# **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 in Java**

**Dictionary Meaning:** Exception is an abnormal condition.

In Java, an 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 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;

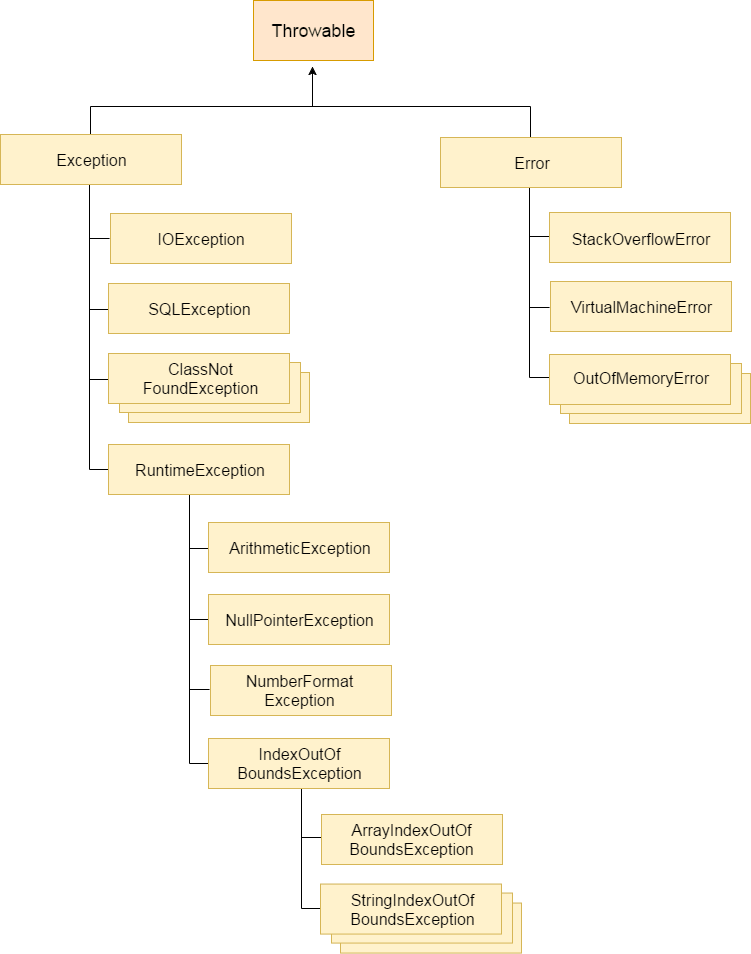
statement 9;

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.](https://www.javatpoint.com/java-tutorial)

## **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

#### **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.

#### **Unchecked Exception**

The classes which inherit RuntimeException are known as unchecked exceptions e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc. Unchecked exceptions are not checked at compiletime, but they are checked at runtime.

#### **Error**

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

**Exception handling keywords**

**The try Keyword**

The try keyword is used to define a block of code that will be tested for exceptions during its execution. If an exception occurs within the try block, it is thrown.

Syntax:

try

{

// Code that might throw an exception

}

Example:

public class TryExample

{

public static void main(String[] args)

{

try

{

int result = 10 / 0; // This will throw ArithmeticException

}

}

}

**Explanation:**

The try block contains code that might throw an exception. If an exception occurs, it will be caught by the corresponding catch block.

**2. The catch Keyword**

The catch keyword is used to handle the exception that occurs in the associated try block. You can have multiple catch blocks to handle different types of exceptions.

Syntax:

try

{

// Code that might throw an exception

}

catch (ExceptionType e)

{

// Code to handle the exception

}

Example:

public class CatchExample

{

public static void main(String[] args)

{

try

{

int result = 10 / 0; // This will throw ArithmeticException

}

catch (ArithmeticException e)

{

System.out.println("Caught exception: " + e.getMessage());

}

}

}

**Output:**

Caught exception: / by zero

Explanation:

The catch block handles the ArithmeticException thrown by the try block.

**3. The finally Keyword**

The finally keyword is used to define a block of code that will always execute, regardless of whether an exception was thrown or caught. It is typically used for resource cleanup.

**Syntax:**

try

{

// Code that might throw an exception

}

catch (ExceptionType e)

{

// Code to handle the exception

}

finally

{

// Code that will always execute

}

Example:

public class FinallyExample

{

public static void main(String[] args)

{

try

{

int result = 10 / 0; // This will throw ArithmeticException

}

catch (ArithmeticException e)

{

System.out.println("Caught exception: " + e.getMessage());

}

finally

{

System.out.println("Finally block executed.");

}

}

}

**Output:**

Caught exception: / by zero

Finally block executed.

Explanation:

The finally block is executed after the catch block, regardless of whether an exception was thrown.

**4. The throw Keyword**

The throw keyword is used to explicitly throw an exception. You can throw both built-in and custom exceptions using this keyword.

**Syntax:**

throw new ExceptionType("Exception message");

Example:

public class ThrowExample

{

public static void main(String[] args)

{

try

{

validateAge(15); // This will throw an IllegalArgumentException

}

catch (IllegalArgumentException e)

{

System.out.println("Caught exception: " + e.getMessage());

}

}

public static void validateAge(int age)

{

if (age < 18)

{

throw new IllegalArgumentException("Age must be 18 or older.");

}

System.out.println("Age is valid.");

}

}

**Output:**

Caught exception: Age must be 18 or older.

Explanation:

The validateAge method throws an IllegalArgumentException if the age is less than 18.

