**National University of Computer & Emerging Sciences, Karachi**

**Computer Science Department**

**Fall 2024, Lab Manual - 02**

|  |  |
| --- | --- |
| **Course Code: SL3001** | **Course: Software Development and construction** |
| **Instructor:** | **Yasir Arfat** |

**Lab # 02**

**Exception Handling**

A Java exception is an object that describes an exceptional (that is, error) condition that has occurred in a piece of code. When an exceptional condition arises, an object representing that exception is created and thrown in the method that caused the error.

The **Exception Handling in Java** is one of the powerful mechanism to handle the runtime errors so that the normal flow of the application can be maintained.

**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 need to handle exceptions. Let's consider a scenario:

1. statement 1;
2. statement 2;
3. statement 3;
4. statement 4;
5. statement 5;//exception occurs
6. statement 6;
7. statement 7;
8. statement 8;
9. statement 9;
10. statement 10;

Suppose there are 10 statements in a Java program and an exception occurs at statement 5; the rest of the code will not be executed, i.e., statements 6 to 10 will not be executed. However, when we perform exception handling, the rest of the statements will be executed. That is why we use exception handling in [Java](https://www.javatpoint.com/java-tutorial)

## **Hierarchy of Java Exception classes**

A picture containing timeline

Description automatically generated

**Error vs Exceptions**

Chart, funnel chart

Description automatically generated

Diagram

Description automatically generated

### Types of Java Exceptions

Diagram

Description automatically generated

There are mainly two types of exceptions: checked and unchecked. An error is considered as the unchecked exception. However, according to Oracle, there are three types of exceptions namely:

1. Checked Exception
2. Unchecked Exception
3. Error

## **Difference between Checked and Unchecked Exceptions**

### 1) Checked Exception

The classes that directly inherit the Throwable class except RuntimeException and Error are known as checked exceptions. For example, IOException, SQLException, etc. Checked exceptions are checked at compile-time.

### 2) Unchecked Exception

The classes that inherit the RuntimeException are known as unchecked exceptions. For example, ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException, etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

### 3) Error

Error is irrecoverable. Some example of errors are OutOfMemoryError, VirtualMachineError, AssertionError etc.

Diagram

Description automatically generated

## **Java Exception Keywords**

Diagram, timeline

Description automatically generated

**Java exception handling is managed via five keywords: try, catch, throw, throws, and finally. Briefly, here is how they work. Program statements that you want to monitor for exceptions are contained within a try block. If an exception occurs within the try block, it is thrown. Your code can catch this exception (using catch) and handle it in some rational manner. Systemgenerated exceptions are automatically thrown by the Java run-time system. To manually throw an exception, use the keyword throw. Any exception that is thrown out of a method must be specified as such by a throws clause. The "throws" keyword is used to declare exceptions. It specifies that there may occur an exception in the method. It doesn't throw an exception. It is always used with method signature.Any code that absolutely must be executed after a try block completes is put in a finally block.**

**Graphical user interface, text, application, email

Description automatically generated**

**Uncaught Exceptions**

Before you learn how to handle exceptions in your program, it is useful to see what happens when you don’t handle them

**public** **class** Uncaught\_Exception {

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

**int** d =0 ;

**int** a = 2/d;

System.***out***.println("hello nida");

}

}

When the Java run-time system detects the attempt to divide by zero, it constructs a new exception object and then throws this exception.

**Graphical user interface, text

Description automatically generated**

Notice how the **class name**, Uncaught\_Exception; the **method name**, main; the **filename**, Uncaught\_Exception.java; and the **line number**, 4, are all included in the simple stack trace. Also, notice that the **type of exception** thrown is a subclass of Exception called ArithmeticException, which more specifically describes what type of error happened.

