Green University of Bangladesh

Department of Computer Science and Engineering (CSE)

Faculty of Sciences and Engineering

Semester: (Spring, Year:2023), B.Sc. in CSE (Day)

Lab Report NO 02

Course Title	**: Object Oriented Programming Lab**
Course Code	**: CSE 202**
Section	**: D1**

Lab Report on the Topic: Java Exception Handling

<u>Student Details</u>

	Name	**ID**
1.	Jahidul Islam	221002504

Lab Date	**: 23/05/2023**
Submission Date	**: 30/05/2023**
Course Teacher's Name	**: Ayesha Khatun**

<u>Lab Report Status</u>

Marks: ………………………………… **Signature:.....................**

Comments:... **Date:..............................**

Title:

Exception Handling in Java: A Comparative Study

Abstract:

Exception handling is an essential aspect of software development, enabling programmers to gracefully handle and recover from unexpected errors or exceptional conditions that may occur during program execution. This lab report explores Java exception handling, covering concepts such as try-catch blocks, checked and unchecked exceptions, exception propagation, and best practices for effective exception handling.

1. Introduction:

Exception handling plays a vital role in creating robust and fault-tolerant Java applications. By anticipating and handling exceptions appropriately, developers can ensure that their programs handle unexpected situations gracefully and continue executing without crashing or producing incorrect results. This lab report presents an overview of Java exception handling mechanisms and demonstrates their practical usage through various examples.

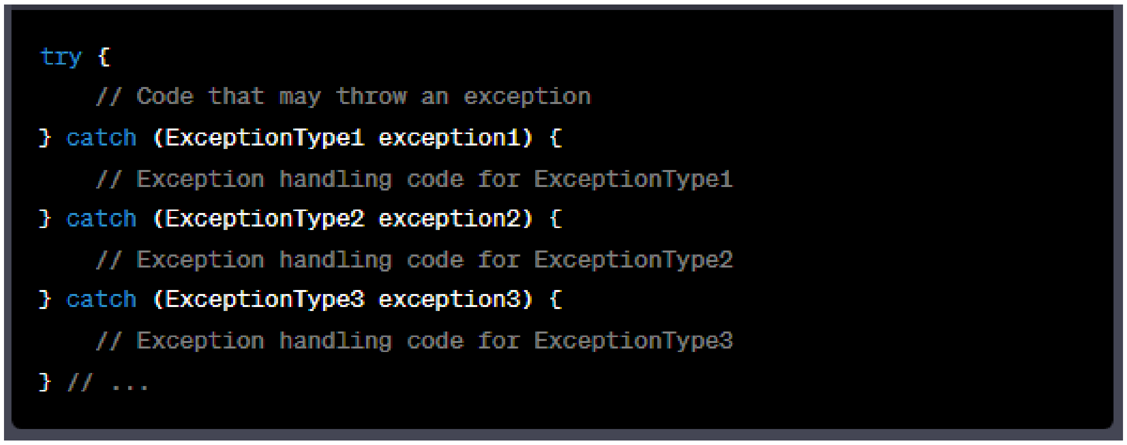
2. Exception Handling in Java:

2.1. Try-Catch Blocks:

Java's try-catch blocks allow developers to catch and handle exceptions thrown during program execution. The "try" block encloses the code that may potentially throw an exception, while the "catch" block catches and handles the exception if it occurs. We can have multiple catch blocks to handle different types of exceptions, allowing for more specific error handling.

