420-301-VA: Programming Patterns

Lab 5: The Java Collections Framework - Sets and Maps

Computer Science & Technology
Vanier College

Introduction

In this lab, we continue our exploration of the Java Collections Framework by focusing on two powerful interfaces: `Set` and `Map`. While `List` collections are ordered and allow duplicates, a `Set` is an unordered collection that stores only unique elements. A `Map`, on the other hand, is not a traditional collection but a unique structure for storing key-value pairs, much like a dictionary. Understanding these structures is crucial for tasks like removing duplicates, fast lookups, and associating data, which are common challenges in software development.

Objectives

This lab provides a hands-on introduction to Sets and Maps in Java. By the end of this lab, you should be comfortable with:

- Understanding the fundamental differences between a `Set` and a `List`, particularly regarding uniqueness and performance.
- Using the primary `Set` implementations: `HashSet`, `LinkedHashSet`, and `TreeSet`.
- Understanding the concept of key-value pairs and the purpose of the `Map` interface.
- Using the primary `Map` implementations: `HashMap`, `LinkedHashMap`, and `TreeMap` with custom objects.
- Applying these data structures to solve practical problems like finding unique elements and counting frequencies.

These skills are essential for writing efficient, clean, and effective Java code, especially when dealing with data management and retrieval.

Instructions

- Please try out the examples first to deepen your understanding.
- For each exercise, create the required classes and a main method to test your implementation.
- Copy-paste your formatted code for each exercise into a separate document, followed by screenshots of the console outputs to validate that your code worked as expected.
- You can work with your peers, but each person must write, execute, and document their own work.

1 Part 1: The `Set` Interface - For Unique Elements

1.1 Concept Review: `HashSet`, `LinkedHashSet`, and `TreeSet`

A `Set` is a collection that cannot contain duplicate elements. It models the mathematical set abstraction. Unlike a `List`, it provides no guarantee on the iteration order of its elements and does not support index-based access.

- `HashSet`: Stores elements in a hash table. It is the best-performing implementation but makes no guarantees concerning the order of iteration.
- `LinkedHashSet`: Maintains the insertion order of elements.
- `TreeSet`: Stores elements in a sorted order. Elements must be comparable.

```java
import java.util.HashSet;
import java.util.Set;

public class SetExample {
    public static void main(String[] args) {
        Set<String> cities = new HashSet<>();
        cities.add("London");
        cities.add("Paris");
        cities.add("New York");
        cities.add("Beijing");
        cities.add("Paris"); // This duplicate is ignored

        System.out.println("Set of cities: " + cities);

        // Iterate with a for-each loop
        System.out.print("\nCities in uppercase: ");
        for (String city : cities) {
            System.out.print(city.toUpperCase() + " ");
        }

        // Iterate with the forEach method and a lambda expression
        System.out.print("\nCities in lowercase: ");
        cities.forEach(city -> System.out.print(city.toLowerCase() +
            " "));
        System.out.println();
    }
}
```

Listing 1: Using a HashSet and iterating over elements

Exercise 1.1: Understanding Sets vs. Lists

In this exercise, you will use a `Set` to filter out duplicate entries from a list of names, highlighting a key difference between Sets and Lists.

1. Create an `ArrayList` of strings and add the following names: "Alice", "Bob", "Charlie", "Alice", "David", "Bob".

2. Print the original list to the console to show that it contains duplicates.

3. Create a `LinkedHashSet` from the `ArrayList`. The constructor `new LinkedHashSet<>( yourList)` will automatically remove duplicates while preserving the order of first appearance.

4. Print the set to the console. Observe that the duplicates are gone.

5. Use a 'for-each' loop to iterate through the set and print each unique name in all uppercase letters.

Exercise 1.2: Comparing Set Implementations with Custom Objects

Here, you will observe the different ordering behaviors of `HashSet`, `LinkedHashSet`, and `TreeSet` using a custom `Book` class.

1. Create a `Book` class with two private attributes: `title` (`String`) and `publishYear` (`int`). Implement a constructor, getters, and override `toString()` to return a formatted string.

2. In your `main` method, create an instance of `HashSet<Book>`, `LinkedHashSet<Book>`, and `TreeSet<Book>`.

3. Create several `Book` objects and add the exact same objects to all three sets in a non-alphabetical order.

4. To use `TreeSet`, you must define a natural ordering. Modify the `Book` class to implement `Comparable<Book>` and override the `compareTo` method to sort books alphabetically by title.

5. Print all three sets to the console. Add labels to your output.

6. **Observe and Compare:** Note the random order of `HashSet`, the insertion order of `LinkedHashSet`, and the alphabetical (natural) order of `TreeSet`. Write a short comment in your code explaining the differences you see.

