



Southeast University

Department of Computer Science and Engineering (CSE)

School of Sciences and Engineering

Semester: (Summer, Year: 2025)

LAB REPORT NO: 09

Course Title: Introduction to Programming Language II (Java) Lab

Course Code: CSE282.2

Batch: 65

Lab Experiment Name: Exception Handling and reading/writing in a file.

Student Details

	Name	ID
1.	Md. Mahmud Hossain	2023200000799

Submission Date : 03-10-25

Course Teacher's Name : Dr. Mohammed Ashikur Rahman

Lab Report Status

Marks:

Signature:.....

Comments:.....

Date:.....

Lab Task 9: Exception Handling and reading/writing in a file

PROBLEM:

1. Read a file containing names of employees and their salaries from the last 3 workplaces. Calculate the average salary for each employee and write their name and average to a file. Handle `FileNotFoundException`, `IOException`, `NumberFormatException`, and `NegativeNumberException`.
2. Read a file line by line where each line has information about a football match (i.e. France 3 Italy 2). Write down the result of each match to another file (Winner and the number of goals scored by them). If number of goals are equal for both teams, write "Match Drawn". Handle `FileNotFoundException`, `IOException`, `NumberFormatException`, and `NegativeNumberException`.
3. Read a file containing names of cities and their populations. Determine the city with the maximum population and write the information (city name, population). Handle `FileNotFoundException`, `IOException`, `NumberFormatException`, and `NegativeNumberException`.

Solution:

1.

Problem Analysis:

Organizations need to maintain employee salary data across multiple months or workplaces. Manually calculating averages can be error-prone. This program automates the process by reading salary records from a file and generating an output file with each employee's average salary.

Key challenges include:

- Handling invalid salary inputs (non-numeric values).
- Rejecting negative salary values.
- Ensuring robust file handling for reading and writing.

The program addresses these issues using exception handling and custom exceptions.

Background Theory:

1. File I/O in Java:

- BufferedReader and FileReader are used to read employee data from the input file.
- BufferedWriter and FileWriter are used to write the computed averages to the output file.

2. Exception Handling:

- NumberFormatException: Handles invalid salary formats (e.g., text instead of numbers).
- NegativeNumberException: A custom exception defined to handle cases where salary is negative.
- FileNotFoundException: Handles missing input file scenarios.
- IOException: Handles general input/output errors.

3. String Processing:

- `split("\\s+")` is used to tokenize employee records.
- First token = employee name, remaining tokens = salary values

4. Average Calculation:

- $\text{Sum of salaries} \div \text{number of entries}$.

Algorithm Design:

1. Define a custom exception `NegativeNumberException` for negative salary inputs
2. Open input file (`employees.txt`) using `BufferedReader`
3. Open output file (`employee_average.txt`) using `BufferedWriter`.
4. Read the input file line by line.
 - Split the line into tokens: first token = employee name, rest = salaries.
 - Convert salaries to numbers, validating for negative values.
 - If valid, calculate average and write to the output file.
5. Catch and handle exceptions:
 - Invalid salary format (`NumberFormatException`).
 - Negative salary (`NegativeNumberException`).
 - File not found (`FileNotFoundException`).
 - General read/write errors (`IOException`).
6. Close all resources automatically using `try-with-resources`.
7. Print confirmation message after processing.

Code:

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

class NegativeNumberException extends Exception {
    public NegativeNumberException(String message) {
        super(message);
    }
}

public class EmployeeSalary {
    public static void main(String[] args) {
        String inputFile = "employees.txt";
        String outputFile = "employee_average.txt";

        try (BufferedReader br = new BufferedReader(new FileReader(inputFile));
            BufferedWriter bw = new BufferedWriter(new FileWriter(outputFile))) {

            String line;
            while ((line = br.readLine()) != null) {
                try {
                    // Example line: John 3000 3200 3100
                    String[] parts = line.split("\\s+");
                    String name = parts[0];
                    double sum = 0;
                    int count = 0;

                    for (int i = 1; i < parts.length; i++) {
                        double salary = Double.parseDouble(parts[i]);
                        if (salary < 0) throw new NegativeNumberException("Negative salary found for "
+ name);
                        sum += salary;
                        count++;
                    }

                    double avg = sum / count;
                    bw.write(name + " " + avg);
                    bw.newLine();

