LAB#9
Exception Handling

Objective:
Understand how runtime errors are being handled in Java.

Theory:
An exception is an abnormal condition that arises in a code sequence at run time. In other words, an exception is a run-time error. In computer languages that do not support exception handling, errors must be checked and handled manually—typically through the use of error codes, and so on. This approach is as cumbersome as it is troublesome. Java's exception handling avoids these problems and, in the process, brings run-time error management into the object oriented world.
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. That method may choose to handle the exception itself, or pass it on. Either way, at some point, the exception is caught and processed.
Java exception handling is managed via five keywords: *try, catch, throw, throws*, and *finally*.

Lab Task:
To guard against and handle a run-time error, simply enclose the code that you want to monitor inside a try block. Immediately following the try block, includes a catch clause that specifies the exception type that you wish to catch. To illustrate how easily this can be done, the following program includes a try block and a catch clause that processes the ArithmeticException generated by the division-by-zero error:

```
class ExceptionHandling {
public static void main(String args[]) {
int d, a;
try
{ // monitor a block of code.
  d = 0;
  a = 42 / d;
  System.out.println("This will not be printed.");
}
catch (ArithmeticException e)
{ // catch divide-by-zero error
  System.out.println("Division by zero. " + e);
}
finally {
  System.out.println("finally");
 }
System.out.println("After catch statement.");
}
}
```

Here is a sample program that creates and throws an exception. The handler that catches the exception rethrows it to the outer handler.

```
// Demonstrate throw.
class ThrowDemo {
static void demoproc() {
try {
throw new NullPointerException("demo");
} catch(NullPointerException e) {
System.out.println("Caught inside demoproc.");
throw e; // rethrow the exception
}
}
public static void main(String args[]) {
try {
demoproc();
} catch(NullPointerException e) {
System.out.println("Recaught: " + e);
}
}
}
```

Lab Task:

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;
import java.util.Scanner;
public class ValidateCNIC {
  public void method() {
 String CNIC;// = "42101-1234567-2";
        Scanner sc= new Scanner(System.in);
        System.out.println("Enter CNIC in the following pattern XXXXX-XXXXXXX-X");
        CNIC=sc.next();
  Pattern pattern = Pattern.compile("\\d{5}-\\d{7}-\\d{1}");
    Matcher matcher = pattern.matcher(CNIC);
    try
    {if (matcher.matches())
    System.out.println("CNIC Valid");
    else throw new InvalidCNICException("The CNIC you Entered is not valid");
    }
    catch(InvalidCNICException e) {
    System.out.println("The CNIC you Entered is not valid");
    }
```

```
    }
  public static void main(String[] args) {
 ValidateCNIC test = new ValidateCNIC();
 test.method();
  }
 }
public class InvalidCNICException extends Exception{
  public InvalidCNICException() {}
  InvalidCNICException(String s){ super(s);}}
```

Lab Assignment:

1. Design a program to implement calculator's basic functionality with an exception handler that deals with nonnumeric operands; then Design another program without using an exception handler to achieve the same objective. Your program should display a message that informs the user of the wrong operand type before exiting.

Conclusion: