

EXCEPTION HANDLING

Fundamentals of exception handling:

- Exception Handling is a mechanism to handle exception at runtime.
- Exception is a condition that occurs during program execution and lead to program termination abnormally.
- There can be several reasons that can lead to exceptions, including programmer error, hardware failures, files that need to be opened cannot be found etc.
- The problem with the exception is, it terminates the program and skips rest of the execution--- that means if a program have 100 lines of code and at line 10 an exception occur then program will terminate immediately by skipping execution of rest 90 lines of code.
- To handle this problem, we use exception handling that avoid program termination and continue the execution by skipping exception code.

Uncaught Exceptions

- When we don't handle the exceptions, it leads to unexpected program termination.
- In this program, an ArithmeticException will be thrown due to divide by zero.

class UncaughtException

```
{    public static void main(String args[])  
  
    {  
  
        int a = 0;  
  
        int b = 7/a;    // Divide by zero, will lead to exception  
  
    } }
```

- This will lead to an exception at runtime , hence the Java run-time system will construct an exception and then throw it.
- As we don't have any mechanism for handling exception in the above program, hence the default handler (JVM) will handle the exception and will print the details of the exception on the terminal.

java.lang.ArithmeticException: / by zero
at UncaughtException.main(UncaughtException.java:4)

name and description of Exception

class name

file name

Stack Trace
(line at which exception occurred)

Handling Exceptions

- Java provides keywords to handle exceptions in the program. They are
 - Try : code doubtful of raising an exception is put in try block
 - Catch: It acts as exception handler.
 - Finally: the code in finally block gets executed compulsorily irrespective of the exception is raised or not
 - Throw: It throws the exception explicitly.
 - Throws: It informs for the possible exception raised by a method

try & catch block

- Try and catch both are Java keywords and used for exception handling.
- The try block is used to enclose the doubtful code.
- Doubtful code is a code that may raise an exception during program execution.
- The catch block also known as handler is used to handle the exception.
- It handles the exception thrown by the code enclosed into the try block.
- Try block must provide a catch handler or a finally block.
- The catch block must be used after the try block only.

To declare try catch block, a general syntax is given below.

```
try{  
    // doubtful code  
}catch(ExceptionClass ob){}
```

Exception handling is done by transferring the execution of a program to an appropriate exception handler (catch block) when exception occurs.

Example:

```
class Excp
{
    public static void main(String args[])
    {
        int a,b,c;

        try
        {
            a = 0;

            b = 10;

            c = b/a;

            System.out.println("This line will not be executed");

        }

        catch(ArithmeticException e)
        {
            System.out.println("Divided by zero");

        }

        System.out.println("After exception is handled");

    }
}
```

- An exception will be thrown by this program as we are trying to divide a number by zero inside try block.
- The program control is transferred outside try block at that line.
- Thus the line "*This line will not be executed*" is never parsed by the compiler.
- The exception thrown is handled in catch block.
- Once the exception is handled, the program control is continued with the next line in the program i.e after catch block.
- Thus the line "*After exception is handled*" is printed.

Multiple catch blocks

- A try block can be followed by multiple catch blocks.
- It means we can have any number of catch blocks after a single try block.
- If an exception occurs in the try block, the exception is passed to the first catch block in the list.
- If the exception type matches with the first catch block it gets caught, if not the exception is passed down to the next catch block.
- This continues until the exception is caught or falls through all catches.

Multiple Catch Syntax

To declare the multiple catch handler, we can use the following syntax.

```
try
{
    // suspected code
}
catch(Exception1 e)
{
    // handler code
}
catch(Exception2 e)
{
    // handler code
}
```

- The multiple catch blocks are useful when we are not sure about the type of exception during program execution.

EXAMPLE 1:

```
class MultipleCatchDemo1 {    public static void main(String[] args) {  
  
    try  
  
    {  
  
        Integer in = new Integer("abc");  
  
        in.intValue();  
  
        }  
    catch (ArithmeticException e)  
  
    {  
  
        System.out.println("Arithmetic " + e);  
  
    }  
    catch (NumberFormatException e)  
  
    {  
  
        System.out.println("Number Format Exception " + e);  
  
    } } }
```

OUTPUT: Number Format Exception java.lang.NumberFormatException: For input string: "abc"

- In the above example, we used multiple catch blocks and based on the type of exception second catch block is executed.

