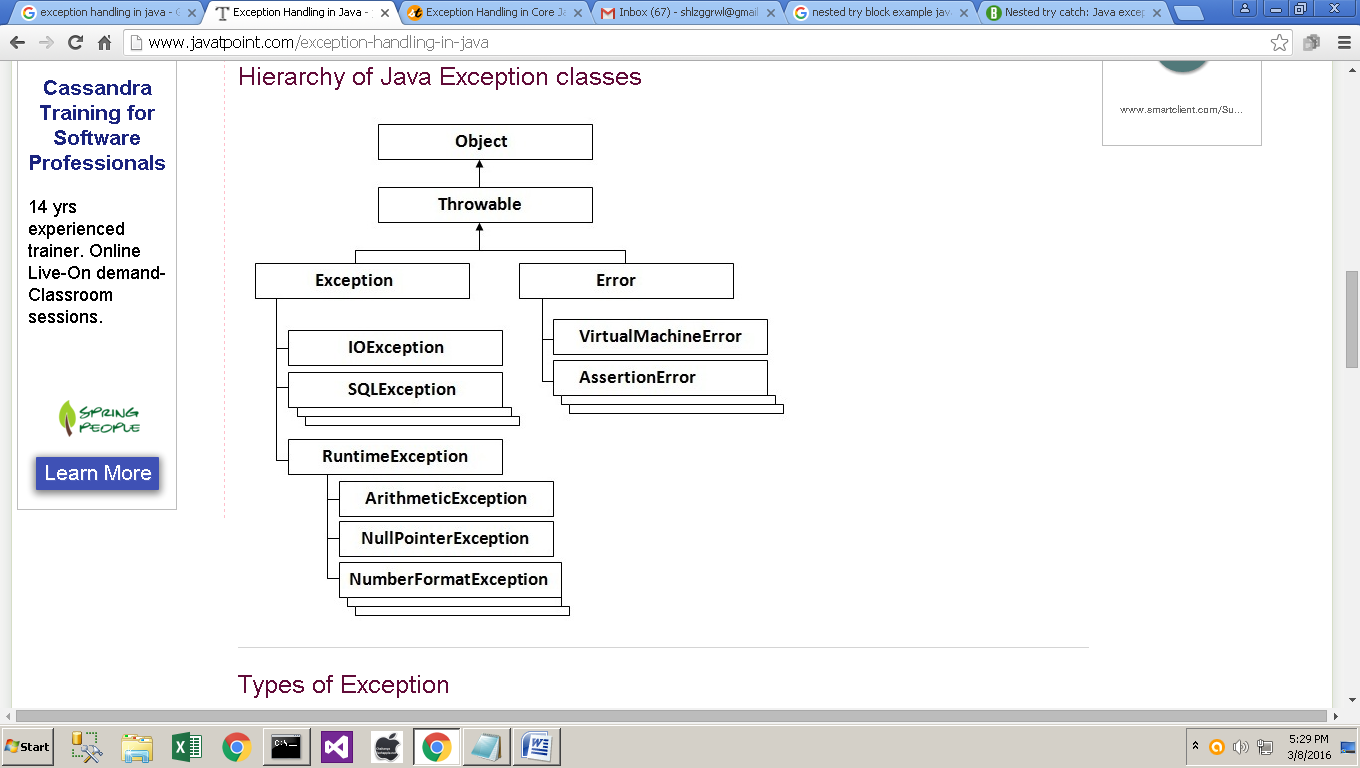
statement 9;

statement 10;

Suppose there is 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. The sun Microsystems says there are three types of exceptions:

1. Checked Exception
2. Unchecked Exception
3. Error

**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. 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 anNullPointerException.

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";

inti=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 Exception Handling Keywords**

There are 5 keywords used in java exception handling.