2 Part 2: Performance Comparison - `Set` vs. `List`

While Sets and Lists can both store elements, their internal structures are very different. This leads to significant performance differences for common operations like searching for an element (`contains`) or removing an element (`remove`). Sets, especially `HashSet`, are optimized for fast lookups, whereas Lists can be very slow if you have to search through many elements.

```java
import java.util.*;

public class SetListPerformanceTest {
    static final int N = 50000;

    public static void main(String[] args) {
        // Create a list of N integers
        List<Integer> list = new ArrayList<>();
        for (int i = 0; i < N; i++) {list.add(i);}
        Collections.shuffle(list); // Shuffle for realistic testing

        // Test HashSet performance
        Collection<Integer> hashSet = new HashSet<>(list);
        System.out.println("Member test time for HashSet is " +
           getTestTime(hashSet) + " ms");
        System.out.println("Remove element time for HashSet is " +
           getRemoveTime(hashSet) + " ms");

        // Test ArrayList performance
        Collection<Integer> arrayList = new ArrayList<>(list);
        System.out.println("Member test time for ArrayList is " +
           getTestTime(arrayList) + " ms");
        System.out.println("Remove element time for ArrayList is " +
            getRemoveTime(arrayList) + " ms");
    }

    public static long getTestTime(Collection<Integer> c) {
        long startTime = System.currentTimeMillis();
        // Test if a number is in the collection N times
        for (int i = 0; i < N; i++) { c.contains((int)(Math.random()
            * 2 * N));
        }
        return System.currentTimeMillis() - startTime;
    }

    public static long getRemoveTime(Collection<Integer> c) {
        long startTime = System.currentTimeMillis();
        for (int i = 0; i < N; i++) {
            c.remove(i);
        }
        return System.currentTimeMillis() - startTime;
    }
}
```

Listing 2: SetListPerformanceTest.java (Adapted from Liang Chapter 21)

Exercise 2: Analyze the Performance Results

1. Create a new Java class and copy the code from the `SetListPerformanceTest.java` listing above.

2. Run the code. You may need to run it a few times to see a stable average, as timings can vary.

3. Record the output in your submission document.

4. **Answer the following questions** in comments at the bottom of your code file:

   (a) Which data structure was faster for the membership test (the `contains` method)? By how much?

   (b) Based on what you know about how a `HashSet` works (using hash codes), why do you think it is so much faster for searching?

   (c) If Sets are so much faster for searching, why would a programmer ever choose to use a `List`? (Hint: Think about what a `List` can do that a `Set` cannot).

3 Part 3: The `Map` Interface - For Key-Value Pairs

3.1 Concept Review: `HashMap` and `TreeMap`

A `Map` is an object that maps keys to values. A map cannot contain duplicate keys; each key can map to at most one value. It is useful when you need to retrieve, update, or delete elements based on a unique identifier (the key).

- `HashMap`: Stores key-value pairs in a hash table. It provides the best performance for insertions and lookups but does not maintain any order.
- `LinkedHashMap`: Maintains insertion order or access order of keys.
- `TreeMap`: Stores key-value pairs sorted according to the natural ordering of its keys, or by a `Comparator`.

```java
import java.util.HashMap;
import java.util.Map;

public class MapExample {
    public static void main(String[] args) {
        // Create a Map with student names (String) as keys and
           grades (Integer) as values
        Map<String, Integer> studentGrades = new HashMap<>();

        // Add key-value pairs (entries) to the map
        studentGrades.put("Smith", 85);
        studentGrades.put("Lewis", 92);
        studentGrades.put("Cook", 78);

        System.out.println("Map of student grades: " + studentGrades
           );

        // Retrieve a value using its key
        int lewisGrade = studentGrades.get("Lewis");
        System.out.println("Lewis's grade: " + lewisGrade);

        // Update a value
        studentGrades.put("Smith", 88);
        System.out.println("Updated map: " + studentGrades);

        // Iterate over the map entries
        for (Map.Entry<String, Integer> entry : studentGrades.
           entrySet()) {
            System.out.println("Student: " + entry.getKey() + ",
               Grade: " + entry.getValue());
        }
    }
}
```

Listing 3: Using a HashMap for student grades

Exercise 3.1: Managing a Product Catalog with Custom Objects

You will create a product catalog where each product has a unique ID (key) and is represented by a `Product` object (value).

1. Create a `Product` class with `name` (`String`) and `price` (`double`). Include a constructor, getters, and a `toString()` method.

2. In your `main` method, create a `HashMap<Integer, Product>`. The key is the product ID.

3. **Add Products:** Create at least four `Product` objects. Use the `put()` method to add them to your map with unique integer IDs.

4. **Retrieve a Product:** Use the `get()` method to retrieve the `Product` object for a specific ID and print its details using your `toString()` method.

5. **List All Products:** Iterate through the map's `entrySet()` and print each product's ID and its corresponding `Product` details.