EXAMPLE 2:

```
class MultipleCatchDemo2
{
    public static void main(String[] args)
    {
        try
        {
            int a[]=new int[10];
            System.out.println(a[20]);
        }
        catch(ArrayIndexOutOfBoundsException e)
        {
            System.out.println("ArrayIndexOutOfBoundsException --> "+e);
        }
        catch(ArithmeticException e)
        {
            System.out.println("Arithmetic Exception --> "+e);
        }
    }
}
```

OUTPUT:

ArrayIndexOutOfBoundsException -->java.lang. ArrayIndexOutOfBoundsException:20

- In the above example, we used multiple catch blocks and based on the type of exception first catch block is executed.

NOTE:

- *While using multiple catch statements, it is important to remember that sub classes of class *Exception* inside catch must come before any of their super classes otherwise it will lead to compile time error.*
- This is because in Java, if any code is unreachable, then it gives compile time error.

```
class Excep
```

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

{    try

    {

        int arr[]={1,2};

        arr[2]=3/0;

    }

    catch(Exception e)  //This block handles all Exception

    {

        System.out.println("Generic exception");

    }

    catch(ArrayIndexOutOfBoundsException e)  //This block is unreachable

    {

        System.out.println("array index out of bound exception");
```

```
} }}
```

Nested try statements: 1

```
class NestedTryDemo1
```

```
{
```

```
    public static void main(String args[])
```

```
{
```

```
    try{
```

```
        int a=args.length;
```

```
        int b=5/a;
```

```
        try
```

```
        {
```

```
            int c=45/(a-1);
```

```
            int p[]=new int[-5];
```

```
        }
```

```
    catch(NegativeArraySizeException e)
```

```
    {
```

```
        System.out.println("negative array size exception in nested catch");
```

```
}
```

```
}//closing of outer try
```

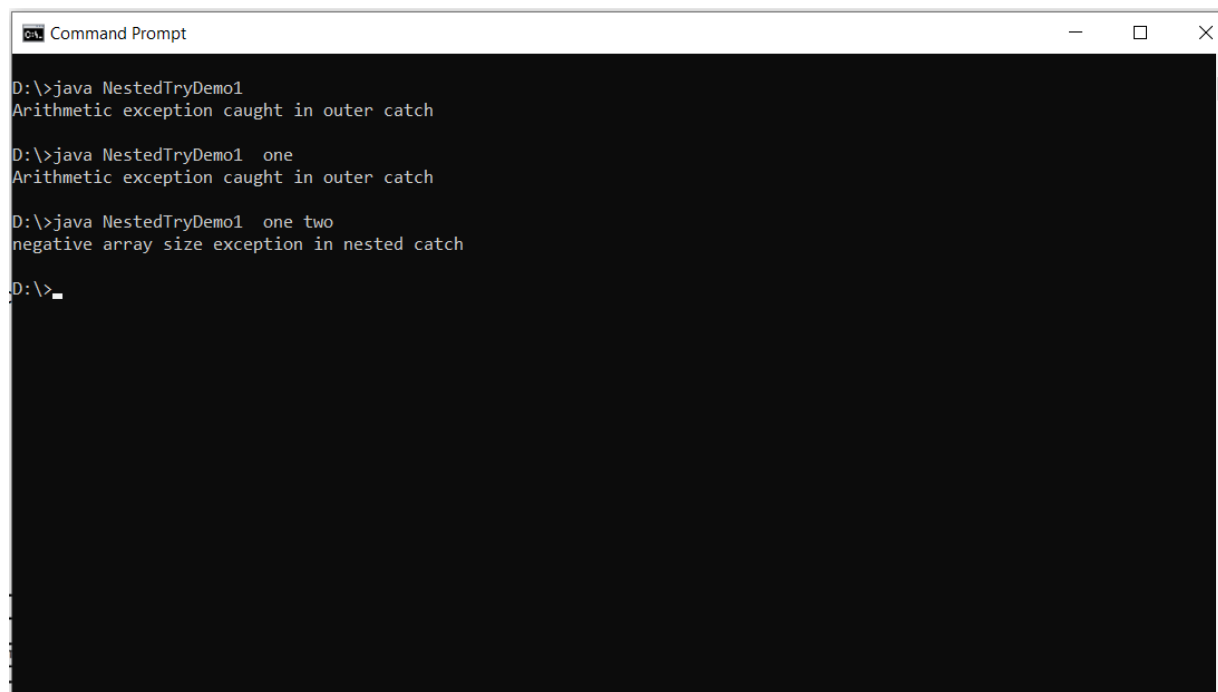
```
catch(ArithmeticException e)
```

