Exception Handling in Java with Examples – 2024

Exception handling in java is one of the powerful mechanisms to handle runtime errors caused by exceptions. Exception handling plays an important role in software development. This article helps you understand java exception, exception in java, java exception handling, java exception hierarchy, types of exception in java, and many more.

What is Exception Handling in Java?

Exception handling in java helps in minimizing exceptions and helps in recovering from exceptions. It is one of the powerful mechanisms to handle runtime exceptions and makes it bug-free. Exception handling helps in maintaining the flow of the program. An exception handling is defined as an abnormal condition that may happen at runtime and disturb the normal flow of the program.

What is an Exception?

An expectation is an unexpected event that occurs while executing the program, that disturbs the normal flow of the code.

Exception handling in java with an example:

Let's say,

```
statement
statement
statement
exception …… an exception occurred, then JVM will handle it and
will exit the prog.
statement
statement

statement
```

For handling exceptions, there are **2 possible approaches**

1. JVM

If an exception is not handled explicitly, then JVM takes the responsibility of handling the exception.

Once the exception is handled, JVM will halt the program and no more execution of code will take place

- **Example**:

```
import java.util.*;

class Main {
    public static void main (String[] args) {
        System.out.println(5/0);
        System.out.println("End of program!");
     }
}
```

Runtime Error:

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

 at Main.main(File.java:5)
```

2. Developer

Developers can explicitly write the implementation for handling the exception. Once an exception is handled, the normal execution of code will continue.

Preferable: handle exceptions to ensure your code gets executed normally.

Java Exception Hierarchy

Exception Hierarchy – Following is the Exception Handling in Java handling hierarchy.

- **Throwable** –
    - It is the root class for the exception hierarchy in java.
    - It is in the java.lang package.
- **Error** –
    - Subclass of Throwable.
    - Consist of abnormal condition that is out of one's control and depends on the environment
    - They can't be handled and will always result in the halting of the program.
    - Eg: StackOverFlowError that can happen in infinite loop or recursion
- **Exception** –
    - Subclass of Throwable.
    - Consist of abnormal conditions that can be handled explicitly.
    - If one handles the exception then our code will continue to execute smoothly.

Types of exception in Java

- **Checked Exceptions**
    - Those exceptions that are checked at compile-time comprises checked exceptions.
    - They are child classes of Exception except for RuntimeException.
    - The program will not compile if they are not handled.
    - Example: IOException, ClassNotFoundException, etc.
    -
- **Unchecked Exceptions**
    - Those exceptions that are checked at runtime comprises unchecked exceptions.
    - They are child classes of RuntimeException.
    - They give runtime errors if not handled explicitly.
    - Example: ArithmeticException, NullPointerException etc.

Difference between Checked and Unchecked Exception

Checked Exceptions	Unchecked Exceptions
Occur at compile time.	Occur at runtime.
The compiler checks for a checked exception.	The compiler doesn’t check for exceptions.
Can be handled at the compilation time.	Can’t be caught or handled during compilation time.
The JVM requires that the exception be caught and handled.	The JVM doesn’t require the exception to be caught and handled.
Example of Checked exception- ‘File Not Found Exception’	Example of Unchecked Exceptions- ‘No Such Element Exception’

Java Exception Index

Java Exception Keywords

Exception Handling in java is managed via five keywords: try, catch, throw, throws, and finally. Here are 5 keywords that are used in handling exceptions in Java

Keyword	Description

try	This keyword is used to specify a block and this block must be followed by either catch or finally. That is, we can't use try block alone.
catch	This keyword must be preceded by a try block to handle the exception and can be followed by a final block later.
finally	This keyword is used to execute the program, whether an exception is handled or not.
throw	This keyword is used to throw an exception.
throws	This keyword is used to declare exceptions.

Java Try-Catch Block

Try-catch syntax:

```
try{
}
catch(Exception e){

}
```

- **Try-catch Example:**

```
public class ExceptionDemo {
     public static void main (String[] args) {
          int a=10;
          for(int i=3;i>=0;i--)
             try{
               System.out.println(a/i);
```

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

Output:

```
3
5
10

java.lang.ArithmeticException: / by zero
```

- **try** block contains the code that might throw an exception. Don't write anything extra in try as statements after the exception will not get executed if the exception occurred. Try must be immediately followed by catch or finally block.