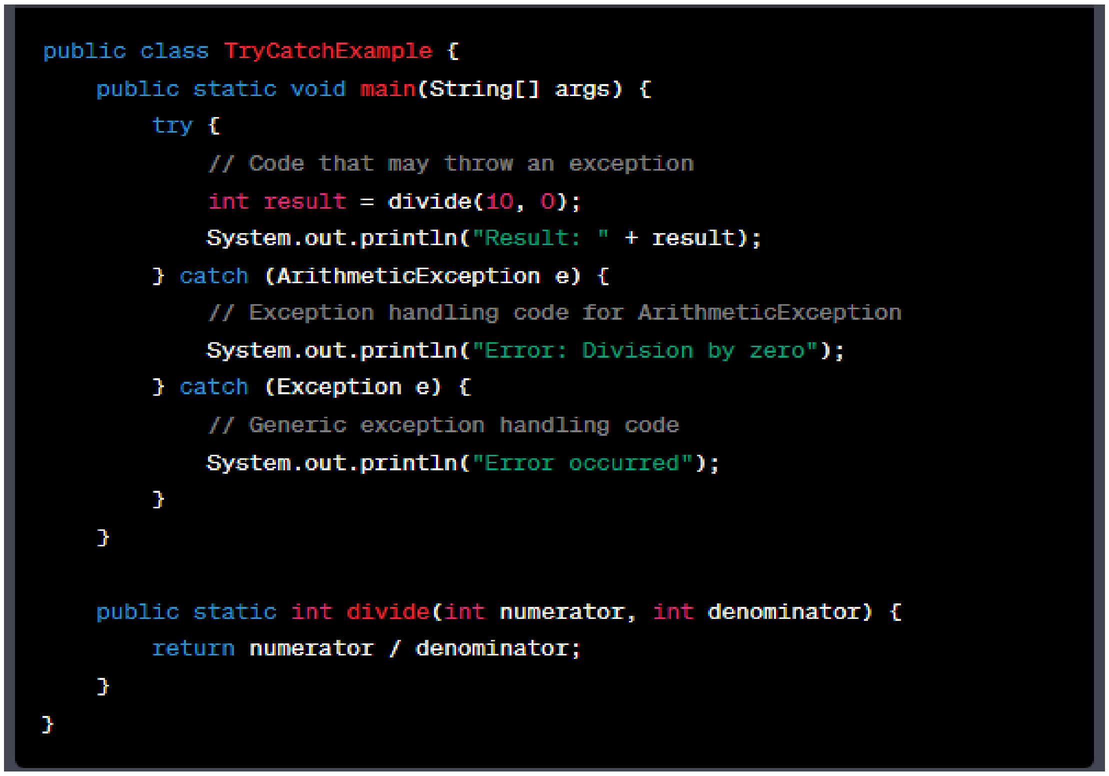
The try-catch block is used for exception handling, allowing you to catch and handle exceptions that may occur during the execution of your program. It consists of a "try" block followed by one or more "catch" blocks.

The general syntax of a try-catch block in Java is as follows:



```java
try {
    // Code that may throw an exception
} catch (ExceptionType1 exception1) {
    // Exception handling code for ExceptionType1
} catch (ExceptionType2 exception2) {
    // Exception handling code for ExceptionType2
} catch (ExceptionType3 exception3) {
    // Exception handling code for ExceptionType3
} // ...
```

Here's an example to illustrate the usage of try-catch block:



```java
public class TryCatchExample {
    public static void main(String[] args) {
        try {
            // Code that may throw an exception
            int result = divide(10, 0);
            System.out.println("Result: " + result);
        } catch (ArithmeticException e) {
            // Exception handling code for ArithmeticException
            System.out.println("Error: Division by zero");
        } catch (Exception e) {
            // Generic exception handling code
            System.out.println("Error occurred");
        }
    }

    public static int divide(int numerator, int denominator) {
        return numerator / denominator;
    }
}
```

2.2. Checked and Unchecked Exceptions:

Java distinguishes between checked and unchecked exceptions. Checked exceptions must be declared in the method signature or caught within the method body, forcing developers to handle or propagate them. Unchecked exceptions, on the other hand, do not require explicit handling, but can still be caught and handled if desired.

3. Exception Propagation:

Exceptions can propagate up the call stack if they are not handled at the point where they occur. The calling method can choose to handle the exception or propagate it further. This mechanism allows for centralized error handling and avoids cluttering every method with exception handling code.

4. Best Practices for Exception Handling:

4.1. Use Specific Exception Types:

When catching exceptions, it is good practice to catch specific exception types rather than using a general catch-all block. This approach enables more targeted error handling and improves code readability.

4.2. Handle Exceptions at the Appropriate Level:

Exceptions should be handled at a level of the code where meaningful recovery or corrective actions can be taken. This ensures that exceptions are appropriately dealt with and prevents the propagation of errors to higher levels of the application.

4.3. Provide Clear and Informative Error Messages:

Exception messages should be informative, concise, and clear, providing developers and users with useful information about the cause and potential remedies for the error. Meaningful error messages aid in troubleshooting and facilitate the resolution of issues.

