MODULE: Java Programming

ASSIGNMENT 3 – File I/O, Buffering, Serialization & Exception Handling

Learning Objectives

By completing this assignment, learners will:

- Understand Java's character and byte stream classes
- Perform file copy operations using multiple stream types
- Measure performance benefits of buffered I/O
- Learn object serialization and deserialization
- Practice building custom exceptions
- Handle common runtime errors using try–catch–finally
- Apply file-based data parsing and collections together

General Instructions

1. Each question must be implemented in separate `.java` files.
2. Use **try-with-resources** wherever possible (recommended).
3. Display clear error messages for missing files or invalid inputs.
4. Validate input and handle exceptions gracefully.
5. Keep code modular by separating logic into methods.
6. Use appropriate stream types (character vs. byte).
7. All file paths must be configurable, not hardcoded.

Estimated Time & Difficulty

Question	Time	Difficulty
Q1	30–45 min	Beginner → Intermediate
Q2	20–30 min	Intermediate
Q3	45–60 min	Intermediate
Q4	60–90 min	Intermediate → Advanced
Q5	30–45 min	Intermediate
Q6	30–45 min	Intermediate
Q7	20–30 min	Beginner

Evaluation Rubric

Criteria	Weight
Correctness & Functionality	50%
Exception Handling Quality	20%
Code Structure & Modularity	15%
Output Clarity	10%
Performance Consideration	5%

Q1. File Copy Using Character Streams & Byte Streams

Requirements

1. Choose a local file (e.g., an image or any binary file).
2. Write a program to copy this file **character-by-character** using:
    - `FileReader` → `FileWriter`
3. Write a second program using **byte streams**:
    - `FileInputStream` → `FileOutputStream`
4. Ensure that the copied file exists and matches the original size.

Notes

- Use `File.length()` to check file sizes.
- Handle `FileNotFoundException` & `IOException`.

Q2. Buffered Streams Performance Comparison

Requirements

1. Update Q1 by adding:
    - `BufferedReader` / `BufferedWriter`
    - `BufferedInputStream` / `BufferedOutputStream`
2. Measure copy time for:
    - Non-buffered version
    - Buffered version
3. Print time taken for both approaches and compare.

Hint

Use `System.currentTimeMillis()` to measure performance.

Expected Output Example

```
Without Buffering: 125 ms
With Buffering: 23 ms
Performance improved by: 102 ms
```

Q3. Object Serialization & Deserialization

Given Class Structure

```
Employee
  int id
  String name
  Address address
  double salary (non-serializable)
```

Requirements

1. Mark Employee as `Serializable`.
2. Make `salary` non-serializable using `transient`.
3. Implement `display()` method.
4. Serialize an Employee object to a file.
5. Deserialize the object and show:
    - non-transient fields restored
    - salary becomes default (0.0)

Notes

- Ensure `Address` class is also serializable.
- Use `ObjectOutputStream` & `ObjectInputStream`.

Q4. Book File Parsing and LinkedList Operations

Input File Format (`books.txt`)

```
121:A234:java:raj:456
102:S234:c++:ekta:567
```

Each line contains:

```
id:isbn:title:author:price
```

Requirements

Part A — Read File & Populate Data

- Read each line
- Split using `:`
- Create Book objects
- Store them in a `LinkedList<Book>`

Part B — Methods to Implement

1. `searchBook(int id)`
    - Return the matching book or message if not found
2. `sellBook(String isbn, int noOfCopies)`
    - Reduce price or quantity (as defined by your system rule)

    - If insufficient copies → throw `NotSufficientBookException`

3. `purchaseBook(String isbn, int noOfCopies)`
    - Increase stock

Part C — Create `BookApp`

Perform:

- Load data from file
- Search for books
- Sell and purchase books
- Display updated info

Q5. User Registration With Custom Exception

Requirements

Part A — Create Exception

Create custom class:

```
InvalidCountryException
```

Overload constructors and pass meaningful messages.

Part B — registerUser Method

```
registerUser(String username, String userCountry)
```

Logic:

- If `userCountry` is not `"India"` → throw `InvalidCountryException`
- Else print:

```
User registration done successfully
```

Part C — Test the Method

Call registerUser from main and observe behavior both with valid and invalid inputs.

Q6. Validate Name & Age From Command-Line Arguments

Requirements

1. Accept **name** and **age** from command-line (`args[]`).
2. Validate age:
    - age ≥ 18
    - age < 60
3. If invalid, throw a custom exception (e.g., `InvalidAgeException`).
4. If valid → print details.

Expected Behavior

```
Input: Rajeev 25
Output: Name: Rajeev, Age: 25
```

If invalid:

```
InvalidAgeException: Age must be between 18 and 59
```

Q7. Division Program With Try–Catch–Finally

Requirements

1. Prompt user to enter two integers.
2. Compute `a / b`.
3. Handle `ArithmeticException` for division by zero.
4. Use `finally` block to print:

```
Inside finally block
```

Sample Outputs

Case 1

```
Enter the 2 numbers: 5 2
The quotient of 5/2 = 2
Inside finally block
```

Case 2

```
Enter the 2 numbers: 5 0
DivideByZeroException caught
Inside finally block
```

BONUS CHALLENGES (Optional)

Bonus 1

Extend the BookApp to store data back to the file after purchase/sell.

Bonus 2

Measure read/write performance using NIO `Files.copy()` and compare with streams.

Bonus 3

Add validation: prevent negative prices or negative book quantities.

Bonus 4

Add menu-driven CLI interface for BookApp.

Reflection Questions

1. Why are buffered I/O streams significantly faster than non-buffered streams?
2. What are risks of serializing classes without version control (`serialVersionUID`)?
3. Why should custom exceptions carry meaningful messages?
4. How does finally guarantee cleanup in exception handling?

5. What challenges did you face parsing file-based data structures?