                } catch (NumberFormatException e) {
                    System.out.println("Invalid number format in line: " + line);
                } catch (NegativeNumberException e) {
                    System.out.println(e.getMessage());
                }
            }
        }
    }
}
```

```

    }

    System.out.println("Average salaries written to " + outputFile);

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

```

Output:

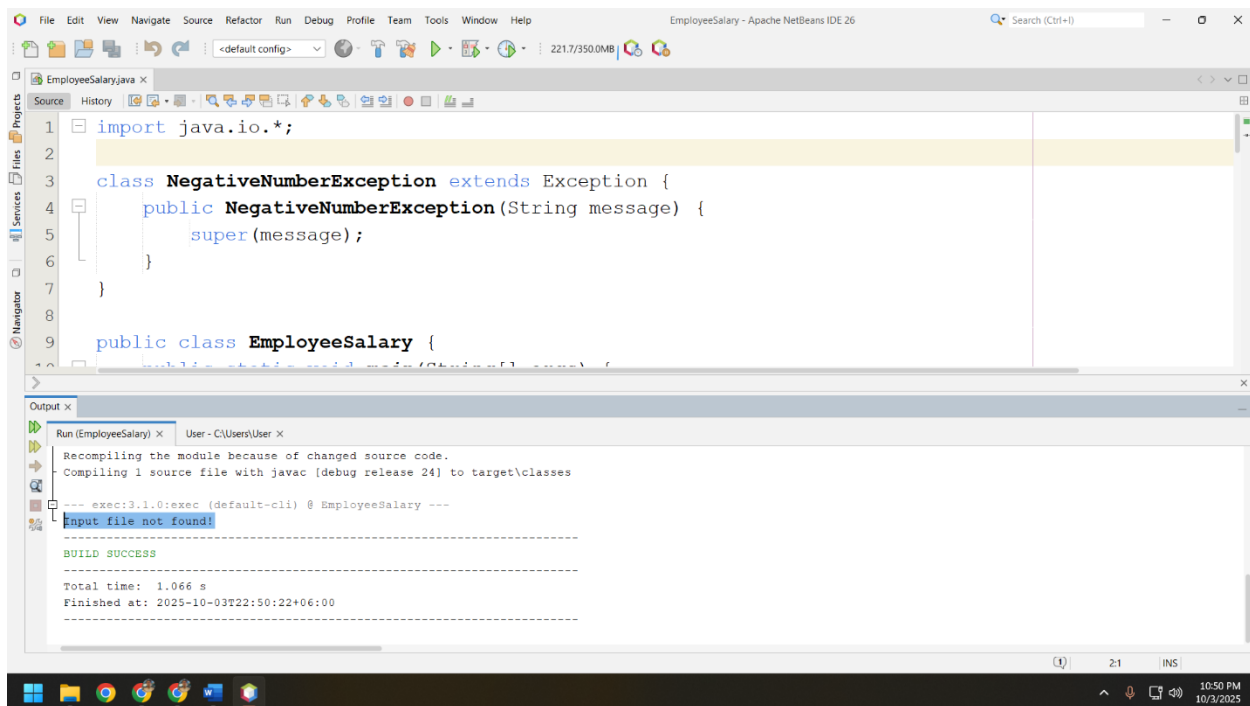


Figure 1: Output

2.

Problem Analysis:

Managing football match data manually can be error-prone, especially when calculating results for multiple matches. This program automates the process by reading match details from a text file, determining winners, and generating an output file with the results.

Challenges addressed in the system include:

- Handling negative goal values.
- Managing invalid number formats.
- Ensuring reliable reading and writing of files.

The system solves these issues using Java exception handling and a custom exception class.

Background Theory:

1. File I/O in Java:

- `BufferedReader` and `FileReader` are used to read input (`matches.txt`).
- `BufferedWriter` and `FileWriter` are used to write results (`results.txt`).

2. Exception Handling:

- `NumberFormatException`: Handles cases where non-numeric values are entered as scores.
- `NegativeNumberException2`: A custom exception created to reject negative goal values.
- `FileNotFoundException`: Handles missing input file errors.
- `IOException`: Handles general file read/write errors.

3. String Processing:

- Input lines are split using `split("\\s+").`
- First team name, its score, second team name, and its score are extracted.

4. Logic:

- If `score1 > score2` → team1 wins.
- If `score2 > score1` → team2 wins.
- If equal → match drawn.

Algorithm Design:

1. Define a custom exception `NegativeNumberException2`.
2. Open input file (`matches.txt`) with `BufferedReader`.
3. Open output file (`results.txt`) with `BufferedWriter`.

4. Read file line by line.
 - Extract team1, score1, team2, score2.
 - Validate scores (no negatives allowed).
 - Determine match result (winner/draw).
5. Write result into results.txt.
6. Catch exceptions for invalid formats, negative values, and file issues.
7. Close all resources automatically using try-with-resources.

Code:

```
import java.io.*;

class NegativeNumberException2 extends Exception {
    public NegativeNumberException2(String message) {
        super(message);
    }
}

public class FootballMatches {
    public static void main(String[] args) {
        String inputFile = "matches.txt";
        String outputFile = "results.txt";

        try (BufferedReader br = new BufferedReader(new FileReader(inputFile));
            BufferedWriter bw = new BufferedWriter(new FileWriter(outputFile))) {

            String line;
            while ((line = br.readLine()) != null) {
                try {
                    // Example line: France 3 Italy 2
                    String[] parts = line.split("\\s+");
                    String team1 = parts[0];
                    int score1 = Integer.parseInt(parts[1]);
                    String team2 = parts[2];
                    int score2 = Integer.parseInt(parts[3]);

                    if (score1 < 0 || score2 < 0) throw new NegativeNumberException2("Negative goals
in: " + line);

                    if (score1 > score2) {
```

```

        bw.write("Winner: " + team1 + " with " + score1 + " goals");
    } else if (score2 > score1) {
        bw.write("Winner: " + team2 + " with " + score2 + " goals");
    } else {
        bw.write("Match Drawn");
    }
    bw.newLine();

    } catch (NumberFormatException e) {
        System.out.println("Invalid number format in: " + line);
    } catch (NumberFormatException2 e) {
        System.out.println(e.getMessage());
    }
}

System.out.println("Match results written to " + outputFile);