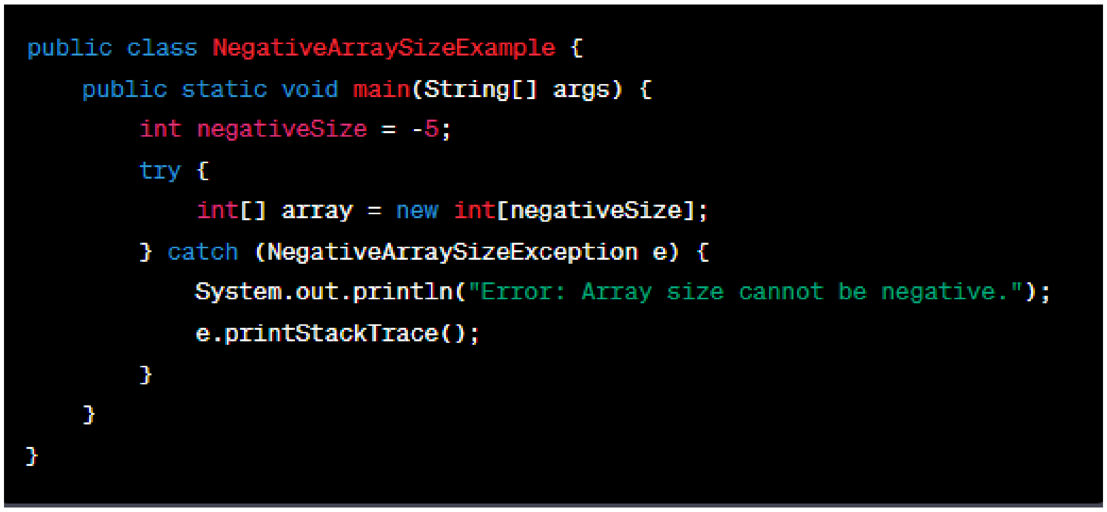
5. Case Study with code example:

In this lab report, we will explore three common exceptions in Java: **NegativeArraySizeException**, **NullPointerException**, and **SecurityException**. We will discuss each exception, its cause, and provide code examples to demonstrate how exception handling can be used to handle these exceptions effectively.

5.1. NegativeArraySizeException:

The **NegativeArraySizeException** is thrown when an array is initialized with a negative size. This exception occurs when the length parameter provided while creating an array is negative.

5.1.1. Code Example:



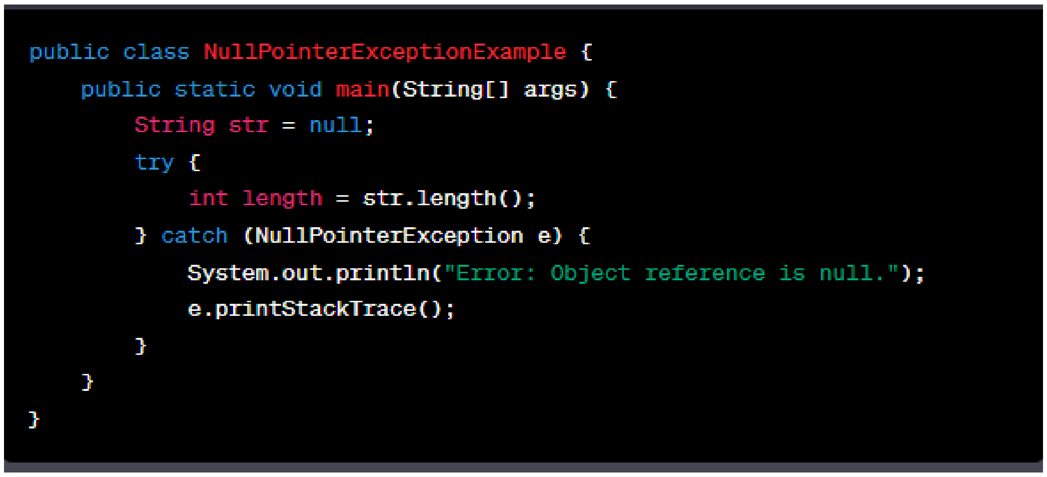
```java
public class NegativeArraySizeExample {
    public static void main(String[] args) {
        int negativeSize = -5;
        try {
            int[] array = new int[negativeSize];
        } catch (NegativeArraySizeException e) {
            System.out.println("Error: Array size cannot be negative.");
            e.printStackTrace();
        }
    }
}
```

We attempt to create an array with a negative size (-5). Inside the try block, the code creates an array using the negative size. When the exception occurs, the catch block is executed. It prints an error message and the stack trace using *e.printStackTrace().*

5.2. NullPointerException:

The **NullPointerException** is one of the most common exceptions in Java. It is thrown when an object reference is null and we attempt to access or invoke methods on that null reference.

