**Activity 10 – Exception Handling**

**Program 1**

import java.util.Scanner;

public class ExceptionExample

{

 public static void main(String[] args)

{

 Scanner scanner = new Scanner(System.in);

try

{

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

 Int number = scanner.nextInt();

 // Division by zero will cause an ArithmeticException

 int result = 10 / number;

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

 }

 catch (ArithmeticException e)

{

 System.out.println("Error: Cannot divide by zero. Please enter a non-zero number.");

 }

catch (Exception e)

{

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

 }

finally

{

 // The code in this block will be executed whether an exception occurs or not

 System.out.println("Finally block: This code always runs.");

 // It's a good practice to close resources, like a Scanner, in the finally block

 scanner.close();

 }

 System.out.println("Program execution continues after the try-catch-finally block.");

 }

}

**In this example:**

* The user is asked to input a number, and the program attempts to perform a division operation.
* If the user enters 0, an ArithmeticException will be caught in the catch block specifically for arithmetic exceptions.
* If the user enters any other non-integer input or there is another unexpected error, it will be caught in the more general catch block.
* The finally block is used to ensure that certain code (such as closing a resource like a Scanner) is executed regardless of whether an exception occurs or not.

**Remember that it's crucial to handle exceptions appropriately in your programs to ensure robustness and graceful error recovery.**

**Program 2**

import java.util.Scanner;

public class ArrayExample

{

 public static void main(String[] args)

{

 Scanner scanner = new Scanner(System.in);

 try

{

 System.out.print("Enter the size of the array: ");

 int size = scanner.nextInt();

 // Attempt to create an array with the specified size

 int[] numbers = new int[size];

 System.out.println("Enter elements of the array:");

 for (int i = 0; i < size; i++)

{

 System.out.print("Element " + (i + 1) + ": ");

 numbers[i] = scanner.nextInt();

 }

 // Attempt to access an index beyond the array's bounds

 System.out.print("Enter the index to retrieve from the array: ");

 int index = scanner.nextInt();

 System.out.println("Value at index " + index + ": " + numbers[index]);

 }

catch (NegativeArraySizeException e)

{

 System.out.println("Error: Array size cannot be negative.");

 }

catch (ArrayIndexOutOfBoundsException e)

{

 System.out.println("Error: Index is out of bounds. Please enter a valid index.");

 }

catch (Exception e)

{

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

 }

finally

{

 // Close the Scanner in the finally block to ensure proper resource cleanup

 scanner.close();

 System.out.println("Finally block: This code always runs.");

 }

 System.out.println("Program execution continues after the try-catch-finally block.");

 }

}

**In this example:**

* The program prompts the user to enter the size of the array and the elements of the array.
* It attempts to create an array of the specified size and populate it with user-inputted values.
* The user is then asked to enter an index, and the program attempts to retrieve the value at that index.
* Various exceptions, such as NegativeArraySizeException and ArrayIndexOutOfBoundsException, are caught in the appropriate catch blocks.
* The finally block is used to close the Scanner to ensure proper resource cleanup.

**This example demonstrates how to handle exceptions related to array operations.**

**Program 3**

This program simulates a simple bank account with withdrawal and deposit operations.

import java.util.Scanner;

public class BankTransactionExample

{

 private static double balance = 1000; // Initial balance

 public static void main(String[] args)

{

 Scanner scanner = new Scanner(System.in);

 try

{

 System.out.println("Welcome to the Bank Transaction Program!");

 // Perform transactions until the user chooses to exit

 while (true)

{

 System.out.println("\n1. Deposit");

 System.out.println("2. Withdraw");

 System.out.println("3. Check Balance");

 System.out.println("4. Exit");

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

 int choice = scanner.nextInt();

 switch (choice)

{

 case 1:

 System.out.print("Enter the deposit amount: $");

 double depositAmount = scanner.nextDouble();

 deposit(depositAmount);

 break;

 case 2:

 System.out.print("Enter the withdrawal amount: $");

 double withdrawalAmount = scanner.nextDouble();

 withdraw(withdrawalAmount);

 break;

 case 3:

 checkBalance();

 break;

 case 4:

 System.out.println("Exiting the program. Thank you!");

 return;

 default:

 System.out.println("Invalid. Please enter a valid option.");

 }

 }

 }

catch (Exception e)

{

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

 }

finally

{

 // Close the Scanner in the finally block to ensure proper resource cleanup

 scanner.close();

 System.out.println("Finally block: This code always runs.");

 }

 }

private static void deposit(double amount)

{

 if (amount <= 0)

{

 throw new IllegalArgumentException("Deposit amount must be greater than zero.");

 }

 balance += amount;

 System.out.println("Deposit successful. New balance: $" + balance);

}

private static void withdraw(double amount)

{

 if (amount <= 0)

{

 throw new IllegalArgumentException("Withdrawal amount must be greater than zero.");

 }

 if (amount > balance)

{

 throw new IllegalStateException("Insufficient funds. Cannot withdraw $" + amount);

 }

 balance -= amount;

 System.out.println("Withdrawal successful. New balance: $" + balance);

}

private static void checkBalance()

{

 System.out.println("Current balance: $" + balance);

}

}

**In this example:**

* The program presents a simple menu to the user with options for deposit, withdrawal, checking balance, and exiting the program.
* The deposit and withdraw methods simulate transactions and throw exceptions for invalid inputs or insufficient funds.
* The try-catch block in the main method catches any unexpected exceptions, providing a user-friendly error message.
* The finally block ensures that the Scanner is closed, regardless of whether an exception occurs or not.

**This business example demonstrates how exception handling can be used in a banking context to ensure the program handles user interactions and transactions robustly.**

**Program 4**

Copy your program for Activity 9 and add Try, Catch and Finally code as necessary.

**Program 5**

Create a program that manages a library's catalogue of books. Users can borrow and return books, and the program will handle exceptions related to book availability and invalid inputs.

Hints

* Display menu options – Borrow a book, Return a book, check available books and exit
* Use arrays to store the books and quantities in the library
* Use Methods for “borrow book”, return book” and “check available books” code
* Add try, catch and finally code to catch unexpected exceptions