**Java Exception Handling - Student Handbook**

**Table of Contents**

1. Introduction to Java Exception Handling
2. Exception Handling Structure in Java
3. **Exercises**
   * Exercise 1: Basic Try-Catch Block
   * Exercise 2: Multiple Exceptions Handling
   * Exercise 3: Custom Exception
   * Exercise 4: Throwing an Exception
   * Exercise 5: Nested Try-Catch
   * Exercise 6: Handling FileNotFoundException
4. Additional Resources
5. Conclusion

**1. Introduction to Java Exception Handling**

**What is Exception Handling?**

Exception handling in Java is a mechanism to handle runtime errors, ensuring the normal flow of the application. It helps in detecting and managing unexpected conditions that might arise during the execution of a program (like division by zero, file not found, etc.).

**Why Do We Need Exception Handling?**

* It allows for graceful recovery from errors, instead of crashing the program.
* It makes the program more robust and user-friendly.
* It separates error-handling code from regular code, making it easier to read and maintain.

**Commonly Used Exception Classes**

* ArithmeticException: For mathematical errors like division by zero.
* NullPointerException: When trying to access or modify an object that hasn’t been initialized.
* InputMismatchException: When the user provides input that does not match the expected type.
* FileNotFoundException: For errors related to file handling.
* IllegalArgumentException: For invalid arguments passed to a method.

**2. Exception Handling Structure in Java**

**Try-Catch Block**

The basic structure for exception handling involves the try and catch blocks.

try {

// Code that might throw an exception

} catch (ExceptionType e) {

// Code to handle the exception

}

* **try block**: Contains code that may cause an exception.
* **catch block**: Catches and handles the exception.

**Finally Block**

The finally block is optional and is used for code that should always run, whether an exception occurs or not (e.g., closing files or releasing resources).

```java

try {

// Potentially risky code

} catch (ExceptionType e) {

// Exception handling

} finally {

// Always runs, e.g., closing resources

}

```

**Throwing Exceptions**

You can throw exceptions manually using the throw keyword.

throw new Exception("This is a custom exception");

**Custom Exceptions**

You can define your own exception classes by extending the Exception class.

class CustomException extends Exception {

public CustomException(String message) {

super(message);

}

}

**3. Exercises**

**Exercise 1: Basic Try-Catch Block**

**Objective**: Handle division by zero using a try-catch block.

**Instructions:**

1. Write a program that asks the user to input two integers: numerator and denominator.
2. Use a try-catch block to catch any ArithmeticException (division by zero) and print a message: "Denominator cannot be zero."

**Solution:**

import java.util.Scanner;

public class BasicExceptionHandling {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

System.out.print("Enter the numerator: ");

int numerator = scanner.nextInt();

System.out.print("Enter the denominator: ");

int denominator = scanner.nextInt();

try {

int result = numerator / denominator;

System.out.println("Result: " + result);

} catch (ArithmeticException e) {

System.out.println("Denominator cannot be zero. Please provide a valid denominator.");

} finally {

scanner.close();

}

}

}

**Exercise 2: Multiple Exceptions Handling**

**Objective**: Handle both ArithmeticException and InputMismatchException for user input.

**Instructions:**

1. Write a program that accepts two numbers from the user.
2. Handle:
   * ArithmeticException if division by zero occurs.
   * InputMismatchException if the user inputs non-integer values.

**Solution:**

import java.util.InputMismatchException;

import java.util.Scanner;

public class MultipleExceptions {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

try {

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

int num1 = scanner.nextInt();

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

int num2 = scanner.nextInt();

int result = num1 / num2;

System.out.println("Result: " + result);

} catch (ArithmeticException e) {

System.out.println("Denominator cannot be zero.");

} catch (InputMismatchException e) {

System.out.println("Invalid input. Please enter integers.");

} finally {

scanner.close();

}

}

}

**Exercise 3: Custom Exception**

**Objective**: Define and throw a custom exception.

**Instructions:**

1. Create a custom exception InvalidAgeException that extends Exception.
2. In the main method, ask the user for their age.
3. If the age is less than 18 or greater than 120, throw and catch the InvalidAgeException.

**Solution:**

import java.util.Scanner;

class InvalidAgeException extends Exception {

public InvalidAgeException(String message) {

super(message);

}

}

public class CustomExceptionHandling {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

System.out.print("Enter your age: ");

int age = scanner.nextInt();

try {

if (age < 18 || age > 120) {

throw new InvalidAgeException("Invalid age entered. Age must be between 18 and 120.");

} else {

System.out.println("Your age is valid.");

}

} catch (InvalidAgeException e) {

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

} finally {

scanner.close();

}

}

}

**Exercise 4: Throwing an Exception**

**Objective**: Demonstrate how to throw exceptions manually.

**Instructions:**

1. Write a method checkEligibility that takes an age parameter.
2. If the age is less than 18, throw an IllegalArgumentException with the message "Not eligible for the program."
3. In the main method, call this method and catch the exception.

**Solution:**

public class ThrowingException {

public static void main(String[] args) {

try {

checkEligibility(16); // Change this to test different ages

} catch (IllegalArgumentException e) {

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

}

}

public static void checkEligibility(int age) {

if (age < 18) {

throw new IllegalArgumentException("Not eligible for the program.");

}

System.out.println("Student is eligible for the program.");

}

}

**Exercise 5: Nested Try-Catch**

**Objective**: Demonstrate handling multiple exceptions in nested blocks.

**Instructions:**

1. Write a program that asks for two numbers from the user.
2. Try dividing the first number by the second.
3. If an exception occurs, catch it, and inside that block, try reading a valid number.