5.2.1. Code Example:



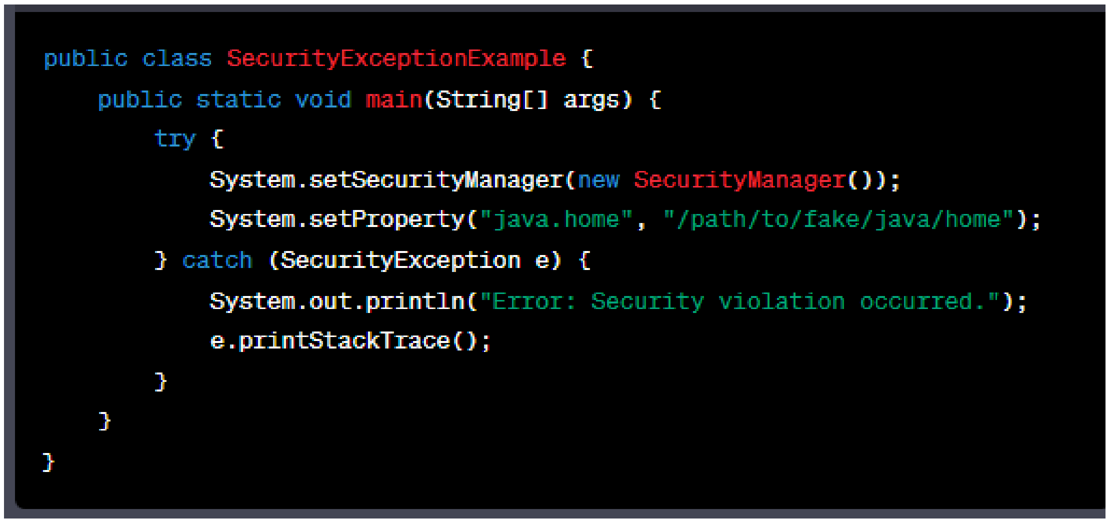
```
public class NullPointerExceptionExample {
    public static void main(String[] args) {
        String str = null;
        try {
            int length = str.length();
        } catch (NullPointerException e) {
            System.out.println("Error: Object reference is null.");
            e.printStackTrace();
        }
    }
}
```

In the above code, we declare a string variable str and assign it a null value. Inside the try block, we invoke the **length()** method on the null reference str. As a result, a **NullPointerException** is thrown. The catch block catches the exception, displays an error message, and prints the stack trace.

5.3. SecurityException:

The **SecurityException** is thrown when a security violation is detected. It typically occurs when a Java application tries to perform an operation that is not permitted by the security policy.