```
{
```

```
System.out.println("Arithmetic exception caught in outer catch");
```

```
}
```

```
}}
```



```
Command Prompt
D:\>java NestedTryDemo1
Arithmetic exception caught in outer catch

D:\>java NestedTryDemo1 one
Arithmetic exception caught in outer catch

D:\>java NestedTryDemo1 one two
negative array size exception in nested catch

D:\>_
```

- In the above program there are 3 test cases

Test case1 – we did not give any command line arguments which raised an arithmetic exception at the line

`int b=5/a;` where variable 'a' is the no. of command line arguments given.

This exception as is raised in the outer try , the outer catch which is written to handle arithmetic exceptions will handle it

Test case2 – we gave one command line argument which made the value of variable 'a' as 1, the statement

`int c=45/(a-1);`

raised an arithmetic exception. This statement is inside the nested try but the nested catch is not having appropriate classname so it checks the outer catch and as the outercatch matches with that type of exception raised in the inner try – outer catch is executed.

Test case3 – we gave three command line arguments which made the value of the variable 'a' as 2; so the two statements which raised exception in test cases 1 & 2 don't raise an exception now. The statement `int p[]=new int[-5];` in the inner try raises a `NegativeArraySizeException` and so the inner catch which matches with it is executed.

-> this illustrates the fact that the exceptions raised by inner try block also can be handled by the outer catch if inner catch doesn't match.

Nested try statements: 2

-> the following program explains that if a called method definition has try and catch blocks and if that method call is placed inside a try block ,then also it is treated as nested try.

```
class NestedTryDemo2
```

```
{
```

```
static void meth(int a)
```

```
{  
try  
  
    {  
        int c=45/(a-1);  
        int p[]=new int[-5];  
    }  
catch(NegativeArraySizeException e)  
  
    {  
        System.out.println("negative array size exception in nested catch");  
    }  
}  
  
public static void main(String args[])  
  
{  
int a=args.length;  
try{
```

```
int b=5/a;

meth(a);

} //closing of outer try

catch(ArithmeticException e)

{

System.out.println("Arithmetic exception caught in outer catch");

