|  |  |  |  |
| --- | --- | --- | --- |
| **Course Name:** | **Object Oriented Programming** | **Semester:** | **III** |
| **Date of Performance:** | **25 / 09 / 2023** | **Batch No:** | **A – 3** |
| **Faculty Name:** | **Pragya Gupta** | **Roll No:** | **16014022050** |
| **Faculty Sign & Date:** |  | **Grade/Marks:** | **\_\_\_ / 25** |

|  |  |  |
| --- | --- | --- |
| **Writing Program (07)** | **Performance in lab**  **and Viva (05+03)** | **Post lab questions, conclusion and**  **completion (03+02+05)** |
|  |  |  |

**Experiment No: 6**

**Title: Exception Handling in Java**

|  |
| --- |
| **Aim and Objective of the Experiment:** |
| Learn the how to handle exceptions in Java. |

|  |
| --- |
| **COs to be achieved:** |
| **CO3:** Define exceptions and use I/O streams. |

|  |
| --- |
| **Tools used:** |
| 1. Java Development Kit (JDK) 2. Visual Studio Code |

|  |
| --- |
| **Theory:** |
| 1. (About types of exceptions, the methods used for exceptions and keywords used for exception handling)   In Java, exceptions are a way to handle unexpected or exceptional situations in your code. Java provides a robust exception handling mechanism to make your code more robust and reliable. Here are some key concepts related to exceptions in Java:   1. **Types of Exceptions:**   Java exceptions can be broadly categorized into two types:   * **Checked Exceptions:** These are exceptions that the compiler forces you to handle or declare. They typically represent recoverable errors. Examples include IOException, SQLException, and ClassNotFoundException. * **Unchecked Exceptions (Runtime Exceptions):** These exceptions do not need to be explicitly caught or declared. They often represent programming errors or situations that can't be easily recovered from. Examples include NullPointerException, ArrayIndexOutOfBoundsException, and ArithmeticException.  1. **Exception Hierarchy:** All exceptions in Java are subclasses of the java.lang.Throwable class. The Throwable class has two main subclasses:  * java.lang.Exception: This is the base class for all checked exceptions. * java.lang.RuntimeException: This is the base class for all unchecked exceptions.  1. **Handling Exceptions:** Exception handling in Java is done using a combination of try, catch, finally, and throw blocks.  * try: The try block encloses the code that might throw an exception. * catch: The catch block follows a try block and is used to catch and handle exceptions. You can have multiple catch blocks to handle different types of exceptions. * finally: The finally block is used to specify code that will be executed regardless of whether an exception is thrown or not. It's often used for cleanup tasks. * throw: The throw keyword is used to explicitly throw an exception.   Example:  try {  // Code that may throw an exception  } catch (ExceptionType1 e1) {  // Handle ExceptionType1  } catch (ExceptionType2 e2) {  // Handle ExceptionType2  } finally {  // Cleanup code (optional)  }   1. **Exception Keywords:**  * try: Encloses the code that may throw an exception. * catch: Handles exceptions by specifying the type of exception to catch and the code to run when that exception occurs. * finally: Specifies code that is always executed, whether an exception is thrown or not. It's typically used for cleanup operations. * throw: Throws a custom exception or a predefined exception explicitly. * throws: Declares that a method may throw one or more exceptions. It's used in method signatures.   Here's a simple example of exception handling in Java:  public class Example {  public static void main(String[] args) {  try {  int result = divide(10, 0);  System.out.println("Result: " + result);  } catch (ArithmeticException e) {  System.err.println("Division by zero is not allowed.");  } finally {  System.out.println("Finally block always runs.");  }  }  public static int divide(int a, int b) {  if (b == 0) {  throw new ArithmeticException("Division by zero");  }  return a / b;  }  }  In this example, we handle the ArithmeticException thrown by the divide method using a try-catch block and ensure the finally block runs regardless of whether an exception is thrown or not. |

