

| **TITLE : User Defined Exception** |
| --- |

**AIM:**

Create a user defined exception subclass NumberException with necessary constructor and overridden toString method. Write a program which accepts a number from the user. It throws an object of the NumberException class if the number contains digit 3 otherwise it displays the appropriate message. On printing, the exception object should display an exception name, appropriate message for exception.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Expected OUTCOME of Experiment:**

**CO1:** Understand the features of object oriented programming compared with procedural approach with C++ and Java

**CO4:**Explore the interface, exceptions, multithreading, packages **\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Books/ Journals/ Websites referred:**

1.Ralph Bravaco , Shai Simoson , “Java Programming From the Group Up” Tata McGraw-Hill.

2.Grady Booch, Object Oriented Analysis and Design.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Pre Lab/ Prior Concepts:**

**Exception handling** in java is a powerful mechanism or technique that allows us to handle runtime errors in a program so that the normal flow of the program can be maintained. All the exceptions occur only at runtime. A syntax error occurs at compile time.

**Exception in Java:**

In general, an exception means a problem or an abnormal condition that stops a computer program from processing information in a normal way.

An exception in java is an object representing an error or an abnormal condition that occurs at runtime execution and interrupts (disrupts) the normal execution flow of the program.

An exception can be identified only at runtime, not at compile time. Therefore, it is also called runtime errors that are thrown as exceptions in Java. They occur while a program is running.

For example:

* If we access an array using an index that is out of bounds, we will get a runtime error named ArrayIndexOutOfBoundsException.
* If we enter a double value while the program is expecting an integer value, we will get a runtime error called InputMismatchException.

When JVM faces these kinds of errors or dividing an integer by zero in a program, it creates an exception object and throws it to inform us that an error has occurred.If the exception object is not caught and handled properly, JVM will display an error message and will terminate the rest of the program abnormally.

If we want to continue the execution of remaining code in the program, we will have to handle exception objects thrown by error conditions and then display a user-friendly message for taking corrective actions. This task is known as exception handling in java.

**Types of Exceptions in Java**

Basically, there are two types of exceptions in java API. They are:

1. Predefined Exceptions (Built-in-Exceptions)

2. Custom (User defined)Exceptions

**Predefined Exceptions:**

Predefined exceptions are those exceptions that are already defined by the Java system. These exceptions are also called built-in-exceptions.Java API supports exception handling by providing the number of predefined exceptions. These predefined exceptions are represented by classes in java.

When a predefined exception occurs, JVM (Java runtime system) creates an object of predefined exception class. All exceptions are derived from java.lang.Throwable class but not all exception classes are defined in the same package. All the predefined exceptions supported by java are organized as subclasses in a hierarchy under the Throwable class.

All the predefined exceptions are further divided into two groups:

1. Checked Exceptions: Checked exceptions are those exceptions that are checked by the java compiler itself at compilation time and are not under runtime exception class hierarchy. If a method throws a checked exception in a program, the method must either handle the exception or pass it to a caller method.

2. Unchecked Exceptions: Unchecked exceptions in Java are those exceptions that are checked by JVM, not by java compiler. They occur during the runtime of a program. All exceptions under the runtime exception class are called unchecked exceptions or runtime exceptions in Java.

**Custom exceptions:**

Custom exceptions are those exceptions that are created by users or programmers according to their own needs. The custom exceptions are also called user-defined exceptions that are created by extending the exception class.

So, Java provides the liberty to programmers to throw and handle exceptions while dealing with functional requirements of problems they are solving.

**Exception Handling Mechanism using Try-Catch block:**

The general syntax of try-catch block (exception handling block) is as follows:

**Syntax:**

try

{

// A block of code; // generates an exception

}

catch(exception\_class var)

{

// Code to be executed when an exception is thrown.

}

**Example:**

public class TryCatchEx

{

public static void main(String[] args)

{

System.out.println("11");

System.out.println("Before divide");

int x = 1/0;

System.out.println("After divide");

System.out.println("22");

}

}

**Output:**

11

Before divide

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

**Class Diagram:**

+--------------------+

| NumberException |

+--------------------+

| -message: String |

+--------------------+

| +NumberException() |

| +NumberException(message: String) |

| +toString(): String |

+--------------------+

|

|

|

V

+--------------------+

| MainClass |

+--------------------+

| +main(String[]): void |

+--------------------+

**Algorithm:**

**Start**.

Define the class NumberException which extends the Exception class:

* Initialize the constructor to accept a string message.
* Override the toString() method to return "NumberException: " concatenated with the message.

Define the class NumberCheck:

* Create a method checkNumber(String number):
  1. Check if the input number contains the digit '3'.
  2. If yes, throw a new NumberException with the message "The number contains the digit 3".
  3. If no, print "The number is valid and does not contain the digit 3".
* Define the main(String[] args) method:
  1. Read a number from the user.
  2. Call the checkNumber() method.
  3. Use a try-catch block to catch the NumberException.
  4. If the exception is caught, print the exception using toString().

