

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.

In this page, we will learn about Java exceptions, its type and the difference between checked and unchecked exceptions.

What is Exception in Java

Dictionary Meaning: Exception is an abnormal condition.

What is Exception Handling

Exception Handling is a mechanism to handle runtime errors such as `ClassNotFoundException`, `IOException`, `SQLException`, `RemoteException`, etc.

Advantage of Exception Handling

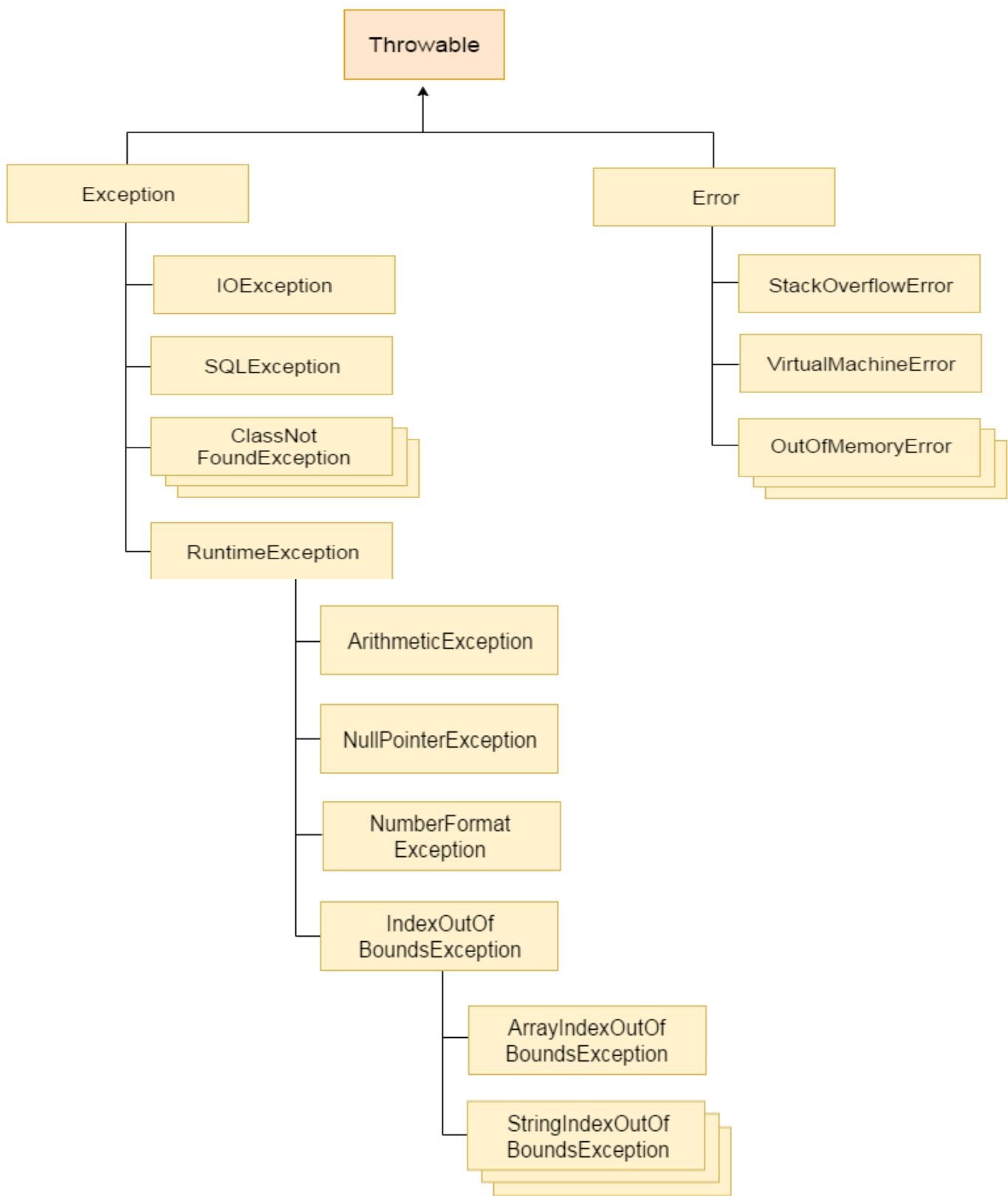
The core advantage of exception handling is **to maintain the normal flow of the application**. An exception normally disrupts the normal flow of the application that is why we use exception handling. Let's take a scenario:

```
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, the rest of the code will not be executed i.e. statement 6 to 10 will not be executed. If we perform exception handling, the rest of the statement will be executed. That is why we use exception handling in Java.

Hierarchy of Java Exception classes

The `java.lang.Throwable` class is the root class of Java Exception hierarchy which is inherited by two subclasses: `Exception` and `Error`. A hierarchy of Java Exception classes are given below:



Types of Java Exceptions

There are mainly two types of exceptions: checked and unchecked. Here, an error is considered as the unchecked exception. According to Oracle, 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 which directly inherit 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 which inherit RuntimeException are known as unchecked exceptions e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

3) Error

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

Java Exception Keywords

There are 5 keywords which are used in handling exceptions in Java.

| Keyword | Description |
|---------|---|
| try | The "try" keyword is used to specify a block where we should place exception code. The try block must be followed by either catch or finally. It means, we can't use try block alone. |
| catch | The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later. |
| finally | The "finally" block is used to execute the important code of the program. It is executed whether an exception is handled or not. |
| throw | The "throw" keyword is used to throw an exception. |

Java Exception Handling Example

Let's see an example of Java Exception Handling where we using a try-catch statement to handle the exception.