|  |
| --- |
| **Code:** |
| 1. **The program takes two numbers from users in command prompt, Num1 and Num2. The division of Num1 and Num2 is displayed in the Result. If Num1 or Num2 were not an integer, the program would throw a NumberFormatException. If Num2 were Zero, the program would throw an Arithmetic Exception and Display the exception in a result.**     1. **Using try and multiple catch.**   import java.util.\*;  public class exp6\_q1\_a  {      public static void main(String[] args)      {          Scanner sc = new Scanner(System.in);          System.out.println("using try and multiple catch");          try {              System.out.print("\nenter Num1: ");              int num1 = Integer.parseInt(sc.nextLine());              System.out.print("enter Num2: ");              int num2 = Integer.parseInt(sc.nextLine());              int result = divideNumbers(num1, num2);              System.out.println("result: " + result);          } catch (NumberFormatException e) {              System.err.println("NumberFormatException: Please enter valid integers for Num1 and Num2.");          } catch (ArithmeticException e) {              System.err.println("ArithmeticException: Division by zero is not allowed.");          } finally {              sc.close();          }      }      public static int divideNumbers(int num1, int num2) {          if (num2 == 0) {              throw new ArithmeticException("division by zero");          }          return num1 / num2;      }  }   * 1. **Using multiple exceptions in a single catch block.**   import java.util.\*;  public class exp6\_q1\_b  {      public static void main(String[] args)      {          Scanner sc = new Scanner(System.in);          System.out.println("using multiple exceptions in single catch block");          try {              System.out.print("\nenter num1: ");              int num1 = Integer.parseInt(sc.nextLine());              System.out.print("enter num2: ");              int num2 = Integer.parseInt(sc.nextLine());              int result = divideNumbers(num1, num2);              System.out.println("result: " + result);          } catch (Exception e) {              System.err.println("exception: " + e.getMessage());          } finally {              sc.close();          }      }      public static int divideNumbers(int num1, int num2) {          if (num2 == 0) {              throw new ArithmeticException("division by zero");          }          return num1 / num2;      }  }   1. **Create a user defined exception subclass TimeException with necessary constructors and overridden toString method. Write a program which accepts two integers with time in minutes and seconds and find the sum. It throws an object of the TimeException class if the value exceeds 60seconds otherwise it displays the total time. On printing, the exception object should display an exception name, appropriate message for exception.**   import java.util.\*;  class TimeException extends Exception {      public TimeException(String message) {          super(message);      }      @Override      public String toString() {          return "TimeException: " + getMessage();      }  }  public class exp6\_q2 {      public static void main(String[] args) {          Scanner sc = new Scanner(System.in);          try {              System.out.print("\nenter minutes 1: ");              int min1 = sc.nextInt();              System.out.print("enter seconds 1: ");              int sec1 = sc.nextInt();              if (sec1 > 60) {                  throw new TimeException("seconds must be less than or equal to 60.");              }              System.out.print("enter minutes 2: ");              int min2 = sc.nextInt();              System.out.print("enter seconds 2: ");              int sec2 = sc.nextInt();              if (sec2 > 60) {                  throw new TimeException("seconds must be less than or equal to 60.");              }              int totalMinutes = min1 + min2;              int totalSeconds = sec1 + sec2;              if (totalSeconds >= 60) {                  totalMinutes += totalSeconds / 60;                  totalSeconds %= 60;              }              System.out.println("total time: " + totalMinutes + " minutes " + totalSeconds + " seconds");          } catch (TimeException e) {              System.out.println(e);          } catch (Exception e) {              System.out.println("invalid input. enter valid integers.");          } finally {              sc.close();          }      }  } |

