Instituto Superior de Engenharia do Porto

Estruturas de Informação

Projeto 1

Feito por:

André Ferreira, 1190378

Bernardo Barbosa, 1220741

Diogo Cunha, 1221071

Tomás Peixoto,1221948

Professores:

Alberto Sampaio (ACS)

22/9/23

DEPARTAMENTO DE ENGENHARIA INFORMÁTICA
Instituto Superior de Engenharia do Porto

Índice

Glossário

1. **Array**: Estrutura de dados que armazena elementos do mesmo tipo acessíveis por índices.
2. **Atributos:** Variáveis que definem características de objetos em classes.
3. **Carga (Stalls):** Pontos de carregamento para carros elétricos, semelhantes a postos de abastecimento de combustível para carros a gasolina.
4. **Cidades (Cities):** Áreas urbanas ou locais específicos onde os pontos de carregamento para veículos elétricos estão localizados.
5. **CSV (Comma-Separated Values):** Formato de arquivo que armazena dados em forma de tabela, com informações separadas por vírgulas, semelhante a uma planilha do Excel.
6. **Diagnóstico de Classes (Diagrama de Classes):** Representação gráfica que ilustra o comportamento a relação, necessidade e funcionalidade doas classes.
7. **Distância Haversine:** Fórmula matemática que calcula a distância entre dois pontos na Terra usando coordenadas de latitude e longitude, semelhante a medir a distância em um globo terrestre.
8. **Mapa (Map):** Estrutura de dados que associa chaves a valores.
9. **Objeto (classe em Java):**
10. **Parâmetro:** Valor de entrada que personaliza o comportamento de uma função ou método em programação, muitas vezes valor pedido em um método ou função.
11. **Pares de chave-valor:** Associação de os primeiros parâmetros em um mapa (chaves) a sua associação: valores.
12. **Powertrain:** Componentes de um veículo elétrico para propulsão.
13. **Quota:** Proporção de uma população em relação ao todo.
14. **Set:** Coleção de elementos únicos.
15. **SortedMap:** Tipo específico de Mapa organizado em ordem alfabética.

Introdução

Neste relatório pretendemos apresentar a logística contida na resolução de cada um dos exercícios propostos no primeiro projeto dado na unidade curricular **Estruturas de Informação**.

O principal objetivo durante o projeto foi a aplicação dos conceitos e técnicas aprendidos na unidade curricular para a resolução de exercícios estes apresentados ao grupo.

O grupo pretendeu obter a máxima compreensão e simplicidade na ideia que transmitiu no relatório, desta forma para ilustrar as ideias apresentadas e explicar o comportamento dos conceitos (classes) foi gerado um diagrama de classes (figura 1).

De facto, o relatório em questão realça os respetivos algoritmos de cada exercício (exercício 1 a exercício 8) pela ordem mencionada. Cada exercício contém o seu título, figura (imagem) e uma breve mas concisa explicação de como o algoritmo resolve a questão proposta.

Por fim o relatório apresenta as fontes, onde foi adquirido conhecimento posterior para a resolução e melhor compreensão do trabalho.

Diagrama de Classes

Nesta secção, iremos apresentar o diagrama de classes elaborado para o projeto.

