МИНИСТЕРСТВО ЦИФРОВОГО РАЗВИТИЯ СВЯЗИ И МАССОВЫХ КОММУНИКАЦИЙ РОССИЙСКОЙ ФЕДЕРАЦИИ

Ордена Трудового Красного Знамени федеральное государственное бюджетное образовательное учреждение высшего образования

«МОСКОВСКИЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ СВЯЗИ И ИНФОРМАТИКИ»

(МТУСИ)

Кафедра «Математическая Кибернетика и Информационные технологии»

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ №6

по дисциплине

«Информационные технологии и программирование»

на тему

“Работа с коллекциями”

Выполнил:

студент группы БВТ2302

Миронов А. А.

Москва, 2024 г.

Задание 1:

Написать программу, которая считывает текстовый файл и выводит на экран топ-10 самых часто встречающихся слов в этом файле. Для решения задачи использовать коллекцию Map, где ключом будет слово, а значением - количество его повторений в файле.

```
import java.io.FileNotFoundException;
import java.util.Scanner;
import java.util.AbstractMap;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.Map;
import java.util.List;

public class MostOften10 {
    Run | Debug
    public static void main(String[] args) {
    String filePath = "ray451.txt";
    File file = new File(filePath);
    Scanner scanner = null;
    String buffer = "";
    try {
    scanner = new Scanner(file);
    } catch (FileNotFoundException e) {
        System.out.println(x:"Can't see the file in the current path");
    e.printStackTrace();
    }


    List<Map.Entry<String, Integer>> mlist1 = new ArrayList<>();

    while (scanner.hasNextLine()) {
        buffer += scanner.nextLine();
    }
```

Для начала передадим в экземпляр класса File путь к файлу с текстом, в котором и будем считать самые часто встречающиеся слова. Затем передадим

этот экземпляр в scanner. Поместим этот кусок кода в блок try, чтобы при выбрасывании исключения о том, что файл не найден, оно обрабатывалось. Создаём ссылку на объект вида списка из пар Map.entry слово-количество в тексте, помещаем внутрь этой ссылки Arraylist. Дальше просто с помощью цикла while проверяем есть ли в объекте класса сканнер следующая строка нашего файла, и если да, то добавляем её в специальный буфер, просто являющийся строкой. Когда процесс завершён, закрываем сканнер и переводим строку нашего буфера в нижний регистр, чтобы заглавные и маленькие буквы считались за одни. Создаём массив строк, куда помещаем наш буффер, разделённый с помощью метода сплит на отдельным словам: регекс \W+ означает один или более символ, не являющийся буквой, цифрой или нижним подчёркиванием.

```
while (scanner.hasNextLine()) {
    buffer += scanner.nextLine();
}
scanner.close();

buffer = buffer.toLowerCase();
String[] bufferarr = buffer.split(regex:"\\W+");


for (int i=0; i<bufferarr.length;i++){
    int count = 0;
    for (int j=0; j<bufferarr.length;j++){
        if (bufferarr[i].equals(bufferarr[j])){
            count++;
        }
    }
    boolean exists = false;
    for (Map.Entry<String, Integer> entry : mlist1) {
        if (entry.getKey().equals(bufferarr[i])) {
        exists = true;
        break;
        }
    }
        if (!exists) {
            mlist1.add(new AbstractMap.SimpleEntry<>(bufferarr[i], count));
```

Я решил сделать основной алгоритм подсчёта частоты использования слов через двойной цикл, где я при каждом совпадении очередного слова с любым словом из нашего массива строк увеличиваю локальный счётчик на единицу. Чтобы избежать дубликатов, я каждый раз проверяю циклом, есть ли уже такое слово в нашем списке слов, сравнивая ключ-слово каждой мап энтри нашего эрэйлиста с текущим словом, и если оно совпало хоть с одним словом из нашего эрэйлиста, то не добавляем это слово-количество в эрэйлист. Иначе же мы добавляем его. Так как мы не можем добавить Map.Entry (Entry существует внутри реализации Map и нужен для представления пар ключ-значение) напрямую объявив его, ведь он является интерфейсом, а не классом и не может быть напрямую инстанцирован. Но мы можем добавить вместо этого в эррэйлист объект класса SimpleEntry, являющийся вложенным статическим классом AbstractMap с ключом-словом и значением-его частотой в тексте. SimpleEntry также является одной из реализаций интерфейса Map.Entry, поэтому мы можем складывать его объекты в список для map.entry.

```
        }
            if (!exists) {
                mlist1.add(new AbstractMap.SimpleEntry<>(bufferarr[i], count));

            }
    }


    Collections.sort(mlist1, new Comparator<Map.Entry<String, Integer>>() {
    @Override
    public int compare(Map.Entry<String, Integer> o1, Map.Entry<String,
    Integer> o2) {
        return o2.getValue().compareTo(o1.getValue());
    }
    });

    for (int i=0; i<10;i++){
        System.out.println(mlist1.get(i));
    }

    }


}
```