1. try
2. catch
3. finally
4. throw
5. throws

**Java try block**

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{

//code that may throw exception

}catch(Exception\_class\_Name ref){}

**Syntax of try-finally block**

try{

//code that may throw exception

}finally{}

**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

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

**There can be 100 lines of code after exception. So all the code after exception will not be executed.**

**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...");

} }

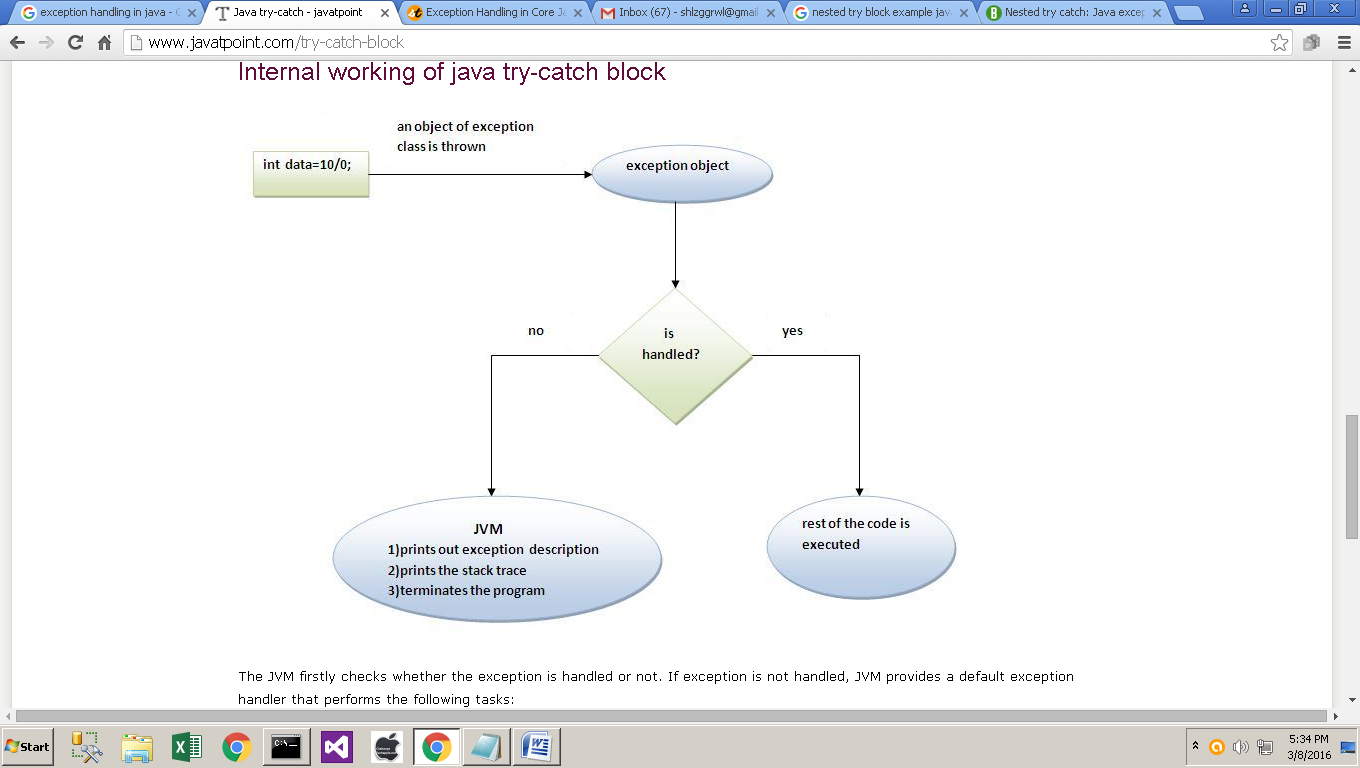
**Output:**

Exception in thread main 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.

**Internal working of java try-catch block**

****

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 catch multiple exceptions**

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

**Let's see a simple example of 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 completed

rest of the code...

**Rule: At a time only one Exception is occured 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:**

Compile-time error

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**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.

**Example 1:**

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("Exception: e1");

}

//Child try block2

try{

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

int b =45/0;

System.out.println(b);

}

catch(ArrayIndexOutOfBoundsException e2){

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

}

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

}

catch(ArithmeticException e3){

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

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

}

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

Arithmetic Exception

Inside parent try catch block

Next statement..