```
public class JavaExceptionExample{
    public static void main(String args[]){
        try{
            //code that may raise exception
            int data=100/0;
        }catch(ArithmeticException e){System.out.println(e);}
        //rest code of the program
        System.out.println("rest of the code...");
    }
}
```

✓ Test it Now

Output:

```
Exception in thread main java.lang.ArithmeticException:/ by zero
rest of the code...
```

Common Scenarios of Java Exceptions

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

1) A scenario where ArithmeticException occurs

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

```
int a=50/0;//ArithmeticException
```

2) A scenario where NullPointerException occurs

If we have a null value in any variable, performing any operation on the variable throws a NullPointerException.

```
String s=null;
System.out.println(s.length());//NullPointerException
```

3) A scenario where NumberFormatException occurs

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

```
String s="abc";
int i=Integer.parseInt(s);//NumberFormatException
```

4) A scenario where ArrayIndexOutOfBoundsException occurs

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

```
int a[]=new int[5];
a[10]=50; //ArrayIndexOutOfBoundsException
```

Java try block

Java **try** block is used to enclose the code that might throw an exception. It must be used within the method.

If an exception occurs at the particular statement of try block, the rest of the block code will not execute. So, it is recommended not to keeping the code in try block that will not throw an exception.

Java try block must be followed by either catch or finally block.

Syntax of Java try-catch

```
try{
//code that may throw an exception
}catch(Exception_class_Name ref){}
```

Syntax of try-finally block

```
try{
//code that may throw an exception
}finally{}
```

Java catch block

Java catch block is used to handle the Exception by declaring the type of exception within the parameter. The declared exception must be the parent class exception (i.e., Exception) or the generated exception type. However, the good approach is to declare the generated type of exception.

The catch block must be used after the try block only. You can use multiple catch block with a single try block.

Problem without exception handling

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

Example 1

```
public class TryCatchExample1 {


    public static void main(String[] args) {

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

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

    }

}
```

 **Test it Now**

Output:

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

As displayed in the above example, the **rest of the code** is not executed (in such case, the **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 the above problem by a java try-catch block.

Example 2

```
public class TryCatchExample2 {  
  
    public static void main(String[] args) {  
        try  
        {  
            int data=50/0; //may throw exception  
        }  
        //handling the exception  
        catch(ArithmeticException e)  
        {  
            System.out.println(e);  
        }  
        System.out.println("rest of the code");  
    }  
}
```

 Test it Now

Output:

```
java.lang.ArithmeticException: / by zero  
rest of the code
```

Here, we handle the exception using the parent class exception.

