MODULE: Java Programming

ASSIGNMENT 4 – Java Collections Framework

Learning Objectives

By completing this assignment, learners will:

- Work with core Java collection types (List, Map, PriorityQueue)
- Parse text files and perform word-frequency analysis
- Understand custom sorting using Comparator
- Practice object comparison and equality logic
- Apply collection algorithms to find maximum values
- Implement real-world data processing tasks using Collections

General Instructions

1. Write each question in a separate `.java` file unless question states otherwise.
2. Use **java.util collections only** (no external libraries).
3. Validate input and handle file-related exceptions gracefully.
4. Use meaningful class names, attributes, method names, and comments.
5. Keep code modular by breaking logic into helper methods.
6. Do **not** use hard-coded file paths; read from configurable paths.
7. Print clean, readable output (no messy formatting).

Estimated Time & Difficulty

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

Evaluation Rubric

Criteria	Weight
Correctness & Output	40%

Criteria	Weight
Collection Usage	25%
Code Quality	15%
Input/File Handling	10%
Sorting/Algorithms	10%

Q1. Word Frequency Counter from File (Map Usage)

Requirements

1. Create a text file named **story.txt** containing sample text.
2. Write a program to read all words from this file.
3. Use a `Map<String, Integer>` (preferably `HashMap`) to store:

```
word → frequency
```

4. Print each word with the number of times it appears.

Example Input (story.txt)

```
life is full of fun life is full of fun life
```

Expected Output

```
life appears 3 times
is appears 2 times
full appears 2 times
of appears 2 times
fun appears 2 times
```

Notes

- Normalize words (lowercase).
- Ignore punctuation.
- Use `String.split("\\s+")` to tokenize.

Common Mistakes to Avoid

- Not trimming input lines
- Case-sensitive counting
- Not handling empty lines

Q2. BookCollection Class & Custom Sorting

Real-World Context

Personal or business libraries often store book lists with sorting/search functionality.

Part A: Define Book Class

Attributes:

- title
- author

- price (optional)

Override:

- `toString()` to display book details cleanly
- `equals()` & `hashCode()` so that two books are equal if:
    - same title
    - same author

Part B: Define BookCollection Class

Attributes:

1. `String ownerName`
2. `Book[] books` (array of books)

Required Methods

1. boolean hasBook(Book b)

Return `true` if the array already contains the same book (using equals).

2. void sort()

Sort by:

1. Book title (lexicographically)
2. If titles match → sort by author

Use one of:

- `Comparator`
- `Arrays.sort()` with custom comparator

3. toString()

Print all books in a clean, formatted layout.

Part C: Test BookCollection

1. Create your own BookCollection.
2. Check if the book **"Java in Depth"** exists.
3. Sort the collection.
4. Print the final sorted collection.

Expected Output Example

```
Owner: Rajeev Gupta
Books:
Java Basics – Raj
Java in Depth – Mehta
Spring Boot – Sharma
```

Q3. Read Doubles from File and Find Maximum Value

Requirements

1. Create a file named **data.txt** containing double values, one per line.
2. Read all double values using a `Scanner` or `BufferedReader`.
3. Store values in a `List<Double>`.
4. Find and print the **largest double**.

Example Input

```
97.59
23.70
72.97
18.98
```

Expected Output

```
Largest value: 97.59
```

Notes

- Handle empty file scenario.
- Handle invalid numeric values using `try-catch`.

Q4. PriorityQueue with Custom Comparator for Products

Real-World Context

Companies prioritize items for dispatch or restocking based on pricing or urgency.
PriorityQueue is often used to determine items with highest or lowest priority.

Product Class

Attributes:

- productId (int)
- productName (String)
- productPrice (double)

Override:

- `toString()` to print product details.

Task Requirements

1. Create a **PriorityQueue<Product>**.
2. Provide a **custom Comparator** that sorts products based on `productPrice`:
    - Highest price first **OR** lowest first (choose and document your rule).
3. Insert at least **5 Product objects**.
4. Poll elements from the queue to show the order of priority.

Expected Example Output (If lowest price first)

```
Processing product: Pen (₹10.0)
Processing product: Notebook (₹25.0)
Processing product: Bag (₹400.0)
Processing product: Laptop (₹60000.0)
```

Notes

- Test both ascending and descending comparators.
- PriorityQueue does NOT guarantee sorted iteration — only sorted removal.

BONUS CHALLENGES (Optional)

Bonus 1 — Use LinkedHashMap

Maintain insertion order while performing word frequency.

Bonus 2 — Sort Word Frequency by Count

Use `List<Map.Entry<>>` + Comparator to sort by frequency descending.

Bonus 3 — Convert BookCollection to ArrayList Version

Enhance flexibility and simplify resizing.

Bonus 4 — Top 3 Most Expensive Products

Use a max-heap (PriorityQueue descending order).

Reflection Questions

1. Why is HashMap the preferred structure for word-frequency counting?
2. How does defining `equals()` & `hashCode()` improve object comparison?
3. What real-world systems rely heavily on sorting and prioritization?
4. Why does PriorityQueue not behave like a fully sorted list?
5. What challenges did you face while parsing and storing file data?