} catch (FileNotFoundException e) {
    System.out.println("Input file not found!");
} catch (IOException e) {
    System.out.println("File read/write error!");
}
}
}

```


Output:

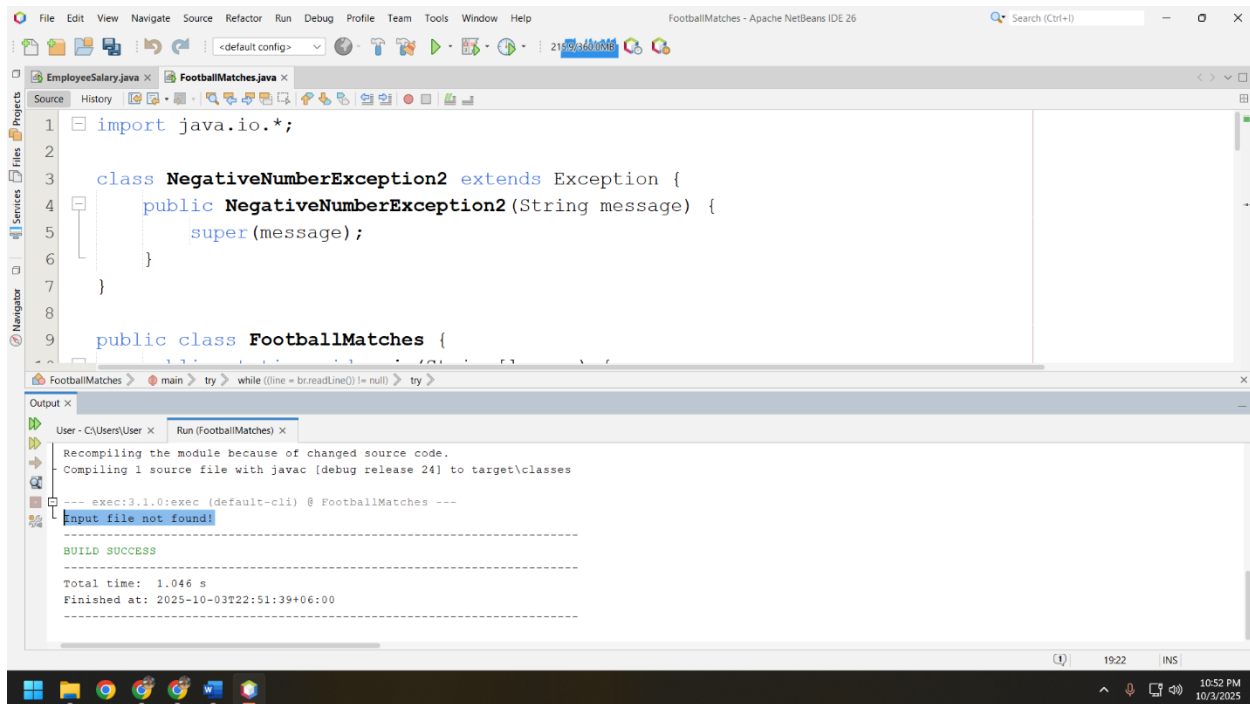


Figure 2: Output

3.

Problem Analysis:

Managing city population data manually can be error-prone, especially when identifying the most populated city from large datasets. This program automates the process by reading city data from an input file, validating it, and writing the city with the maximum population into an output file.

Challenges addressed include:

- Handling negative population values.
- Managing invalid number formats.
- Ensuring efficient file reading and writing.

By using custom exceptions and try-with-resources, the system ensures robustness and error-free execution.

Background Theory:

1. File I/O in Java:

- `BufferedReader` and `FileReader` read input from `cities.txt`.
- `BufferedWriter` and `FileWriter` write output to `max_population.txt`.

2. Exception Handling:

- `NumberFormatException`: Handles cases where population data is not a valid integer.
- `NegativeNumberException3`: Custom exception to reject negative population values.
- `FileNotFoundException`: Handles missing input file errors.
- `IOException`: Handles general file read/write errors.

3. String Processing:

- Input lines are split using `split("\\s+").`
- The first token is taken as the city name, and the second as the population.

4. Logic:

- Maintain a running maximum of population.
- Update `maxCity` whenever a higher population is found.
- After reading all lines, write the result into the output file.

Algorithm Design:

1. Define a custom exception `NegativeNumberException3`.
2. Initialize `maxCity` and `maxPop`.
3. Open input file (`cities.txt`) with `BufferedReader`.
4. Open output file (`max_population.txt`) with `BufferedWriter`.
5. Read each line:
 - Extract city name and population.
 - Validate population (no negatives allowed).
 - If `population > maxPop`, update `maxCity` and `maxPop`.
6. After all lines, write the city with the maximum population to the output file.
7. Handle exceptions for invalid formats, negative values, and file issues.
8. Close resources automatically using `try-with-resources`.

Code:

```
import java.io.*;

class NegativeNumberException3 extends Exception {
    public NegativeNumberException3(String message) {
        super(message);
    }
}

public class CityPopulation {
    public static void main(String[] args) {
        String inputFile = "cities.txt";
        String outputFile = "max_population.txt";

        String maxCity = "";
        int maxPop = -1;

        try (BufferedReader br = new BufferedReader(new FileReader(inputFile));
            BufferedWriter bw = new BufferedWriter(new FileWriter(outputFile))) {

            String line;
            while ((line = br.readLine()) != null) {
                try {
                    // Example line: Dhaka 21000000
```

```

        String[] parts = line.split("\\s+");
        String city = parts[0];
        int population = Integer.parseInt(parts[1]);

        if (population < 0) throw new NegativeNumberException3("Negative population for
" + city);

        if (population > maxPop) {
            maxPop = population;
            maxCity = city;
        }
    } catch (NumberFormatException e) {
        System.out.println("Invalid number format in line: " + line);
    } catch (NegativeNumberException3 e) {
        System.out.println(e.getMessage());
    }
}

