**PRACTICAL # 04.5**

**OBJECT:**

**Using Exceptions**

**THEORY:**

Exception is some sort of problem during the execution of a program. An exception can occur for many reasons, like:

* A user has entered invalid data.
* The JVM has run out of memory.
* A file that needs to be opened cannot be found.
* A network connection has been lost in the middle of communications.

Some of these exceptions are caused by user error, others by programmer error, and others by physical resources that have failed in some manner.

Three categories of exceptions:

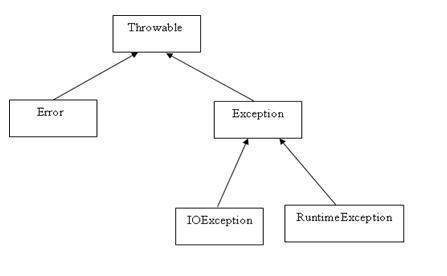
**Checked exceptions:** This exception is typically a user error or a problem that cannot be foreseen by the programmer. For example, if a file is to be opened, but the file cannot be found, an exception occurs. These exceptions cannot simply be ignored at the time of compilation.

**Runtime exceptions:** A runtime exception is an exception that occurs that probably could have been avoided by the programmer. As opposed to checked exceptions, runtime exceptions are ignored at the time of compilation.

**Errors:** These are not exceptions at all, but problems that arise beyond the control of the user or the programmer. Errors are typically ignored in your code because you can rarely do anything about an error. For example, if a stack overflow occurs, an error will arise. They are also ignored at the time of compilation.

## **Exception Hierarchy:**

The exception classes hierarchy is shown in the figure below.



**Fig1: Exception classes hierarchy**

Errors are not normally trapped form the Java programs. These conditions normally happen in case of severe failures, which are not handled by the Java programs. Errors are generated to indicate errors generated by the runtime environment. Example : JVM is out of Memory. Normally programs cannot recover from errors.

## Exceptions Methods:

Some of the important methods for exceptions:

**public String getMessage()**

Returns a detailed message about the exception. This message is initialized in the Throwable constructor.

**public Throwable getCause()**

Returns the cause of the exception as represented by a Throwable object.

**public String toString()**

Returns the name of the class concatenated with the result of getMessage()

**public void printStackTrace()**

Prints the result of toString() along with the stack trace to System.err, the error output stream.

## **Catching Exceptions:**

An exception is handled using a combination of the **try** and **catch** keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code.

**Syntax:**

try

{

//Protected code

}catch(ExceptionName e1)

{

//Catch block

}

A catch statement involves declaring the type of exception to catch. If an exception occurs in protected code, the catch block following the try is checked. If the type of exception that occurred is listed in a catch block, the exception is passed to the catch block.

try{

int a[] = new int[10];

System.out.println("Access element three :" + a[12]);

}catch(ArrayIndexOutOfBoundsException e){

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

}

A try block can be followed by multiple catch blocks. The exception is thrown to the first catch block in the list. If the data type of the exception thrown matches ExceptionType1, it gets caught there. If not, the exception passes down to the second catch statement. This continues until the exception either is caught or falls through all catches, in which case the current method stops execution and the exception is thrown down to the previous method on the call stack.

**Syntax:**

try

{

//Protected code

}catch(ExceptionType1 e1)

{

//Catch block

}catch(ExceptionType2 e2)

{

//Catch block

}catch(ExceptionType3 e3)

{

//Catch block

}

**Example:**

try

{

file = new FileInputStream(fileName);

x = (byte) file.read();

}catch(IOException i)

{

i.printStackTrace();

return -1;

}catch(FileNotFoundException f) //Not valid!

{

f.printStackTrace();

return -1;

}

## **throws/throw Keywords**

If a method does not handle a checked exception, the method must declare it using the **throws** keyword. The throws keyword appears at the end of method signature.

Method can throw an exception, either newly instantiated one or an exception that just caught, by using the **throw** keyword.

