OpenAI

Java Exception Handling

✅ Exception Handling in Java – Complete Guide for Beginners to Experts

🔹 What is Exception Handling?

Exception Handling in Java is a mechanism to handle **runtime errors** so the normal flow of the program can be maintained.

An **exception** is an **unwanted or unexpected event** that occurs during the execution of a program (like dividing by zero, file not found, or null pointer).

🔹 Types of Exceptions in Java

Type	Description	Examples
Checked	Checked at **compile-time**	`IOException`, `SQLException`
Unchecked	Checked at **runtime**	`NullPointerException`, `ArithmeticException`
Error	Serious problems, not handled in code	`OutOfMemoryError`, `StackOverflowError`

🔹 Exception Class Hierarchy

```php
java.lang.Object
  ↳ java.lang.Throwable
       ↳ java.lang.Exception
            ↳ Checked Exceptions
            ↳ Unchecked Exceptions (RuntimeException)
       ↳ java.lang.Error (not typically handled)
```

🔹 Core Keywords for Handling Exceptions

Keyword	Description
`try`	Block of code to monitor for exceptions
`catch`	Block of code to handle the exception
`finally`	Always executed whether exception occurs or not
`throw`	Used to explicitly throw an exception
`throws`	Declares the exceptions a method might throw

🔹 Basic Example

```java
public class Example {
    public static void main(String[] args) {
        try {
            int result = 10 / 0;  // Will throw ArithmeticException
        } catch (ArithmeticException e) {
            System.out.println("Cannot divide by zero!");
        } finally {
            System.out.println("This block is always executed.");
        }
    }
}
```

🔹 Multiple Catch Blocks

```java
try {
    String s = null;
    System.out.println(s.length());  // NullPointerException
} catch (ArithmeticException e) {
    System.out.println("Arithmetic Error");
} catch (NullPointerException e) {
    System.out.println("Null Pointer Error");
} catch (Exception e) {
    System.out.println("General Exception");
}
```

🔹 Using `throw` Keyword

```java
public class Demo {
    static void checkAge(int age) {
        if (age < 18)
            throw new ArithmeticException("Not eligible to vote");
        else
            System.out.println("Eligible to vote");
    }

    public static void main(String[] args) {
        checkAge(16);
    }
}
```

🔹 Using `throws` Keyword

```java
import java.io.*;

public class ThrowsExample {
    static void readFile() throws IOException {
        FileReader fr = new FileReader("file.txt");
    }

    public static void main(String[] args) {
        try {
            readFile();
        } catch (IOException e) {
            System.out.println("File not found");
        }
    }
}
```

🔹 Custom Exception Example

```java
class MyException extends Exception {
    public MyException(String message) {
        super(message);
    }
}

public class Test {
    static void validate(int marks) throws MyException {
        if (marks < 40)
            throw new MyException("Failing marks");
        else
            System.out.println("Passed");
    }

    public static void main(String[] args) {
        try {
            validate(30);
        } catch (MyException e) {
            System.out.println("Custom Exception: " + e.getMessage());
        }
    }
}
```

🔹 Best Practices for Exception Handling

- Catch specific exceptions before general ones.
- Never use exceptions for control flow.
- Always clean up resources (`finally` or try-with-resources).
- Use meaningful messages in custom exceptions.
- Avoid swallowing exceptions without logging.

Would you like a **project or hands-on lab** that practices exception handling (like file reading, login validation, etc.)?

Great! Here's a **hands-on mini-project** for **Exception Handling in Java**—perfect for building your practical skills.

✅ Mini-Project: Student Grade Management with Exception Handling

🎯 Objective:

Create a program to:

- Input student details
- Validate inputs
- Calculate grades
- Handle exceptions like invalid input, division by zero, and custom validation

---

🛠️ Step-by-Step Implementation

🔸 1. Create a Custom Exception

```
java

// File: InvalidMarksException.java
public class InvalidMarksException extends Exception {
    public InvalidMarksException(String message) {
        super(message);
    }
}
```

🔸 2. Create the Main Class

```
java

// File: StudentGradeManager.java
import java.util.Scanner;

public class StudentGradeManager {

    public static String calculateGrade(int marks) throws InvalidMarksException {
        if (marks < 0 || marks > 100) {
            throw new InvalidMarksException("Marks should be between 0 and 100.");
        }

        if (marks >= 90) return "A";
        else if (marks >= 75) return "B";
        else if (marks >= 60) return "C";
        else if (marks >= 40) return "D";
        else return "F";
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        try {
            System.out.print("Enter Student Name: ");
            String name = scanner.nextLine();

            System.out.print("Enter Marks (0-100): ");
            int marks = scanner.nextInt();

            String grade = calculateGrade(marks);
            System.out.println("Student: " + name);
            System.out.println("Marks: " + marks);
            System.out.println("Grade: " + grade);
        } catch (InvalidMarksException e) {
            System.out.println("Error: " + e.getMessage());
        } catch (Exception e) {
            System.out.println("Unexpected Error: " + e.getMessage());
        } finally {
            System.out.println("Grade calculation complete.");
            scanner.close();
        }
    }
}
```