**End**.

**Implementation details :**

1)NumberException.java

public class NumberException extends Exception {

// Constructor accepting a custom message

public NumberException(String message) {

super(message); // Call the superclass constructor with the custom message

}

// Overriding the toString() method

@Override

public String toString() {

return "NumberException: " + getMessage();

}

}

2)NumberCheck.java

import java.util.Scanner;

public class NumberCheck {

// Method to check if the number contains digit 3

public static void checkNumber(String number) throws NumberException {

if (number.contains("3")) {

throw new NumberException("The number contains the digit 3");

} else {

System.out.println("The number is valid and does not contain the digit 3");

}

}

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

System.out.print("Enter a number: ");

String inputNumber = sc.nextLine();

try {

checkNumber(inputNumber);

} catch (NumberException e) {

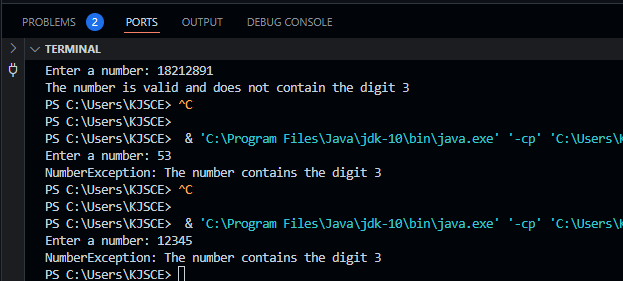
System.out.println(e); // This will invoke the overridden toString() method from NumberException class

}

}

}

**Output:**

****

**Conclusion:**

**Implemented user defined exception in java.**

**Date: \_\_\_\_\_\_\_\_\_\_\_ Signature of faculty in-charge**

**Post Lab Descriptive Questions**

1. Compare throw and throws.

**Purpose**:

* **throw**: Used to **explicitly throw** an exception in the program.
* **throws**: Used to **declare** that a method may throw certain exceptions.

**Usage Context**:

* **throw**: Appears inside a method or block of code where an exception is created and thrown.
* **throws**: Appears in the method signature, indicating potential exceptions that may be thrown by that method.

**Syntax**:

* **throw**: Followed by an instance of an exception (e.g., throw new NullPointerException();).
* **throws**: Followed by the exception type(s) (e.g., void methodName() throws IOException, SQLException).

**Exception Types**:

* **throw**: Can throw both checked and unchecked exceptions.
* **throws**: Typically used for declaring checked exceptions that must be handled or declared in the calling method.

**Flow Control**:

* **throw**: Interrupts the normal flow of the program and transfers control to the nearest catch block.
* **throws**: Provides information to the caller about what exceptions to expect, allowing them to handle or propagate those exceptions appropriately.

1. **Write program to implement following problem statement:**

**Create a User-Defined Exception:**

Define a custom exception class named InsufficientFundsException that extends the built-in Exception class.This exception should be thrown when a withdrawal request exceeds the available balance in the bank account.

**Bank Account Class:**

Create a class named BankAccount with the following attributes and methods:

private double balance (the current balance of the account)

A constructor to initialize the balance.

A method deposit(double amount) to add funds to the account.

A method withdraw(double amount) that throws the InsufficientFundsException if the amount to be withdrawn exceeds the current balance. Otherwise, it should deduct the amount from the balance.

A method getBalance() to return the current balance of the account.

**Main Class:**

In the main method of your application, demonstrate how to use the BankAccount class and handle the InsufficientFundsException.

Create a BankAccount object, perform a few deposits, and attempt to withdraw an amount that might cause an exception. Catch the InsufficientFundsException and print an appropriate error message.

// User-defined exception class

class InsufficientFundsException extends Exception {

public InsufficientFundsException(String message) {

super(message);

}

@Override

public String toString() {

return "InsufficientFundsException: " + getMessage();

}

}

class BankAccount {

// Private balance attribute

private double balance;

// Constructor to initialize the balance

public BankAccount(double initialBalance) {

this.balance = initialBalance;

}

// Method to deposit funds into the account

public void deposit(double amount) {

if (amount > 0) {

balance += amount;

System.out.println("Successfully deposited: $" + amount);

} else {

System.out.println("Invalid deposit amount.");

}

}

// Method to withdraw funds, throws InsufficientFundsException if funds are insufficient

public void withdraw(double amount) throws InsufficientFundsException {

if (amount > balance) {

throw new InsufficientFundsException("Withdrawal amount exceeds available balance.");

} else {

balance -= amount;

System.out.println("Successfully withdrew: $" + amount);

}

}

// Method to get the current balance

public double getBalance() {

return balance;

}

}

public class BankApplication {

public static void main(String[] args) {

// Create a BankAccount object with an initial balance

BankAccount account = new BankAccount(500);

// Perform a few deposits

account.deposit(200);

account.deposit(150);

// Display current balance

System.out.println("Current Balance: $" + account.getBalance());

try {

// Attempt to withdraw an amount that exceeds the current balance

account.withdraw(1000);

} catch (InsufficientFundsException e) {

// Catch and print the custom exception message

System.out.println(e);

}

// Display balance after failed withdrawal

System.out.println("Balance after failed withdrawal: $" + account.getBalance());

try {

// Attempt a valid withdrawal

account.withdraw(300);

} catch (InsufficientFundsException e) {

System.out.println(e);

}

// Display final balance

System.out.println("Final Balance: $" + account.getBalance());

}

}