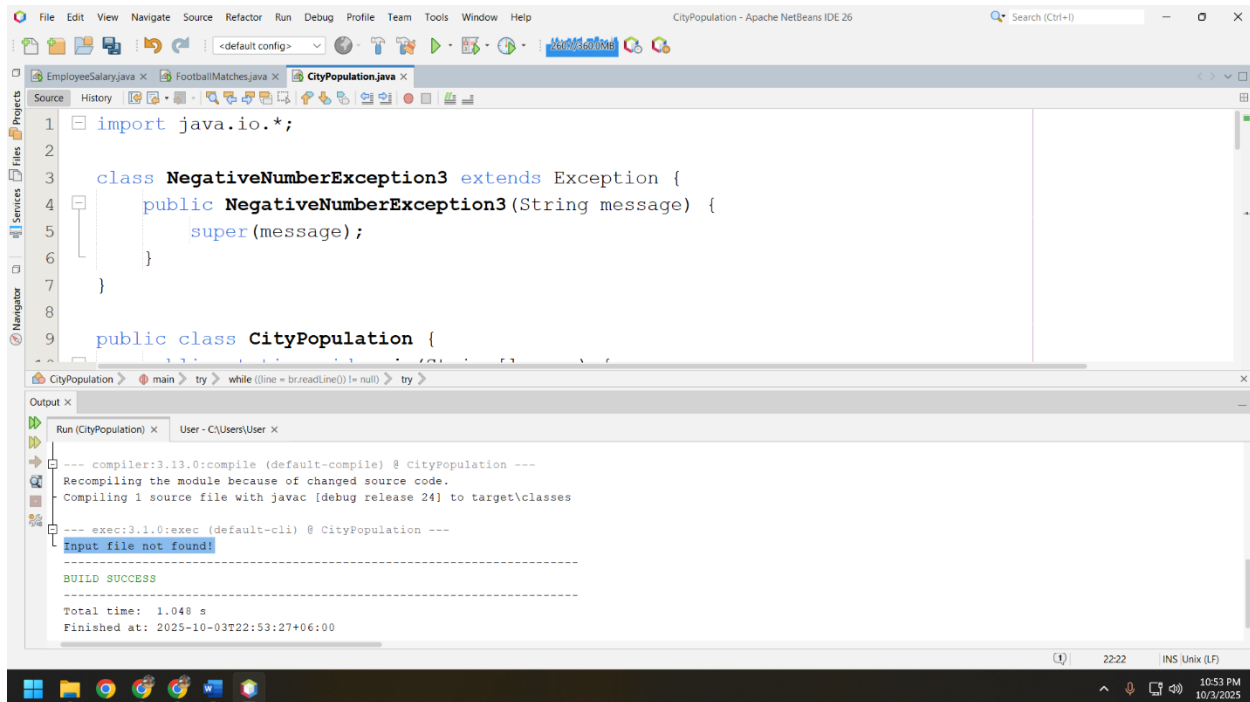
if (maxPop >= 0) {
    bw.write("City with maximum population: " + maxCity + " (" + maxPop + ")");
}

System.out.println("Maximum population written to " + outputFile);

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

```

Output:



The screenshot shows the NetBeans IDE interface. The main editor window displays the following Java code:

```
1 import java.io.*;
2
3 class NegativeNumberException3 extends Exception {
4     public NegativeNumberException3(String message) {
5         super(message);
6     }
7 }
8
9 public class CityPopulation {
```

The output window at the bottom shows the following text:

```
Run (CityPopulation) x User - C:\Users\User x
--- compiler:3.13.0:compile (default-compile) @ CityPopulation ---
Recompiling the module because of changed source code.
Compiling 1 source file with javac [debug release 24] to target\classes
--- exec:3.1.0:exec (default-cli) @ CityPopulation ---
input file not found!
-----
BUILD SUCCESS
-----
Total time: 1.048 s
Finished at: 2025-10-03T22:53:27+06:00
```

The status bar at the bottom indicates the current time is 10:53 PM on 10/3/2025.

Figure 3: Output