```
public class TryCatchExample4 {  
  
    public static void main(String[] args) {  
        try  
        {  
            int data=50/0; //may throw exception  
        }  
        // handling the exception by using Exception class  
        catch(Exception e)  
        {  
            System.out.println(e);  
        }  
        System.out.println("rest of the code");  
    }  
}
```

 **Test it Now**

Output:

```
java.lang.ArithmeticException: / by zero  
rest of the code
```

Let's see an example to print a custom message on exception.

```
public class TryCatchExample5 {  
  
    public static void main(String[] args) {  
        try  
        {  
            int data=50/0; //may throw exception  
        }  
        // handling the exception  
        catch(Exception e)  
        {  
            // displaying the custom message  
            System.out.println("Can't divided by zero");  
        }  
    }  
}
```

 **Test it Now**

Output:

```
Can't divided by zero
```



```
public class TryCatchExample9 {  
  
    public static void main(String[] args) {  
        try  
        {  
            int arr[] = {1,3,5,7};  
            System.out.println(arr[10]); //may throw exception  
        }  
        // handling the array exception  
        catch(ArrayIndexOutOfBoundsException e)  
        {  
            System.out.println(e);  
        }  
        System.out.println("rest of the code");  
    }  
}
```

 **Test it Now**

Output:

```
java.lang.ArrayIndexOutOfBoundsException: 10  
rest of the code
```

Java Multi-catch block

A try block can be followed by one or more catch blocks. Each catch block must contain a different exception handler. So, if you have to perform different tasks at the occurrence of different exceptions, use java multi-catch block.

Points to remember

- At a time only one exception occurs and at a time only one catch block is executed.
- All catch blocks must be ordered from most specific to most general, i.e. catch for `ArithmeticException` must come before catch for `Exception`.

```
public class MultipleCatchBlock1 {  
  
    public static void main(String[] args) {  
  
        try{  
            int a[]=new int[5];  
            a[5]=30/0;  
        }  
        catch(ArithmeticException e)  
        {  
            System.out.println("Arithmetic Exception occurs");  
        }  
        catch(ArrayIndexOutOfBoundsException e)  
        {  
            System.out.println("ArrayIndexOutOfBoundsException occurs");  
        }  
        catch(Exception e)  
        {  
            System.out.println("Parent Exception occurs");  
        }  
        System.out.println("rest of the code");  
    }  
}
```

```
Arithmetic Exception occurs  
rest of the code
```

Example 2

```
public class MultipleCatchBlock2 {  
  
    public static void main(String[] args) {  
  
        try{  
            int a[]=new int[5];  
  
            System.out.println(a[10]);  
        }  
        catch(ArithmeticException e)  
        {  
            System.out.println("Arithmetic Exception occurs");  
        }  
        catch(ArrayIndexOutOfBoundsException e)  
        {  
            System.out.println("ArrayIndexOutOfBoundsException occurs");  
        }  
        catch(Exception e)  
        {  
            System.out.println("Parent Exception occurs");  
        }  
        System.out.println("rest of the code");  
    }  
}
```

Output:

```
ArrayIndexOutOfBoundsException occurs  
rest of the code
```

Example 3

In this example, try block contains two exceptions. But at a time only one exception occurs and its corresponding catch block is invoked.

```
public class MultipleCatchBlock3 {  
  
    public static void main(String[] args) {  
  
        try{  
            int a[]=new int[5];  
            a[5]=30/0;  
            System.out.println(a[10]);  
        }  
        catch(ArithmeticException e)  
        {  
            System.out.println("Arithmetic Exception occurs");  
        }  
        catch(ArrayIndexOutOfBoundsException e)  
        {  
            System.out.println("ArrayIndexOutOfBoundsException occurs");  
        }  
        catch(Exception e)  
        {  
            System.out.println("Parent Exception occurs");  
        }  
        System.out.println("rest of the code");  
    }  
}
```

Output:

```
Arithmetic Exception occurs  
rest of the code
```

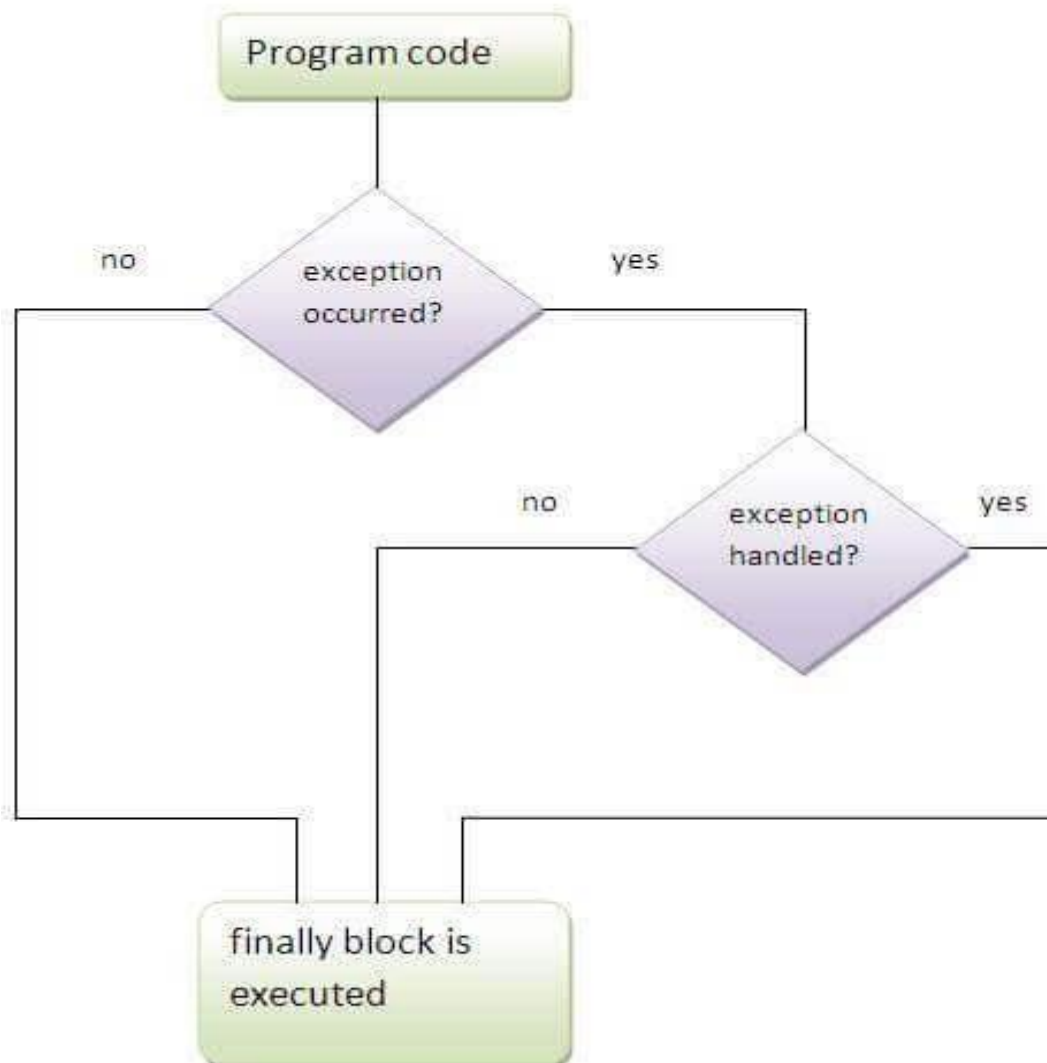
Java finally block



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.



Why use java finally

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

Case 1

Let's see the 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...");  
    }  
}
```

 **Test it Now**

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...");  
    }  
}
```

 **Test it Now**

```
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...");  
    }  
}
```

 **Test it Now**

```
Output:Exception in thread main java.lang.ArithmeticException:/ by zero  
        finally block is always executed  
        rest of the code...
```

Java throw keyword

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

We can throw either checked or unchecked 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;
```

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...");  
    }  
}
```

☒ Test it Now

Output:

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


Difference between final, finally and finalize

[< prev](#)[next >](#)

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

```
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();
}}
```