Используем метод sort из класса collections, который на вход принимает список который надо сортировать, а вторым параметром — анонимный класс,

реализующий интерфейс Comparator, необходимый для переопределения метода int compare, реализующего алгоритм сортировки для sort. Аргументами нашего переопределённого метода будут являться первый и второй элементы Map.Entry<String,Integer>, дальше мы возвращаем сравнение значений-integer с помощью метода compareto, который возвращает 1, если первое, сравниваемое число больше второго, -1, если меньше, и 0, если они равны. Вот, что это значит в контексте сортировки: если объект, для которого вызывается compareto больше, чем объект переданный ему в качестве аргумента, то это значит, что вызывающий объект должен быть размещён после объекта, переданного в качестве аргумента, в отсортированном списке. Если же объект, для которого вызывается compareto меньше, чем объект переданный ему в качестве аргумента, то это значит, что вызывающий объект должен быть размещён перед объектом, переданным в качестве аргумента, в отсортированном списке. Если переменные в compareto указаны в том же порядке, что и в методе compare в параметрах, то сортировка будет по убыванию, если же в обратном, то по возрастанию.

Задание 2:
Написать обобщенный класс Stack<T>, который реализует стек на основе массива. Класс должен иметь методы push для добавления элемента в стек, pop для удаления элемента из стека и peek для получения верхнего элемента стека без его удаления.

Создаём класс stack с видом данных дженерик, т е можем назначать любой тип данных, которыми будет наполненяться стэк при инициализации объекта класса. В конструкторе класса приводим массив объектов длины, указываемой при создании объекта к массиву дженериков, потому что компилятор не работает с дженериками в конструкторе напрямую. Задаём размер 0. При занесении элемента в массив (стэк), мы использую постфиксный инкремент:

сначала указываем позицию, на которую встаёт элемент в массиве (начиная с 0), а потом прибавляет к счётчику элементов в стэке 1.

```
public class Stack<T> {
    private T[] data;
    private int size;
    private int capacity;

    public Stack(int capacity) {
        this.capacity = capacity;
        data = (T[]) new Object[capacity];
        size = 0;
    }

    public void push(T element) {
        if (size == capacity) {
            System.out.println(x:"Stack is full");
        }else{
            data[size++] = element;
        }

    }

    public T peek() {
        if (size == 0) {
            System.out.println(x:"Stack is empty");
            return data[size];
        } else{
        return data[size - 1];
        }
    }
```

```
    public T pop() {
        if (size == 0) {
            System.out.println(x:"Stack is empty");
            return data[size];
        } else{
        T element = data[--size];
        data[size] = null;
        return element;
        }
    }

}
```

При удалении элемента, мы сначала с помощью префиксного декремента минусуем счётчик стэка на 1, и после записываем его в переменную, которая вернётся как результат функции, а также приравниваем к null элемент по номеру счётчика минусованного на 1. Делаем мы это лишь после уменьшения счётчика на 1, потому что когда счётчик на нуле, это значит, что объектов нет, но на нулевом индексе массива в то же время может быть объект. По этой причине индекс последнего элемента массива по идее должен быть на один меньше размера массива (у нас это не в каждом моменте так, а есть некие ухищрения). Когда мы берём самый близкий к концу элемент стэка, мы также берём элемент из массива по индексу на один меньше размера стэка.

Теперь напишем класс методом мэйн, чтобы была возможность протестировать программу.

```
public class mainStack {
    Run | Debug
    public static void main(String[] args) {
        Stack<Integer> stack = new Stack<>(capacity:10);
stack.push(element:1);
stack.push(element:2);
stack.push(element:3);
System.out.println(stack.pop());
System.out.println(stack.peek());
stack.push(element:4);
System.out.println(stack.pop());
    }
}
```

Задание 3:

Необходимо разработать программу для учета продаж в магазине.
Программа должна позволять добавлять проданные товары в коллекцию, выводить список проданных товаров, а также считать общую сумму продаж и наиболее популярный товар.