Graphical user interface, text, application

Description automatically generated

Graphical user interface, application

Description automatically generated

**Try without catch**

**public** **class** Ex\_1 {

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

**int** i = 0;

**int** j = 0;

**try** {

**int** k = i/j;//critical statement inside catch

}

}}

Text

Description automatically generated

Graphical user interface, text

Description automatically generated with medium confidence

**Try with catch**

**public** **class** Ex\_1 {

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

**int** i = 0;

**int** j = 0;

**try** {

**int** k = i/j;

}

**catch**(ArithmeticException e) {

System.***out***.println(e);

}

System.***out***.println("hello");

System.***out***.println("hi");

}}

**Graphical user interface, application, Word

Description automatically generated**

**Multiple Try and catch with printStackTrace method**

**public** **class** Ex\_1 {

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

**int** i = 0;

**int** j = 0;

**try** {

**int** k = i/j;

}

**catch**(ArithmeticException e) {

System.***out***.println(e);}

**try** {

**int**[] n = **new** **int**[2];

n[3]=4;

} **catch** (Exception e) {

e.printStackTrace();

}}}

**Graphical user interface, text, application

Description automatically generated**

## **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.

1. **int** a=50/0;//ArithmeticException

### 2) A scenario where NullPointerException occurs

If we have a null value in any [variable](https://www.javatpoint.com/java-variables)

, performing any operation on the variable throws a NullPointerException.

1. String s=**null**;
2. System.out.println(s.length());//NullPointerException

3) A scenario where NumberFormatException occurs

If the formatting of any variable or number is mismatched, it may result into NumberFormatException. Suppose we have a [string](https://www.javatpoint.com/java-string)

variable that has characters; converting this variable into digit will cause NumberFormatException

1. String s="abc";
2. **int** i=Integer.parseInt(s);//NumberFormatException

4) A scenario where ArrayIndexOutOfBoundsException occurs

When an array exceeds to it's size, the ArrayIndexOutOfBoundsException occurs. there may be other reasons to occur ArrayIndexOutOfBoundsException. Consider the following statements

1. **int** a[]=**new** **int**[5];
2. a[10]=50; //ArrayIndexOutOfBoundsException

## **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.

A picture containing text

Description automatically generated

**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("ArrayIndexOutOfBounds Exception occurs");

                  }

**catch**(Exception e)

                  {

                   System.out.println("Parent Exception occurs");

                  }

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

    }

}

A screenshot of a computer

Description automatically generated with low confidence

**In this example, we generate NullPointerException, but didn't provide the corresponding exception type. In such case, the catch block containing the parent exception class Exception will invoked.**

**public** **class** Ex\_2 {

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

**try**{

String s=**null**;

System.***out***.println(s.length());

}

**catch**(ArrayIndexOutOfBoundsException e)

{

System.***out***.println("ArrayIndexOutOfBounds Exception occurs");

}

**catch**(Exception e)

{

System.***out***.println("Parent Exception occurs");

}

}

}

Graphical user interface, text, application, email

Description automatically generated

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

**public** **class** Ex\_2 {

**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("ArrayIndexOutOfBounds Exception occurs");

}

}

}

Graphical user interface, text, application

Description automatically generated

**an example, to handle the exception without maintaining the order of exceptions (i.e. from most specific to most general).**

**public** **class** Ex\_2 {

**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...");

}

}

Text

Description automatically generated

# **Java Nested try block**

Graphical user interface, text, application

Description automatically generated

### 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.

**public** **class** Ex\_2 {

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

**try**{

**try** {

**int** a;

a=30/0;

}

**catch**(ArithmeticException e){System.***out***.println("task1 is completed");}

}

**catch**(Exception e){System.***out***.println("common task completed");}

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

}

}

Graphical user interface, text, application, email

Description automatically generated

When any try block does not have a catch block for a particular exception, then the catch block of the outer (parent) try block are checked for that exception, and if it matches, the catch block of outer try block is executed.

**public** **class** Ex\_2 {

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

**try**{

**try** {

**int** a;

a=30/0;

}

**catch**(ArrayIndexOutOfBoundsException e){System.***out***.println("task1 is completed");}

}

**catch**(Exception e){System.***out***.println("common task completed");}

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

}

}

Graphical user interface, text, application

Description automatically generated with medium confidence

# **Java finally block**

Diagram

Description automatically generated

## **Why use Java finally block?**

* finally block in Java can be used to put "**cleanup**" code such as closing a file, closing connection, etc.
* The important statements to be printed can be placed in the finally block.