**5. The throws Keyword**

The throws keyword is used in a method signature to declare that the method might throw one or more exceptions. This informs the caller of the method about the exceptions it might throw, allowing them to handle these exceptions.

Syntax:

returnType methodName(parameterList) throws ExceptionType1, ExceptionType2

{

// Method body

}

Example:

import java.io.File;

import java.io.FileNotFoundException;

import java.io.FileReader;

public class ThrowsExample

{

public static void main(String[] args)

{

try

{

readFile("example.txt");

}

catch (FileNotFoundException e)

{

System.out.println("Caught exception: " + e.getMessage());

}

}

public static void readFile(String fileName) throws FileNotFoundException

{

File file = new File(fileName);

FileReader fr = new FileReader(file);

System.out.println("File read successfully");

}

}

**Output:**

Caught exception: example.txt (No such file or directory)

Explanation:

The readFile method declares that it throws a FileNotFoundException.

The caller of readFile must handle the exception, as shown in the main method.

**Multiple catch Blocks**

A method can have multiple catch blocks to handle different types of exceptions separately.

Example:

public class MultipleCatchExample

{

public static void main(String[] args)

{

try

{

int[] numbers = {1, 2, 3};

System.out.println(numbers[10]); // This will throw ArrayIndexOutOfBoundsException

int result = 10 / 0; // This will throw ArithmeticException

}

catch (ArrayIndexOutOfBoundsException e)

{

System.out.println("Array index out of bounds: " + e.getMessage());

}

catch (ArithmeticException e)

{

System.out.println("Arithmetic error: " + e.getMessage());

}

}

}

**Output:**

Array index out of bounds: Index 10 out of bounds for length 3

Explanation:

The first catch block handles ArrayIndexOutOfBoundsException.

The second catch block handles ArithmeticException.

Only the first exception that occurs (ArrayIndexOutOfBoundsException) is caught and handled.

**Nested try Blocks**

You can nest try blocks inside each other to handle exceptions that might occur within multiple levels of operations.

Example:

public class NestedTryExample

{

public static void main(String[] args)

{

try

{

System.out.println("Outer try block");

try

{

int result = 10 / 0; // This will throw ArithmeticException

}

catch (ArithmeticException e)

{

System.out.println("Inner catch: Arithmetic error: " + e.getMessage());

}

int[] numbers = {1, 2, 3};

System.out.println(numbers[10]); // This will throw ArrayIndexOutOfBoundsException

}

catch (ArrayIndexOutOfBoundsException e)

{

System.out.println("Outer catch: Array index out of bounds: " + e.getMessage());

}

finally

{

System.out.println("Outer finally block");

}

}

}

**Output:**

Outer try block

Inner catch: Arithmetic error: / by zero

Outer catch: Array index out of bounds: Index 10 out of bounds for length 3

Outer finally block

Explanation:

The inner try block handles the ArithmeticException.

The outer try block handles the ArrayIndexOutOfBoundsException.

The finally block is executed after both catch blocks.

**Chained Exception in Java**

Chained Exception was added to Java in JDK 1.4. **This feature allows you to relate one exception with another exception, i.e one exception describes cause of another exception.**

For example, consider a situation in which a method throws an ArithmeticException because of an attempt to divide by zero but the actual cause of exception was an I/O error which caused the divisor to be zero. The method will throw only ArithmeticException to the caller. So the caller would not come to know about the actual cause of exception. Chained Exception is used in such type of situations.

Two new constructors and two new methods were added to Throwable class to support chained exception.

Throwable(Throwable cause)

Throwable(String str, Throwable cause)

In the first constructor, the paramter cause specifies the actual cause of exception. In the second form, it allows us to add an exception description in string form with the actual cause of exception.

getCause() and initCause() are the two methods added to Throwable class.

getCause() method returns the actual cause associated with current exception.

initCause() set an underlying cause(exception) with invoking exception.

**Example:**

import java.io.IOException;

public class ChainedException

{

public static void divide(int a, int b)

{

if(b == 0)

{

ArithmeticException ae = new ArithmeticException("top layer");

ae.initCause(new IOException("cause"));

throw ae;

}

else

{

System.out.println(a/b);

}

}

public static void main(String[] args)

{

try

{

divide(5, 0);

}

catch(ArithmeticException ae)

{

System.out.println( "caught : " +ae);

System.out.println("actual cause: "+ae.getCause());

}

}

}

**Creating own exception sub class:**

**When to Create Custom Exceptions**

You should consider creating custom exceptions when:

1. You need to represent specific error conditions that are not covered by existing Java exceptions.
2. You want to provide more meaningful error messages and handling logic specific to your [application](https://www.javaguides.net/2018/06/guide-to-create-custom-exceptions.html).
3. You need to encapsulate additional information about the error condition.

**Steps to Create Custom Exceptions**

**Step 1: Define the Custom Exception Class**

Decide whether your custom exception should be a checked or unchecked exception. Then, create a class that extends Exception (for checked exceptions) or RuntimeException (for unchecked exceptions).

**Step 2: Provide Constructors**

Provide constructors in your custom exception class to accept error messages and optionally, the cause of the exception.

**Step 3: Throw the Custom Exception**

Throw your custom exception in your code where appropriate.

public class InvalidAgeException extends Exception

{

// Constructor that accepts an error message

public InvalidAgeException(String message)

{

super(message);

}

// Constructor that accepts an error message and a cause

public InvalidAgeException(String message, Throwable cause)

{

super(message, cause);

}

}