Exercise 3.2: Comparing Map Implementations

Now, you will see how different `Map` implementations handle ordering.

1. Using the same `Product` data from the previous exercise, create three maps: a `HashMap`, a `LinkedHashMap`, and a `TreeMap`.

2. Add the same products with their IDs to each of the three maps. Make sure you add them in a non-sequential ID order (e.g., ID 205, then 101, then 310).

3. Print the contents of all three maps, with clear labels for each.

4. **Analyze and Conclude:** In your code comments, describe the order of the entries in each map. Explain why `TreeMap` was able to sort the entries by key.

4 Part 4: Practical Application - Aggregating and Grouping Data

4.1 Concept Review: Counting Frequencies with a Map

A frequent and powerful programming task is to analyze data to find the frequency of each item. A `Map` is the perfect tool for this: the item is the key, and its frequency (count) is the value. This pattern is not limited to words; it can be used to count numbers, characters, or any other type of object.

The following example, adapted from the textbook, demonstrates how to count word occurrences. It uses a `TreeMap` so that the final output is automatically sorted alphabetically by word.

```
import java.util.Map;
import java.util.TreeMap;

public class WordCountExample {
    public static void main(String[] args) {
        String text = "Good morning. Have a good class. " +
                      "Have a good visit. Have fun!";

        // Use a TreeMap to hold words as keys and their counts as
           values
        Map<String, Integer> wordCountMap = new TreeMap<>();

        // Pre-process the text: convert to lowercase and split into
            words
        // This regex splits by spaces or any punctuation
        String[] words = text.toLowerCase().split("[\\s\\p{Punct}]+"
           );

        // Iterate through the array of words to count them
        for (String word : words) {
            if (word.isEmpty()) continue; // Skip any empty strings
               from split

            if (!wordCountMap.containsKey(word)) {
                // If the word is not in the map, add it with a
                   count of 1
                wordCountMap.put(word, 1);
            } else {
                // If the word is already in the map, get its
                   current count,
                // increment it, and update the map
                int count = wordCountMap.get(word);
                wordCountMap.put(word, count + 1);
            }
        }

        System.out.println("Word Frequencies (alphabetically sorted)
            :");
        // Use entrySet() to iterate and display the final counts
        for (Map.Entry<String, Integer> entry : wordCountMap.
            entrySet()) {
             System.out.println(entry.getKey() + ":\t" + entry.
                getValue());
        }
    }
}
```

Listing 4: CountOccurrenceOfWords.java (Example)

Exercise 4: Summarizing Monthly Expenses

In this exercise, you will use a `Map` to solve another common data-processing problem: aggregating data into categories. You will take a list of individual financial transactions and calculate the total spending for each category (e.g., "Groceries", "Utilities", "Transport").

1. Create a `Transaction` class with two private attributes: `category` (`String`) and `amount` (`double`). Implement a constructor and getters for both attributes.

2. In a `main` method, create an `ArrayList<Transaction>` to represent a list of expenses. Add several `Transaction` objects, ensuring some of them share the same category. For example:

   - new Transaction("Groceries", 55.75)
   - new Transaction("Transport", 22.50)
   - new Transaction("Utilities", 120.00)
   - new Transaction("Groceries", 34.25)

3. Create a `TreeMap<String, Double>`. The key will be the expense category (`String`), and the value will be the total amount spent in that category (`Double`). Using a `TreeMap` will ensure your final report is sorted alphabetically by category.

4. **Aggregate the Expenses:** Iterate through your list of `Transaction` objects. For each transaction:

   - Get the transaction's category and amount.
   - Check if the category already exists as a key in your map.
   - **If the category is NOT in the map:** This is the first expense for this category. `put` the category and its amount into the map.
   - **If the category IS already in the map:** `get` the current total for that category from the map. Add the current transaction's amount to this total. Then, `put` the category back into the map with the new, updated total.

5. **Display the Expense Report:** After the loop has processed all transactions, iterate through your map's `entrySet()`. For each entry, print the category (the key) followed by the total amount spent (the value), formatted neatly. This will serve as a final summary of expenses.

References

Java Collections Framework

- Oracle Java Docs: java.util: The official documentation for the core utility classes, including all collections. This is the primary source of truth for all methods and behaviors.
- Java Documentation for the Set Interface: Details the contract that all `Set` implementations must follow, such as not allowing duplicates.
- Java Documentation for the Map Interface: Explains the key-value structure and the core methods like `put`, `get`, and `containsKey`.
- Java Documentation for HashSet: Describes the fast but unordered hash table implementation of the `Set` interface.
- Java Documentation for TreeMap: Explains the sorted, tree-based implementation of the `Map` interface, useful for when you need keys in a predictable order.
- Baeldung: A Guide to the Java Collections Framework: An excellent, practical third-party guide with clear code examples and explanations for the entire Collections Framework.