}

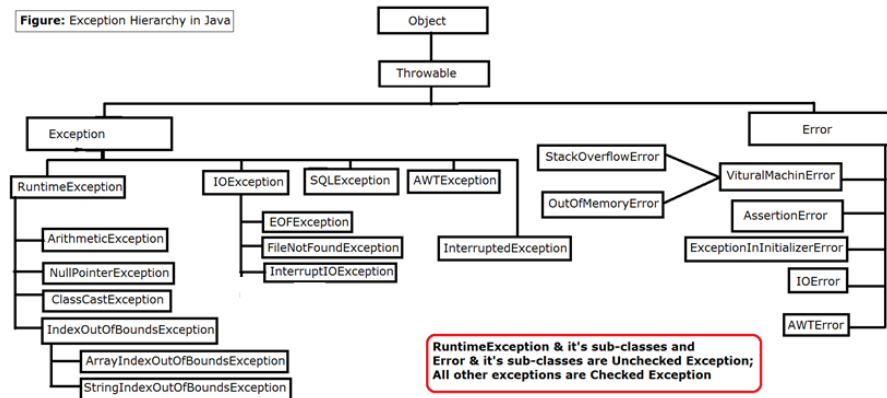
}}
```

Important points

1. If you do not explicitly use the try catch blocks in your program, java will provide a default exception handler, which will print the exception details on the terminal, whenever exception occurs.
2. Super class **Throwable** overrides **toString()** function, to display error message in form of string.
3. While using multiple catch block, always make sure that sub-classes of Exception class comes before any of their super classes. Else you will get compile time error.
4. In nested try catch, the inner try block uses its own catch block as well as catch block of the outer try, if required.
5. Only the object of Throwable class or its subclasses can be thrown.

EXCEPTION HIERARCHY

Figure: Exception Hierarchy in Java



Types of Exceptions

In Java, exceptions broadly can be categories into checked exception, unchecked exception and error based on the nature of exception.

- **Checked Exception**

The exception that can be predicted by the JVM at the compile time for **example**: File that need to be opened is not found, SQLException etc. These types of exceptions must be checked at compile time.

- **Unchecked Exception**

Unchecked exceptions are the class that extends RuntimeException class. Unchecked exception are ignored at compile time and checked at runtime. For **example** : ArithmeticException, NullPointerException, Array Index out of Bound exception. Unchecked exceptions are checked at runtime.

- **Error**

Errors are typically ignored in code because you can rarely do anything about an error. For **example**, if stack overflow occurs, an error will arise. This type of error cannot be handled in the code.

Java Throw

- The throw keyword is used to throw an exception explicitly.
- Only object of Throwable class or its sub classes can be thrown.
- Program execution stops on encountering throw statement, and the closest catch statement is checked for matching type of exception.

Syntax :

```
throw ThrowableInstance
```

```
ex: throw new NullPointerException("test");
```

```
//EXaMPLE PROGRAM
```

```
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(); } }
```

RE-THROWING an Exception

```
class ThrowExcep
{
    static void fun()
    {
        try
        {
```

```
        throw new NullPointerException("demo");
    }
    catch(NullPointerException e)
    {
        System.out.println("Caught inside fun().");
        throw e; // rethrowing the exception
    }
}
```

```
public static void main(String args[])
{
    try
    {
        fun();
    }
    catch(NullPointerException e)
    {
```

```

        System.out.println("Caught in main.");
    }
}
}

```

- ➔ The above program illustrates about rethrowing an exception object
- ➔ It means--- an exception object is thrown from try block(nested) and inner catch handles it and again the inner catch block throws the same object which is handled by the outer catch
- ➔ throw statement can be written in catch if and only if that catch is a nested catch.

Java throws Keyword

- The throws keyword is used to declare the list of exception that a method may throw during execution of program.
- Any method that is capable of causing exceptions must list all the exceptions possible during its execution, so that anyone calling that method gets a prior knowledge about which exceptions are to be handled.
- A method can do so by using the throws keyword.

Syntax:

```

type method_name(parameter_list) throws exception_list
{
    // definition of method
}

```

```
}
```

Example throws Keyword

Here, we have a method that can throw Arithmetic exception so we mentioned that with the method declaration and catch that using the catch handler in the main method.

```
// Java program to demonstrate working of throws
```

```
class ThrowsExcep
```

```
{
```

```
    static void fun() throws IllegalAccessException
```

```
    {
```

```
        System.out.println("Inside fun(). ");
```

```
        throw new IllegalAccessException("demo");
```

```
    }
```

```
    public static void main(String args[])
```

```
    {
```

```
        try
```

```
        {
```

```
            fun();
```

```

    }
    catch(IllegalAccessException e)
    {
        System.out.println("caught in main.");
    }
}
}