**Solution:**

import java.util.InputMismatchException;

import java.util.Scanner;

public class NestedTryCatch {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

try {

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

int num1 = scanner.nextInt();

try {

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

int num2 = scanner.nextInt();

int result = num1 / num2;

System.out.println("Result: " + result);

} catch (ArithmeticException e) {

System.out.println("Cannot divide by zero.");

}

} catch (InputMismatchException e) {

System.out.println("Invalid input for the first number. Please enter an integer.");

} finally {

scanner.close();

}

}

}

**Exercise 6: Handling FileNotFoundException**

**Objective**: Handle file-related exceptions.

**Instructions:**

1. Write a program that tries to open and read from a file named "nonexistentfile.txt".
2. Handle the FileNotFoundException and display an appropriate message.

**Solution:**

import java.io.File;

import java.io.FileNotFoundException;

import java.util.Scanner;

public class FileExceptionHandling {

public static void main(String[] args) {

try {

File file = new File("nonexistentfile.txt");

Scanner fileReader = new Scanner(file);

while (fileReader.hasNextLine()) {

System.out.println(fileReader.nextLine());

}

fileReader.close();

} catch (FileNotFoundException e) {

System.out.println("File not found! Please check the file path.");

}

}

}

**Problem Statement:**

Create a Java program that simulates a basic calculator. It should allow the user to enter two numbers and select an operation (addition, subtraction, multiplication, division). The program should handle the following exceptions:

* **ArithmeticException**: Handle division by zero.
* **NumberFormatException**: Handle invalid number inputs.
* **NullPointerException**: Handle cases where user inputs null (or empty input for any operation).
* **General Exception**: A catch-all to ensure the program doesn't crash unexpectedly.

**Instructions:**

1. Create a Calculator class.
2. In the Calculator class, implement methods for addition, subtraction, multiplication, and division.
3. The main method should interact with the user via console input to perform operations.
4. Use try-catch blocks to handle exceptions as described above.

**Solution Template:**

import java.util.Scanner;

public class Calculator {

// Method for Addition

public static double add(double a, double b) {

return a + b;

}

// Method for Subtraction

public static double subtract(double a, double b) {

return a - b;

}

// Method for Multiplication

public static double multiply(double a, double b) {

return a \* b;

}

// Method for Division

public static double divide(double a, double b) {

if (b == 0) {

throw new ArithmeticException("Error: Cannot divide by zero.");

}

return a / b;

}

// Main method to run the program

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

try {

System.out.println("Enter first number: ");

String input1 = scanner.nextLine();

if (input1 == null || input1.isEmpty()) {

throw new NullPointerException("Error: First input is null or empty.");

}

double num1 = Double.parseDouble(input1); // Convert to number

System.out.println("Enter second number: ");

String input2 = scanner.nextLine();

if (input2 == null || input2.isEmpty()) {

throw new NullPointerException("Error: Second input is null or empty.");

}

double num2 = Double.parseDouble(input2); // Convert to number

System.out.println("Choose operation (+, -, \*, /): ");

String operation = scanner.nextLine();

double result = 0;

switch (operation) {

case "+":

result = add(num1, num2);

break;

case "-":

result = subtract(num1, num2);

break;

case "\*":

result = multiply(num1, num2);

break;

case "/":

result = divide(num1, num2);

break;

default:

System.out.println("Invalid operation selected.");

return;

}

System.out.println("The result of " + num1 + " " + operation + " " + num2 + " is: " + result);

} catch (NumberFormatException e) {

System.out.println("Invalid input! Please enter valid numbers.");

} catch (ArithmeticException e) {

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

} catch (NullPointerException e) {

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

} catch (Exception e) {

System.out.println("An unexpected error occurred: " + e.getMessage());

} finally {

System.out.println("Thank you for using the calculator!");

scanner.close(); // Close the scanner

}

}

}

**Explanation:**

1. **Methods for operations**:
   * Basic operations like addition, subtraction, multiplication, and division are handled in separate methods.
   * The division method throws an ArithmeticException if division by zero is attempted.
2. **Input validation**:
   * The program takes inputs as strings and tries to convert them into numbers (Double.parseDouble()).
   * If the user enters a non-numeric value, a NumberFormatException is caught.
   * The program also checks if the user leaves the input empty, throwing a NullPointerException if the input is null or empty.
3. **Try-catch blocks**:
   * The try block is where user input is collected and processed.
   * The catch blocks handle specific exceptions: NumberFormatException, ArithmeticException, and NullPointerException.
   * A general Exception block is used to catch any unexpected errors.
4. **Finally block**:
   * Ensures that the scanner is closed whether or not an exception occurs.

**Example Output:**

**Case 1: Correct Input**

Enter first number:

10

Enter second number:

5

Choose operation (+, -, \*, /):

+

The result of 10.0 + 5.0 is: 15.0

Thank you for using the calculator!

**Case 2: Invalid Number Format**

Enter first number:

abc

Invalid input! Please enter valid numbers.

Thank you for using the calculator!

**Case 3: Division by Zero**

Enter first number:

10

Enter second number:

0

Choose operation (+, -, \*, /):

/

Error: Cannot divide by zero.

Thank you for using the calculator!

**Learning Points:**

* **Exception Handling**: Understanding how to use try-catch blocks and differentiate between different exceptions.
* **User Input Validation**: Handling invalid user inputs in a user-friendly way.
* **Program Flow**: Structuring the program to handle errors gracefully without crashing.

This exercise is great for beginners to practice handling different types of exceptions and dealing with user input validation in a structured way.