**Example 2:**

classNestingDemo{

public static void main(String args[]){

//main try-block

try{

//try-block2

try{

//try-block3

try{

intarr[]= {1,2,3,4};

/\* I'm trying to display the value of

\* an element which doesn't exist. The

\* code should throw an exception

\*/

System.out.println(arr[10]);

}catch(ArithmeticException e){

System.out.print("Arithmetic Exception");

System.out.println(" handled in try-block3");

}

}

catch(ArithmeticException e){

System.out.print("Arithmetic Exception");

System.out.println(" handled in try-block2");

}

}

catch(ArithmeticException e3){

System.out.print("Arithmetic Exception");

System.out.println(" handled in main try-block");

}

catch(ArrayIndexOutOfBoundsException e4){

System.out.print("ArrayIndexOutOfBoundsException");

System.out.println(" handled in main try-block");

}

catch(Exception e5){

System.out.print("Exception");

System.out.println(" handled in main try-block");

} }}

**Output:**

ArrayIndexOutOfBoundsException handled in main try-block

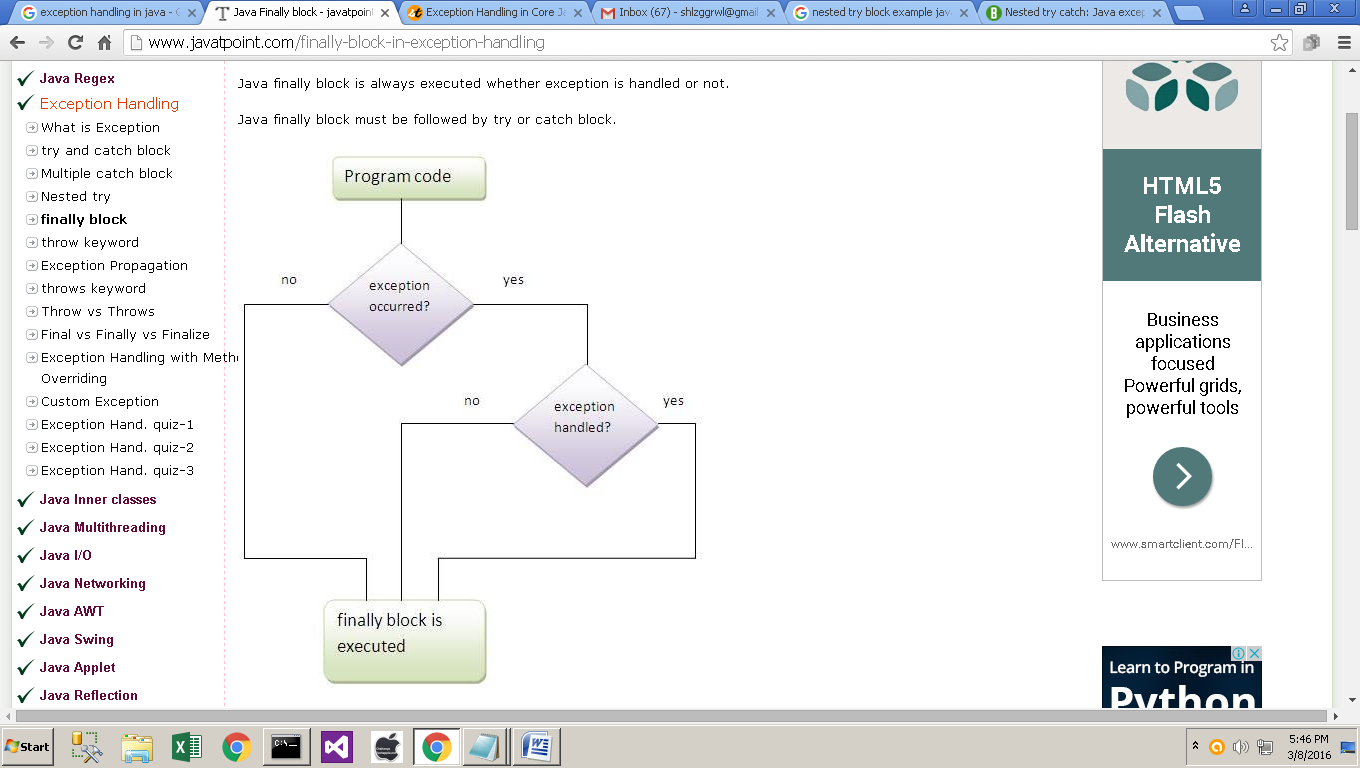
\*\*\*\*\*\*\*\*\*\*\*

**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 must be followed by try or catch block.



Note: If you don't handle exception, before terminating the program, JVM executes finally block(if any).

**Why use java finally**

Finally block in java can be used to put "cleanup" code such as closing a file, closing connection etc.

**Usage of Java finally**

Let's see the different cases where java finally block can be used.

**Case 1**

Let's see the java finally example where exception doesn't occur.

classTestFinallyBlock{

public static void main(String args[]){

try{

int data=25/5;

System.out.println(data);

}

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

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

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

} }

**Output**:

5

finally block is always executed

rest of the code...

**Case 2**

Let's see the java finally example where exception occurs and not handled.

class TestFinallyBlock1{

public static void main(String args[]){

try{

int data=25/0;

System.out.println(data);

}

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

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

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

} }

**Output**:finally block is always executed

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

**Case 3**

Let's see the java finally example where exception occurs and handled.

public class TestFinallyBlock2{

public static void main(String args[]){

try{

int data=25/0;

System.out.println(data);

}

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

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

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

} }

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

finally block is always executed

rest of the code...

**Rule: For each try block there can be zero or more catch blocks, but only one finally block.**

**Note: The finally block will not be executed if program exits(either by calling System.exit() or by causing a fatal error that causes the process to abort).**

\*\*\*\*\*\*\*\*\*\*\*\*\*

**Java throw exception**

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. We will see custom exceptions later.

**The syntax of java throw keyword is given below.**

throw exception;

**Let's see the example of throw IOException.**

throw new IOException("sorry device error”);

**java throw keyword example**

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

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**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**

class TestExceptionPropagation1{

void m(){

int data=50/0;

}

void n(){

m();

}

void p(){

try{

n();

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

}

public static void main(String args[]){

TestExceptionPropagation1 obj=new TestExceptionPropagation1();