```
public class ExceptionDemo {
     public static void main (String[] args) {
          int a=10;
          for(int i=3;i>=0;i--)
             try{
               System.out.println(a/i);
             }
     }
}
```

Compile-time error:

```
prog.java:5: error: 'try' without 'catch', 'finally' or resource
declarations
    try{
    ^

1 error
```

- The catch block is used to catch the exception thrown by statements in the try block. The catch must follow try else it will give a compile-time error.

```
public class ExceptionDemo {
     public static void main (String[] args) {
          int a=10;
          for(int i=3;i>=0;i--)
             try{
               System.out.println(a/i);
             }
             System.out.println("between try and catch");
             catch(ArithmeticException e){
                 System.out.println(e);
             }
     }
}
```

Compile Time Error:

```
prog.java:5: error: 'try' without 'catch', 'finally' or resource
declarations
    try{
    ^
prog.java:9: error: 'catch' without 'try'
    catch(ArithmeticException e){
    ^

2 errors
```

Things to Remember:

Do not keep any code after the statement which is prone to exception. Because if an exception occurred, it will straight away jump to the catch or finally block, ignoring all other statements in the try block.

```
class Main {
     public static void main (String[] args) {
         try
       {
             System.out.println(4/0);
```

```
       //will not get printed
             System.out.println("end of try!");
         }
catch(ArithmeticException e)
         {
             System.out.println("divide by 0");
         }
    }
}
```

Output:

```
divide by 0
```

- While catching the exception in the catch block, either you can have directly the class of exception or its superclass.

Example: Exact Exception

```
class Main {
     public static void main (String[] args) {
        try{
            System.out.println(4/0);
           }

        //ArithmeticException
        catch(ArithmeticException e){
            System.out.println("divide by 0");
        }
    }
}
```

Output:

```
divide by 0
```

Example: Superclass of Exact Exception

```
class Main {
```

```
      public static void main (String[] args) {
         try{
             System.out.println(4/0);
            }

         //superclass of ArithmeticException
         catch(Exception e){
             System.out.println("divide by 0");
         }
      }
}
```

Output:

```
divide by 0
```

Java Multiple Catch Block

If you have multiple catches, you have to maintain the hierarchy from subclass to superclass.

Incorrect:

```
class Main {
      public static void main (String[] args) {
         try{
             System.out.println(4/0);
         }catch(Exception e)
         {
             System.out.println("Exception : divide by 0");
         }catch(ArithmeticException e)
         {
             System.out.println("ArithmeticException :divide by
0");
         }
      }
}
```

Compile-time error:

```
prog.java:11: error: exception ArithmeticException has already
been caught
        }catch(ArithmeticException e)
         ^

1 error
```

Correct:

```
class Main {
     public static void main (String[] args) {
        try{
            System.out.println(4/0);
        }catch(ArithmeticException e)
        {
            System.out.println("ArithmeticException : divide by
0");
        }catch(Exception e)
        {
            System.out.println("Exception : divide by 0");
        }
   }
}
```

Output:

```
ArithmeticException: Divide by 0
```

Java Nested Try

When there is another try block within the try block:

```
class Main {
     public static void main (String[] args) {
        try{
                try{
                    int[] a={1,2,3};
                    System.out.println(a[3]);
                }
   catch(ArrayIndexOutOfBoundsException e)
                {
```

```
                    System.out.println("Out of bounds");
                }
              System.out.println(4/0);
        }
       catch(ArithmeticException e)
        {
            System.out.println("ArithmeticException : divide by
0");
        }
     }
    }
```

Output:

```
Out of bounds

ArithmeticException: Divide by 0
```

Note – If we put code of outer try before inner try, then if an exception occurred, it will ignore the entire inner try and move directly to its catch block.

```
class Main {
     public static void main (String[] args) {
       try{
              System.out.println(4/0);
              try{
                   int[] a={1,2,3};
                   System.out.println(a[3]);
               }
   catch(ArrayIndexOutOfBoundsException e)
               {
                   System.out.println("Out of bounds");
               }
        }
       catch(ArithmeticException e)
        {
            System.out.println("ArithmeticException : divide by
0");
        }
     }
    }
```

Output:

```
ArithmeticException: Divide by 0
```

Java Finally Block

Contains code that must be executed no matter if an exception is thrown or not. It contains code of file release, closing connections, etc.

- **Example:**

```
class Main {
     public static void main (String[] args) {
        try{
            System.out.println(4/0);
        }catch(Exception e)
        {
            System.out.println(e);
        }
        finally
        {
            System.out.println("finally executed");
        }

                    System.out.println("end");
     }
}
```

Output:

```
java.lang.ArithmeticException: / by zero
finally executed

end
```

Finally, will execute even when we do not handle exceptions. Before halting the program, JVM checks if there is a “finally” block.