Варианты выполнения задания:

Вариант 1

Использовать ArrayList для хранения списка проданных товаров.

Для начала создадим класс продукта с наименованием позиции, ценой, соответствующими геттерами и методом перевода в строку

```
public class product {
    private String name;
    private double price;

    public product(String name, double price) {
        this.name = name;
        this.price = price;
    }

    public String getName() {
        return name;
    }

    public double getPrice() {
        return price;
    }

    @Override
    public String toString() {
        return name + " (Price of product: " + price + ")";
    }

}
```

Создаём эррэйлист для списка всех проданных товаров и хэшмап для вычисления самого популярного товара. Прописываем метод sellProduct, который добавляет в эррэйлист проданный товар, а также кладёт одну штуку его в хэшмап.

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

public class sales {
    private List<product> soldProducts;
    private Map<String, Integer> productSalesCount;

    public sales() {
        soldProducts = new ArrayList<>();
        productSalesCount = new HashMap<>();
    }


    public void sellProduct(product product) {
        soldProducts.add(product);
        productSalesCount.put(product.getName(), productSalesCount.getOrDefault(product.getName(),
        defaultValue:0) + 1);
    }


    public void printSoldProducts() {
        System.out.println(x:"List of sold products:");
        for (product product : soldProducts) {
            System.out.println(product.toString());
        }
    }
```

Если товар уже есть в хэшмап, мы перезаписываем его имя, оставляем то же количество, с помощью метода getordefaulf, возвращающего значение продукта из хэшмапы по ключу, если такой ключ есть, а если нет, то просто возвращает 0 (любое указанное число), затем просто к записываемому значению в пут прибавляем 1, это имитирует ещё один проданный товар этого вида.

Дальше всё проще: выводим из эррэйлиста список всех проданных товаров и тут нам пригодится конвертация класса продукт в строку.
В методе, в котором мы получаем общую сумму трат, тоже нет ничего удивительного.
В методе, где мы ищем самый продаваемый продукт, мы превратили хэшмапу в список entrySet() состоящий из пар ключ-значений , чтобы из каждой пары Map.Entry можно было получить ключ и значение с помощью getkey и getvalue и вписать их в переменные, если количество проданного товара больше, чем

максимальное предыдущего вида. Мы используем Map.Entry в данном случае, потому что она представляет собой один элемент коллекции Map, в то время как Map и является этой коллекцией пар ключ-значение.

```
    }

    public double totalSalesAmount() {
        double total = 0;
        for (product product : soldProducts) {
            total += product.getPrice();
        }
        return total;
    }

    public String mostPopularProduct() {
        String mostPopular = null;
        int maxCount = 0;

        for (Map.Entry<String, Integer> entry : productSalesCount.entrySet()) {
            if (entry.getValue() > maxCount) {
                maxCount = entry.getValue();
                mostPopular = entry.getKey();
            }
        }
        return mostPopular;
    }

}
```

А теперь напишем класс методом мэйн для нашего класса с высчитыванием различной информации о проданных товаров и проверим его работоспособность.

```java
public class mainSales {
    Run | Debug
    public static void main(String[] args) {
        product hat = new product(name:"hat", price:10);
        product cat = new product(name:"cat", price:20);
        product watch = new product(name:"watch", price:30);
        product jacket = new product(name:"jacket", price:35);
        product pants = new product(name:"pants", price:25);
        sales newsales = new sales();
        newsales.sellProduct(hat);
        newsales.sellProduct(hat);
        newsales.sellProduct(cat);
        newsales.sellProduct(cat);
        newsales.sellProduct(cat);
        newsales.sellProduct(watch);
        newsales.sellProduct(jacket);
        newsales.sellProduct(pants);
        newsales.printSoldProducts();
        System.out.println(newsales.totalSalesAmount());
        System.out.println(newsales.mostPopularProduct());

    }
}
```

Вывод:

Мы научились работать с основным видами коллекций в джава, поняли, что такое переменные дженерик и поняли как использовать их на практике, открыли для себя работу с компаратором и сортировкой. Уяснили основные различия видов коллекций, и то, какие коллекции лучше подходят для решения задач из определённых сфер.