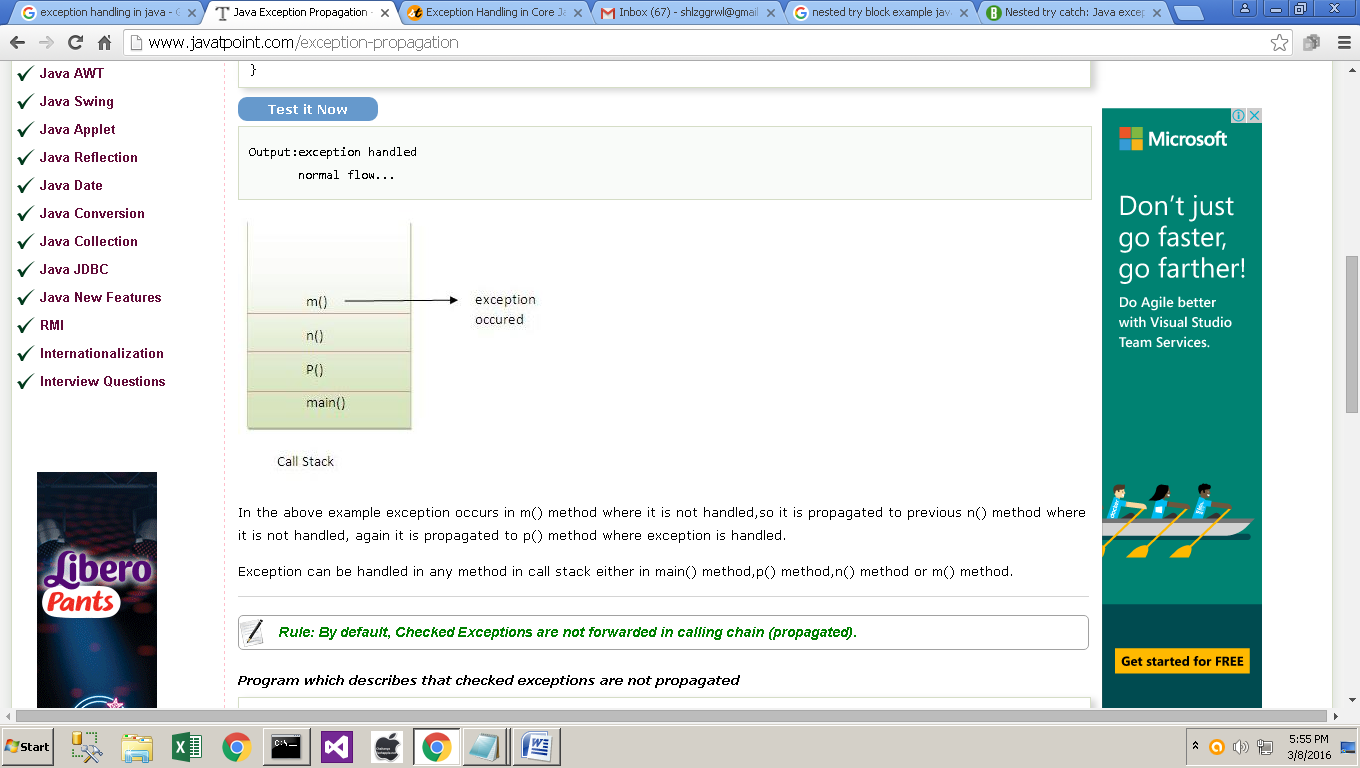
obj.p();

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

} }

**Output**:exception handled

normal flow...



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");//checked exception

}

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**:Compile Time Error

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Java throws keyword**

The Java throws keyword is used to declare an exception. It gives an information to the programmer 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.

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

**Syntax of java throws**

return\_typemethod\_name() throws exception\_class\_name{

//method code

}

**Which exception should be declared**

Ans) checked exception only, because:

* **unchecked Exception**: under your control so correct your code.
* **error**: beyond your control e.g. you are unable to do anything if there occurs VirtualMachineError or StackOverflowError.

**Advantage of Java throws keyword**

Now Checked Exception can be propagated (forwarded in call stack).

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

**Java throws example**

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

importjava.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:**

1. **Case1**:You caught the exception i.e. handle the exception using try/catch.
2. **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 occures, an exception will be thrown at runtime because throws does not handle the exception.

**A)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...");

} }

**Output**:Runtime Exception

**Que) Can we rethrow an exception?**

Yes, by throwing same exception in catch block.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Difference between throw and throws in Java**

There are many differences between throw and throws keywords. A list of differences between throw and throws are given below:

|  |  |  |
| --- | --- | --- |
| **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**

class Test

{ static void avg()

{ try

{ throw new ArithmeticException("demo");

}

catch(ArithmeticException e)

{ System.out.println("Exception caught");

} }

public static void main(String args[])

{ avg();

}}

**Java throws example**

class Test

{

static void check() throws ArithmeticException

{

System.out.println("Inside check function");

throw new ArithmeticException("demo");

}

public static void main(String args[])

{

try

{

check();

}

catch(ArithmeticException e)

{

System.out.println("caught" + e);

} }}

**Java throw and throws example**

void m()throws ArithmeticException{

throw new ArithmeticException("sorry");

}

\*\*\*\*\*\*\*\*\*\*\*\*

**Difference between final, finally and finalize**

There are many differences between final, finally and finalize. A list of differences between final, finally and finalize are given below:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **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**

classFinalExample{

public static void main(String[] args){

finalint x=100;

x=200;//Compile Time Error

}}

**Java finally example**

classFinallyExample{

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**

classFinalizeExample{

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();

}}