```
class Main {
     public static void main (String[] args) {
        try{
            System.out.println(4/0);

        }finally
        {
            System.out.println("cleaning.......");
        }
     }
}
```

Runtime Error:

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

 at Main.main(File.java:4)
```

Output:

```
cleaning.......
```

Java Final vs Finally vs Finalize

Final	**Finally**	**Finalize**
Final is used to apply restrictions on class, method, and variable	Finally is used in coding, it will be executed whether an exception is handled or not.	Finalize is used to perform clean-up processing before garbage is collected.

Final is a keyword in java	Finally is a block in java	Finalize is a method in java
Final is executed upon its call.	Finally executes after”try-catch” block.	finalize executes just before the destruction of the object.

Java Throw Keyword

It is a keyword that is used to explicitly throw an exception.

We can use throw where according to our logic an exception should occur.

Example:

```
public class ExceptionDemo {
      static void canVote(int age){
            if(age<18)
             try{
                 throw new Exception();
             }catch(Exception e){
                 System.out.println("you are not an adult!");
             }
            else
               System.out.println("you can vote!");
      }
      public static void main (String[] args) {
            canVote(20);
            canVote(10);
      }
}
```

Output:

```
you can vote!

you are not an adult!
```

Java Throws Keyword

- Throws keyword is used when callee doesn't want to handle the exception rather it wants to extend this responsibility of handling the exception to the caller of the function.
- Basically says what sort of exception the code can throw and relies on the caller to handle it.
- It is used to handle checked Exceptions as the compiler will not allow code to compile until they are handled.

Example:

```
public class ExceptionDemo {
      static void func(int a) throws Exception{
              System.out.println(10/a);
      }
      public static void main (String[] args) {
            try{
                func(10);
                func(0);
            }catch(Exception e){
               System.out.println("can't divide by zero");
            }

      }
}
```

Output:

```
1

can't divide by zero
```

If callee can throw multiple exceptions, then all will be thrown simultaneously.

```
import java.util.*;
```

```
public class ExceptionDemo {
     static void func(int a,int b) throws ArithmeticException,
ArrayIndexOutOfBoundsException{
              System.out.println(10/a);
              int[] arr={1,2,3};
              System.out.println(arr[b]);
     }
     public static void main (String[] args) {
           Scanner in=new Scanner(System.in);
           for(int i=0;i<3;i++){
           try{
               func(in.nextInt(),in.nextInt());
           }catch(ArithmeticException e){
              System.out.println("can't divide by zero");
           }catch(ArrayIndexOutOfBoundsException e){
              System.out.println("Out of bounds!");
           }
                }

     }
   }
```

Input:

```
2 1
0 1

2 3
```

Output:

```
5
2
can't divide by zero
5

Out of bounds!
```

Java Throw vs Throws

Throw	Throws
This keyword is used to explicitly throw an exception.	This keyword is used to declare an exception.
A checked exception cannot be propagated with throw only.	A checked exception can be propagated with throws.
The throw is followed by an instance and used with a method	Throws are followed by class and used with the method signature.
You cannot throw multiple exceptions.	You can declare multiple exceptions

Java Custom Exception

You can create your own exception and give implementation as to how it should behave. Your exception will behave like a child's class of Exception.

Syntax:

```
1  class YourException extends Exception{}
```

- **Example:**
    - let's say, you are working with an airline company
    - You are in the luggage check-in department and as per rules, you can allow 15kg per customer.
    - So now more than 15kg of weight is an abnormal condition for us or in other words its an exception

- This is our logic-based exception, so we'll create our custom exception WeightLimitExceeded
- As per syntax, it will extend Exception.
- We define the constructor which will get invoked as soon as an exception will be thrown
- We have to explicitly throw the exception and hence we will use throw keyword for that.
- Using throws keyword is as per our need. If we are handling an exception where it is getting thrown then we can avoid throws, else we will use throws and handle it in the caller.