|  |
| --- |
| **Output:** |
|  |

|  |
| --- |
| **Post Lab Subjective/Objective type Questions:** |
| 1. **Compare throw and throws.**   **throw**: throw is a keyword in Java used to explicitly throw an exception from a method or block of code. When you throw an exception using throw, you specify an instance of an exception class or a subclass of Throwable. It is used to indicate that an exceptional situation has occurred within your code, and you want to propagate this exception up the call stack to be handled by an appropriate catch block.  Example:  throw new CustomException("This is a custom exception.");  **throws**: throws is a keyword used in method declarations to indicate that the method might throw one or more exceptions. It is used to specify the exceptions that a method can throw, allowing the calling code to handle those exceptions or propagate them further up the call stack. The throws clause is part of the method signature.  Example:  public void someMethod() throws IOException, SQLException {  // ...  }  In summary, throw is used to throw an exception explicitly within a method or block, while throws is used in a method declaration to specify the exceptions that the method might throw.   1. **What are the advantages of using exception handling?**   Exception handling is a programming technique that helps manage and respond to exceptional conditions or errors in a more structured and controlled manner. Here are the advantages of using exception handling:   1. **Error Management**: Exception handling allows you to handle and manage errors gracefully, improving the robustness and reliability of your code. It helps prevent your program from crashing when an unexpected error occurs. 2. **Separation of Concerns**: Exception handling separates the code that detects and reports errors (the try block) from the code that handles errors (the catch blocks). This separation makes your code more readable and maintainable. 3. **Propagation**: Exceptions can be propagated up the call stack, allowing you to handle them at the appropriate level of your program. This means that errors can be handled at the level where they can be best understood and addressed. 4. **Cleaner Code**: Exception handling can lead to cleaner and more readable code because it eliminates the need for extensive error-checking conditions in your code. This makes the main logic of your program more focused and easier to understand. 5. **Debugging**: Exception stack traces provide valuable information about the error, including the call stack, which helps in debugging and diagnosing issues. 6. **Recovery**: In some cases, you can use exception handling to recover from errors gracefully, allowing your program to continue execution in a controlled manner.   Overall, exception handling is a fundamental feature of modern programming languages like Java, designed to improve code reliability and maintainability in the presence of errors.   1. **What is the use of finally block?**   The finally block in Java is used in conjunction with the try-catch block to ensure that a piece of code is executed whether an exception is thrown or not. The main purposes of the finally block is:   * **Cleanup**: The finally block is often used to perform cleanup operations, such as closing files, releasing resources (e.g., closing a database connection), or deallocating memory. This ensures that resources are properly released, even if an exception occurs. * **Guaranteed Execution**: Code in the finally block is guaranteed to execute, regardless of whether an exception is thrown or caught. This ensures that critical cleanup tasks are not skipped. * **Completing Actions**: The finally block is typically used when you want to ensure that certain actions are completed, such as finalizing a transaction or logging important information.   Example:  try {  // Code that may throw an exception  } catch (Exception e) {  // Handle the exception  } finally {  // Cleanup or other important actions  }  In summary, the finally block is used to ensure that essential cleanup or finalization code is executed, making it an important part of robust exception handling in Java.   1. **Suppose the statement2 causes an exception in following try-catch block:**     **Answer the following questions:**   1. **Will statement3 be executed?**   If statement2 causes an exception and that exception is not caught within the try block, then statement3 will not be executed. Control will immediately transfer to the appropriate catch block (if one matches the thrown exception) or further up the call stack.   1. **If the exception is not caught, will statement4 be executed?**   No, if the exception is not caught within the try block, statement4 will not be executed in the current context. Control will either transfer to a suitable catch block or propagate the exception up the call stack to the caller's context. If there is no suitable catch block in the current context or up the call stack, the program may terminate, and the exception details will be displayed.   1. **If the exception is caught in the catch block, will statement4 be executed?**   Yes, if the exception is caught and handled within one of the catch blocks, control will resume execution after the catch blocks. In this case, statement4 will be executed.   1. **If the exception is passed to the caller, will the statement4 be executed?**   If the exception is not caught within the current method and is instead propagated to the caller (i.e., not caught anywhere in the current call stack), then statement4 will not be executed in the current context. The behavior of statement4 depends on whether the caller of the method containing this try-catch block catches or propagates the exception further. If the exception is not caught at any level up the call stack, the program may terminate without executing statement4. |

|  |
| --- |
| **Conclusion:** |
| In conclusion, this experiment helped us acquire valuable skills in handling exceptions in Java and working with I/O streams. |

|  |
| --- |
| **Signature of faculty in-charge with Date:** |