5.3.1. Code Example:



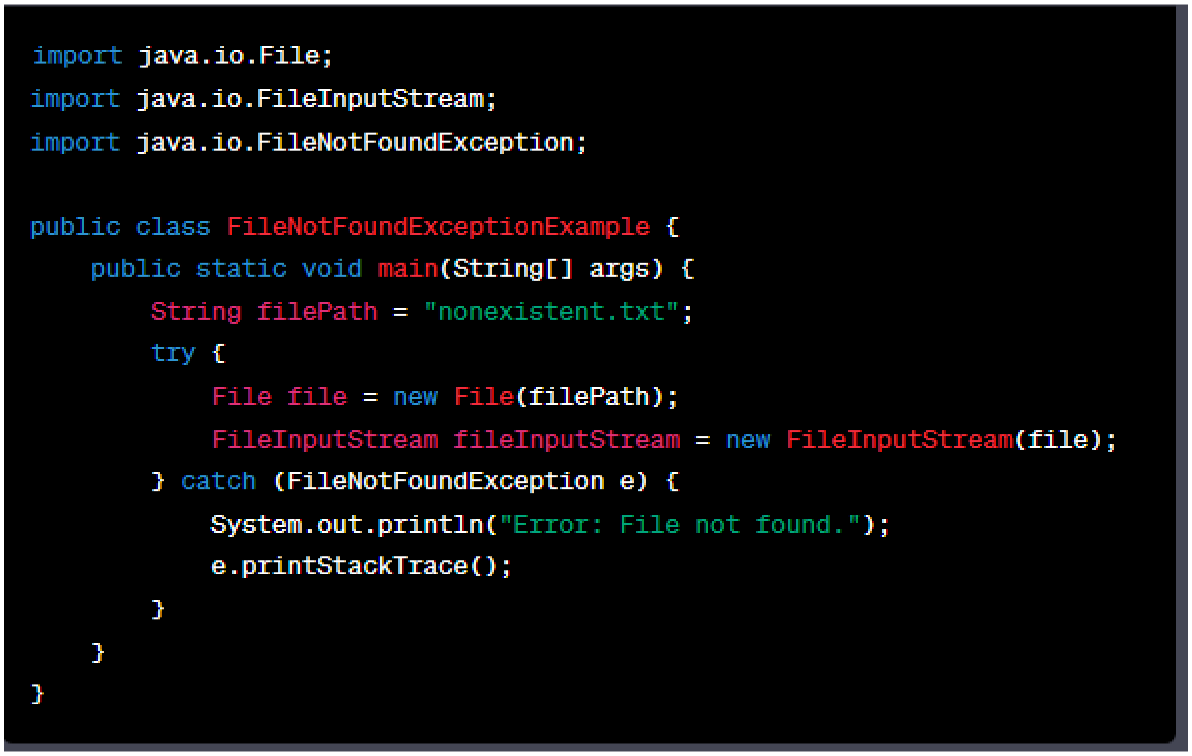
```
public class SecurityExceptionExample {
    public static void main(String[] args) {
        try {
            System.setSecurityManager(new SecurityManager());
            System.setProperty("java.home", "/path/to/fake/java/home");
        } catch (SecurityException e) {
            System.out.println("Error: Security violation occurred.");
            e.printStackTrace();
        }
    }
}
```

We tried to set a custom security manager and modify the system property java.home. This modification is not allowed, and it triggers a **SecurityException**. The catch block catches the exception, displays an error message, and prints the stack trace.

5.4. FileNotFoundException:

The FileNotFoundException is thrown when an attempt is made to access a file that does not exist or cannot be found.

5.4.1. Code Example:



```java
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;

public class FileNotFoundExceptionExample {
    public static void main(String[] args) {
        String filePath = "nonexistent.txt";
        try {
            File file = new File(filePath);
            FileInputStream fileInputStream = new FileInputStream(file);
        } catch (FileNotFoundException e) {
            System.out.println("Error: File not found.");
            e.printStackTrace();
        }
    }
}
```

We have create a **FileInputStream** object for a file that does not exist (nonexistent.txt). As a result, a **FileNotFoundException** is thrown. The catch block catches the exception, displays an error message, and prints the stack trace.

6. Discussion and Conclusion:

Exception handling is a crucial aspect of Java programming as it enables programmers to gracefully handle and manage runtime errors. In this lab, we explored the fundamentals of exception handling and its various components, including try-catch blocks, exception classes, and exception propagation.

One of the primary benefits of exception handling is the ability to handle errors and exceptions in a controlled manner, preventing the abrupt termination of a program. By using try-catch blocks, we can isolate the code that may potentially throw an exception and provide an alternative course of action if an exception occurs. This helps in maintaining program stability and allows for graceful recovery from errors.

During the lab, we learned how to create custom exception classes by extending the predefined exception classes provided by Java. This feature enables us to define our own exception types that are specific to our application's requirements. By doing so, we can categorize and differentiate various types of exceptions based on their causes or contexts, which facilitates more precise exception handling.

Additionally, exception propagation plays a vital role in managing exceptions across different levels of method calls. When an exception is thrown within a method, it can be propagated to the calling method, which can then choose to handle or propagate the exception further. This mechanism allows for flexible error handling strategies at different levels of the program's execution.

In the lab exercises, we encountered different types of exceptions, such as checked and unchecked exceptions. Checked exceptions are verified by the compiler and must be either caught or declared in the method signature, ensuring that the programmer explicitly acknowledges and handles potential exceptions. On the other hand, unchecked exceptions, also known as runtime exceptions, do not require explicit handling, but they can still be caught and handled if necessary.

In conclusion, exception handling is a crucial feature in Java programming that enables programmers to effectively manage runtime errors. By utilizing try-catch blocks, creating custom exception classes, and understanding exception propagation, we can develop robust and reliable applications. Through this lab, we have gained a deeper understanding of exception handling and its significance in building resilient Java programs.