Implementation:

```
import java.util.*;

class WeightLimitExceeded extends Exception{
    WeightLimitExceeded(int x){
        System.out.print(Math.abs(15-x)+" kg : ");
    }
}


class Main {
    void validWeight(int weight) throws WeightLimitExceeded{
        if(weight>15)
            throw new WeightLimitExceeded(weight);
        else
            System.out.println("You are ready to fly!");
    }

      public static void main (String[] args) {
        Main ob=new Main();
        Scanner in=new Scanner(System.in);
        for(int i=0;i<2;i++){
            try{
                ob.validWeight(in.nextInt());
            }catch(WeightLimitExceeded e){
                System.out.println(e);
            }
        }

     }
}
```

Input:

```
20

7
```

Output:

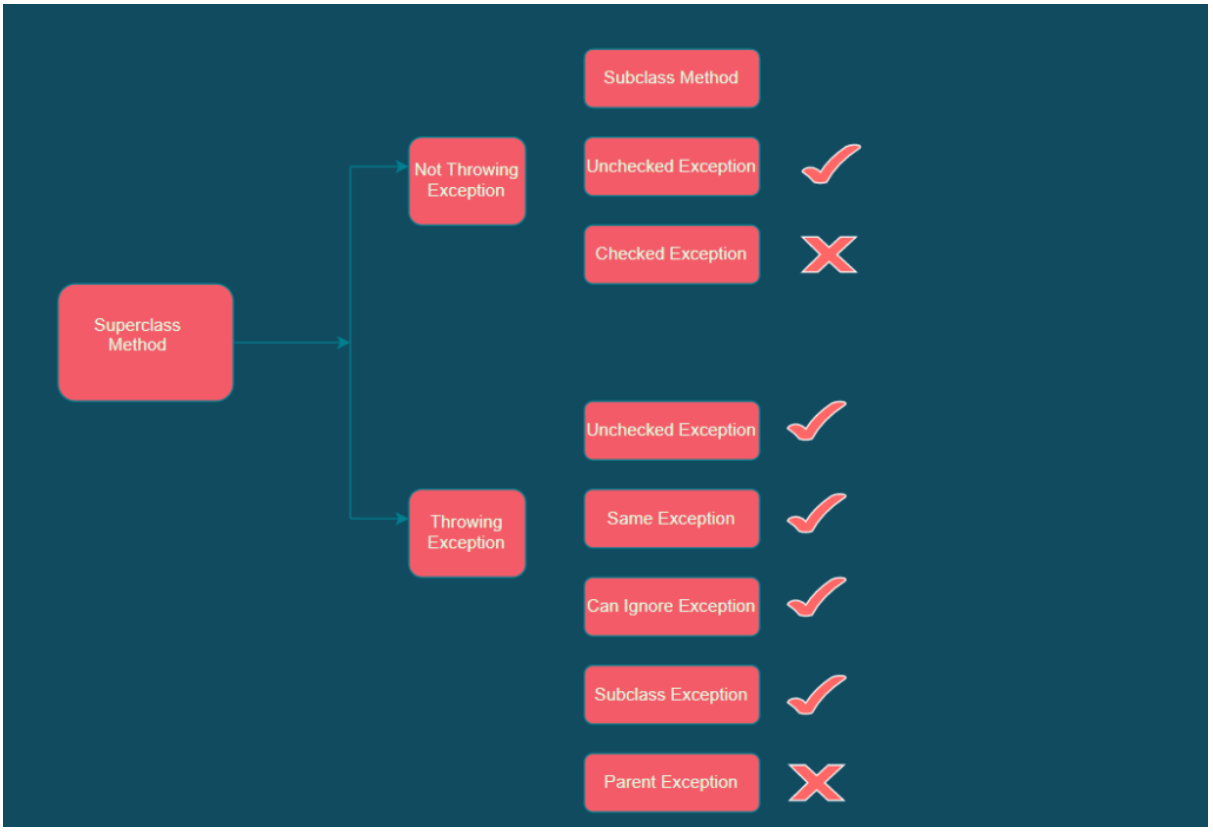
```
5 kg : WeightLimitExceeded

You are ready to fly!
```

Exception Handling in java with method overriding

Exception Handling in Java with Method Overriding is an overridden method that declares to throw an exception and declare that it can throw the same exception or subtype of that exception.

To handle the exception in Java, you will have to follow three important rules. They are depicted in the below figure.



Exception Handling in Java with Method Overriding

Advantages and disadvantages of exception handling in java

Advantages of excepting handling in java

- Separating Error-Handling Code from “Regular” Code
- Propagating Errors Up the Call Stack
- Grouping and Differentiating Error Types

Disadvantages of excepting handling in java

- Experiencing unnecessary overhead
- Not understanding how the application really works
- Filling your logs with noisy events
- Inability to focus on what actually matters