Graphical user interface

Description automatically generated with low confidence

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

### When an exception occurr but not handled by the catch block

**public** **class** Ex\_2 {

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

**try**{

**int** a;

a=30/0;}

**finally** {

System.***out***.println("in finally");

}

}

}

### Graphical user interface, text, application Description automatically generated

### the finally block is executed after the try block and then the program terminates abnormally.

## **Java throw keyword**

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

We specify the **exception** object which is to be thrown. The Exception has some message with it that provides the error description. These exceptions may be related to user inputs, server, etc.

We can throw either checked or unchecked exceptions in Java by throw keyword. It is mainly used to throw a custom exception.

Graphical user interface

Description automatically generated

**public** **class** Ex\_2 {

**public** **static** **void** validate(**int** age) {

**if**(age<18) {

//throw Arithmetic exception if not eligible to vote

**throw** **new** ArithmeticException("Person is not eligible to vote");

}

**else** {

System.***out***.println("Person is eligible to vote!!");

}

}

//main method

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

//calling the function

*validate*(13);

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

}

}

Graphical user interface, text, application

Description automatically generated

# **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 the normal flow of the program can be maintained.

Which exception should be declared?

**Ans:** Checked exception only, because:

* **unchecked exception:** under our control so we can correct our code.
* **error:** beyond our control. For example, we are unable to do anything if there occurs VirtualMachineError or StackOverflowError.

Graphical user interface, application

Description automatically generated

* **In case we declare the exception, if exception does not occur, the code will be executed fine.**
* **import** java.io.FileInputStream;
* **import** java.io.FileNotFoundException;
* **import** java.io.IOException;
* **import** java.util.Scanner;
* **public** **class** FileInputusingscanner {
* **public** **static** **void** main(String[] args) **throws** IOException {
* // **TODO** Auto-generated method stub
* FileInputStream f = **new** FileInputStream("new.txt");
* Scanner s = **new** Scanner(f);
* **while**(s.hasNext()) {
* System.***out***.println(s.nextLine());
* }
* }
* }

Graphical user interface, text, application, email

Description automatically generated

* **In case we declare the exception and the exception occurs, it will be thrown at runtime because throws does not handle the exception.**
* **When there is no such file named new.txt**

Text

Description automatically generated

Diagram, text

Description automatically generated

Graphical user interface, application

Description automatically generated

# **Java Custom Exception**

In Java, we can create our own exceptions that are derived classes of the **Exception** class. Creating our own Exception is known as custom exception or user-defined exception. Basically, Java custom exceptions are used to customize the exception according to user need.

1st step your user define exception must extends exception class

//2nd step exception is thrown using throw keyword

**class** MyException **extends** Exception{}

//1st step your user define exception must extends exception class

**public** **class** customexception {

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

**try** {

**throw** **new** MyException();

//2nd step exception is thrown using throw keyword

} **catch** (MyException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

Another example

**class** MyException **extends** Exception{

**public** **int** a;

**public** MyException(**int** a) {

**this**.a = a;

}

@Override

**public** String toString() {

**return** "MyException [a=" + a + "]";

} }

**public** **class** customexception {

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

**try** {

**throw** **new** MyException(3);

} **catch** (MyException e) {

System.***out***.println(e);

}}}

Graphical user interface, text, application, Word

Description automatically generated

Let's see a simple example of Java custom exception. In the following code, constructor of InvalidAgeException takes a string as an argument. This string is passed to constructor of parent class Exception using the super() method. Also the constructor of Exception class can be called without using a parameter and calling super() method is not mandatory.

// class representing custom exception

**class** InvalidAgeException **extends** Exception

{

**public** InvalidAgeException (String str)

{ // calling the constructor of parent Exception

**super**(str);

}

}

// class that uses custom exception InvalidAgeException

**class** TestCustomException1

{

// method to check the age

**static** **void** validate (**int** age) **throws** InvalidAgeException{

**if**(age < 18){

// throw an object of user defined exception

**throw** **new** InvalidAgeException("age is not valid to vote");

}

**else** {

System.***out***.println("welcome to vote");

}

}

// main method

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

{

**try**

{

// calling the method

*validate*(13);

}

**catch** (InvalidAgeException ex)

{

System.***out***.println("Caught the exception");

// printing the message from InvalidAgeException object

System.***out***.println("Exception occured: " + ex);

}

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

}

}

}