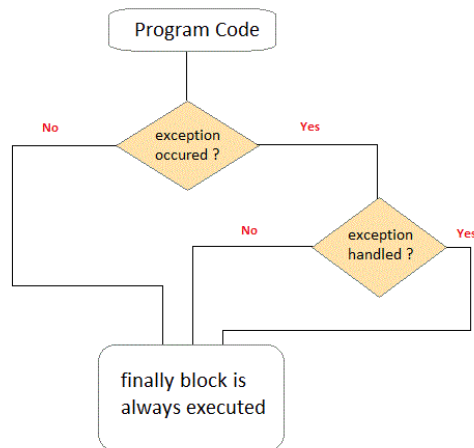
```

Difference between throw and throws

throw	throws
throw keyword is used to throw an exception explicitly.	throws keyword is used to declare an exception possible during its execution.
throw keyword is followed by an instance of Throwable class or one of its sub-classes.	throws keyword is followed by one or more Exception class names separated by commas.
throw keyword is declared inside a method body.	throws keyword is used with method signature (method declaration).
We cannot throw multiple exceptions using throw keyword.	We can declare multiple exceptions (separated by commas) using throws keyword.

finally block

- A finally keyword is used to create a block of code that follows a try block.
- A finally block of code is always executed whether an exception has occurred or not.
- Using a finally block lets you run any clean up type statements that you want to execute, no matter what happens in the protected code.
- A finally block appears at the end of catch block.



java finally example where exception doesn't occur.

```
class TestFinallyBlock{  
    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...");
}
}
```

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...");
    } }
```

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...");  
}  
}
```

//creating own exception sub-classes or user-defined exceptions

```
import java.util.*;  
  
class MyException extends Exception  
{  
    MyException()  
  
    {  
        System.out.println("PASSWORD should be 8 characters long");  
    }  
}  
  
class UserDefinedExceptionDemo  
{  
    static void check(int n)
```

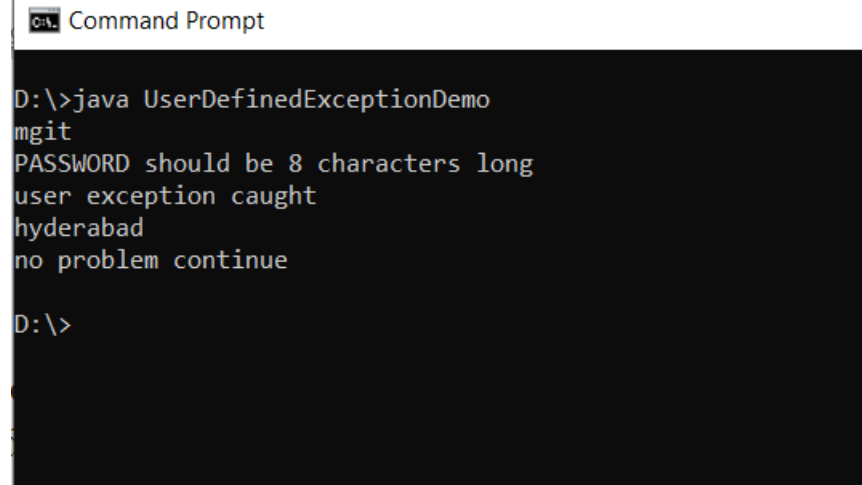


```
{  
    try  
    {  
        if(n<8)  
            throw new MyException();  
    }  
    else  
        System.out.println("no problem continue");  
    }  
    catch(MyException e)  
    {  
        System.out.println("user exception caught");  
    }  
}  
  
public static void main(String args[])  
{  
    Scanner s=new Scanner(System.in);  
    check(s.nextLine().length());  
}
```

```
check(s.nextLine().length());
```

```
}}
```

- The main use of throw keyword can be understood when we wish to throw user defined exceptions
- User defined exceptions are the exception objects which are not predefined but are created by the user.
- A user defined exception object is the object of a class which has to inherit **Exception** class so as to inherit the predefined properties of an exception object.
- The above program reads two strings and if the length of the string is less than 8 then it raises a user defined exception and if the string length is not less than 8 then no exception is raised.



```
C:\> Command Prompt

D:\>java UserDefinedExceptionDemo
mgit
PASSWORD should be 8 characters long
user exception caught
hyderabad
no problem continue

D:\>
```