The following method declares that it throws a RemoteException:

import java.io.\*;

public class className

{

public void deposit(double amount) throws RemoteException

{

// Method implementation

throw new RemoteException();

}

//Remainder of class definition

}

A method can throw more than one exception, declared in a comma separated list.

import java.io.\*;

public class className

{

public void withdraw(double amount) throws RemoteException,

InsufficientFundsException

{

// Method implementation

}

//Remainder of class definition

}

**finally Keyword**

The finally keyword creates a block that follows a try block. A finally block of code always executes, whether or not an exception has occurred.

Using a finally block is used to run cleanup-type statements, no matter what happens in the protected code.

A finally block appears at the end of the catch blocks and has the following syntax:

try

{

//Protected code

}catch(ExceptionType1 e1)

{

//Catch block

}catch(ExceptionType2 e2)

{

//Catch block

}catch(ExceptionType3 e3)

{

//Catch block

}finally

{

//The finally block always executes.

}

**Example:**

public class ExcepTest{

public static void main(String args[]){

int a[] = new int[2];

try{

System.out.println("Access element three :" + a[3]);

}catch(ArrayIndexOutOfBoundsException e){

System.out.println("Exception thrown :" + e);

}

finally{

a[0] = 6;

System.out.println("First element value: " +a[0]);

System.out.println("The finally statement is executed");

}

}

}

* A catch clause cannot exist without a try statement.
* It is not compulsory to have finally clauses when ever a try/catch block is present.
* The try block cannot be present without either catch clause or finally clause.
* Any code cannot be present in between the try, catch, finally blocks.

**Declaring you own Exception:**

You can create your own exceptions in Java. Keep the following points in mind when writing your own exception classes:

All exceptions inherit Throwable.

If you want to write a checked exception that is automatically enforced by the Handle or Declare Rule, you need to extend the Exception class.

To write a runtime exception, you need to extend the RuntimeException class.

We can define our own Exception class as below:

class MyException extends Exception{

}

Just extend the Exception class to create your own Exception class. These are considered to be checked exceptions. The following InsufficientFundsException class is a user-defined exception that extends the Exception class, making it a checked exception. An exception class is like any other class, containing useful fields and methods.

**Example:**

// File Name InsufficientFundsException.java

import java.io.\*;

public class InsufficientFundsException extends Exception

{

private double amount;

public InsufficientFundsException(double amount)

{

this.amount = amount;

}

public double getAmount()

{

return amount;

}

}

To demonstrate using our user-defined exception, the following CheckingAccount class contains a withdraw() method that throws an InsufficientFundsException.

// File Name CheckingAccount.java

import java.io.\*;

public class CheckingAccount

{

private double balance;

private int number;

public CheckingAccount(int number)

{

this.number = number;

}

public void deposit(double amount)

{

balance += amount;

}

public void withdraw(double amount) throws

InsufficientFundsException

{

if(amount <= balance)

{

balance -= amount;

}

else

{

double needs = amount - balance;

throw new InsufficientFundsException(needs);

}

}

public double getBalance()

{

return balance;

}

public int getNumber()

{

return number;

}

}

**Common Exceptions:**

In Java, it is possible to define two catergories of Exceptions and Errors.

**JVM Exceptions:** These are exceptions/errors that are exclusively or logically thrown by the JVM. **Examples:** NullPointerException, ArrayIndexOutOfBoundsException, ClassCastException,

**Programmatic exceptions:** These exceptions are thrown explicitly by the application or the API **programmers Examples:** IllegalArgumentException, IllegalStateException.

**ACTIVITIES**

**Activity 1**

Write the above program in text editor, assemble and link the program to generate executable file. Run the exe file and observe the output.

**Activity 2**

Write the same program in Emu8086 emulator code editor but with different operand values. Emulate and run the program to verify the outputs.

**REVIEW QUESTIONS**

1. Where are the remainder and the quotient found in case of 16 bit number division?
2. What will happen in division operation if dividend is 0 and what will happen if divisor is 0?
3. What will be the result of using uninitialized variable in program without properly initializing its value?
4. Why does the multiplication of two 16-bit numbers require for their product two registers instead of one? Give some suitable example.