🧪 Test Scenarios

Input	Expected Behavior
Name: John Marks: 85	Prints Grade B
Name: Sara Marks: -5	Triggers **`InvalidMarksException`**
Name: Mike Marks: abc	Triggers general **`Exception`** (input mismatch)
Name: Tina Marks: 101	Triggers **`InvalidMarksException`**

📘 Concepts Practiced

✅ **`try-catch-finally`**
✅ Custom exceptions
✅ Input validation
✅ Runtime exception handling
✅ Resource management (**`scanner.close()`**)

Would you like a **next-level project** that includes **file handling + exception handling** (e.g., student report from a file)?

Awesome! Let's take it to the **next level** with a real-world-style mini-project using **file handling + exception handling** in Java.

✅ Project: Student Report Card Generator from File

🎯 Objective:

Read student data (name and marks) from a file, compute grades, and write results to another file while using exception handling to ensure robustness.

📁 Project Structure

```lua
StudentReportCard/
│
├── Student.java                  --> Model class
├── InvalidMarksException.java   --> Custom exception
├── ReportCardGenerator.java     --> Main class
├── students.txt                 --> Input file
└── report.txt                   --> Output file
```

🔶 1. Create the Model Class

```java
// Student.java
public class Student {
    String name;
    int marks;

    public Student(String name, int marks) {
        this.name = name;
        this.marks = marks;
    }
}
```

🔶 2. Custom Exception

```java
// InvalidMarksException.java
public class InvalidMarksException extends Exception {
    public InvalidMarksException(String message) {
        super(message);
    }
}
```

🔶 3. Main Logic with File Handling and Exception Handling

```java
// ReportCardGenerator.java
import java.io.*;
import java.util.*;

public class ReportCardGenerator {

    public static String getGrade(int marks) throws InvalidMarksException {
        if (marks < 0 || marks > 100) {
            throw new InvalidMarksException("Invalid marks: " + marks);
        }

        if (marks >= 90) return "A";
        else if (marks >= 75) return "B";
        else if (marks >= 60) return "C";
        else if (marks >= 40) return "D";
        else return "F";
    }

    public static void main(String[] args) {
        List<Student> students = new ArrayList<>();

        try (BufferedReader br = new BufferedReader(new FileReader("students.txt"));
             BufferedWriter bw = new BufferedWriter(new FileWriter("report.txt"))) {

            String line;

            while ((line = br.readLine()) != null) {
                String[] parts = line.split(",");
                try {
                    String name = parts[0].trim();
                    int marks = Integer.parseInt(parts[1].trim());
                    Student student = new Student(name, marks);
                    String grade = getGrade(marks);

                    bw.write("Name: " + name + ", Marks: " + marks + ", Grade: " + grade +
"\n");
                } catch (InvalidMarksException ime) {
                    bw.write("Invalid entry: " + line + " -> " + ime.getMessage() + "\n");
                } catch (NumberFormatException | ArrayIndexOutOfBoundsException e) {
                    bw.write("Error parsing line: " + line + " -> " + e.getMessage() + "\n");
                }
            }

            System.out.println("Report generated successfully in report.txt");

        } catch (FileNotFoundException e) {
            System.out.println("Input file not found.");
        } catch (IOException e) {
            System.out.println("Error reading/writing file.");
        }
    }
}
```

📝 4. Input File Example – `students.txt`

```
John, 85
Sara, 102
Mike, -5
Tina, abc
David, 76
```

📄 5. Output File – `report.txt`

```less
Name: John, Marks: 85, Grade: B
Invalid entry: Sara, 102 -> Invalid marks: 102
Invalid entry: Mike, -5 -> Invalid marks: -5
Error parsing line: Tina, abc -> For input string: "abc"
Name: David, Marks: 76, Grade: B
```

🧠 Concepts Practiced

- **`BufferedReader`**, **`BufferedWriter`** (File I/O)
- Try-with-resources (Automatic closing)
- Custom exceptions
- Nested **`try-catch`**
- Data validation
- Writing clean logs/output

Would you like a **version using CSV parsing**, or perhaps a **GUI version with Swing** for advanced practice?