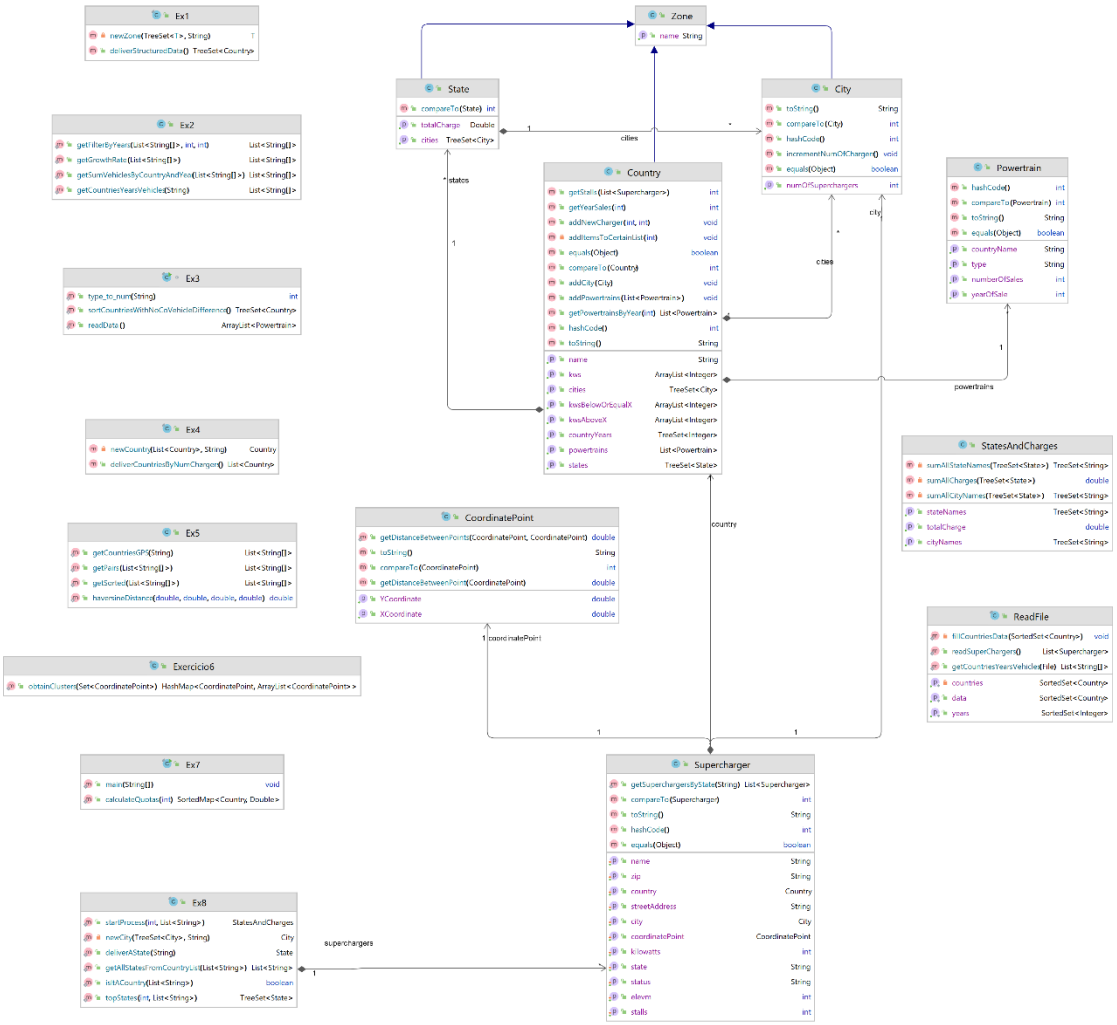

Figura 1. Diagrama de Classes

Através deste diagrama, é possível ver a forma como as várias classes criadas para a resolução do projeto interagem umas com as outras, os seus métodos e a sua importância no projeto.

Efetivamente, o diagrama de classes transmite uma ideia visual de como as classes se comportam e relacionam entre si, mas destaca também, o que cada classe necessita e a sua funcionalidade a partir do seu tipo de associação, atributo(s) e método(s), respetivamente.

Como é notável existe **uma classe para cada exercício** do projeto que não se relaciona com nenhuma ou com muito poucas classes, uma vez que estas classes contém apenas implementação e, por isso, são praticamente independentes neste modelo.

No entanto, tal não se verifica para o resto das classes , como conseguimos verificar a classe *Zones*, é a superclasse de *Country*, City e *State*. O grupo resolveu criar a classe Zones uma vez que cada subclasse a cima apresentada é uma zona , mas uma zona não pode ser considerada um pais ,cidade ou estado – conceito de herança.

Assim, o grupo simplificou o código não tendo de colocar do atributo "name", nas subclasses ,indo estas buscar o atributo a superclasse.

De seguida, conseguimos ver **a importância da classe Country** a partir do número alto de relações que este contém.

Com efeito os países são um conceito mencionado na maior parte dos exercícios, e, é também um parâmetro que esta presente em ambos os ficheiros que contêm as informações do trabalho, desta forma o grupo optou por realçar a sua importância.

No entanto, esta classe é construída por várias outras classes (associação por composição) como ***Powertrain, State ,City,*** uma vez que um pais é um conceito abrangente e contém objetos associados a ele que o constroem .

Por fim, não podemos esquecer de denotar restantes classes não mencionadas ***CoordinatePoint StatesAndCharges*** e ***ReadFile***: estas classes não simbolizam conceitos em si mas sim classes de suporte a outras classes em si como é o casso da classe CoordinatePoint, que fundamenta informação de coordenadas á classe City. O mesmo acontece para StatesAndCharges para a classe state, e, como o próprio nome indica a classe ReadFile contém somente implementação para a leitura e armazenamento de dados em ficheiro que é usada em certas classes de exercícios.

Exercício 1

Neste exercício, queremos devolver os carregadores de baterias numa estrutura de dados para cada país e o número de carregadores elétricos por cidade.

Como não foi especificado a ordem em que os itens têm de estar, decidimos organizar os países por ordem alfabética. Dentro de cada país, as cidades também estão organizadas por ordem alfabética.

```
1 usage  Diogo
public TreeSet<Country> deliverStructuredData(){
    List<Supercharger> superchargers = ReadFile.readSuperChargers();
    TreeSet<Country> countries = new TreeSet<>();
    TreeSet<City> firstCountrysCities = new TreeSet<>();
    firstCountrysCities.add(new City(superchargers.get(0).getCity().getName(), numOfSuperchargers: 1));
    countries.add(new Country(superchargers.get(0).getCountry().getName(), firstCountrysCities));
    for(Supercharger charger : superchargers){
        Country country = newZone(countries, charger.getCountry().getName());
        if(country != null){
            City city = newZone(country.getCities(), charger.getCity().getName());
            if(city != null){
                city.incrementNumOfChargers();
            } else {
                country.addCity(new City(charger.getCity().getName(), numOfSuperchargers: 1));
            }
        } else {
            TreeSet<City> newCountrysCities = new TreeSet<>();
            newCountrysCities.add(new City(charger.getCity().getName(), numOfSuperchargers: 1));
            countries.add(new Country(charger.getCountry().getName(), newCountrysCities));
        }
    }
    return countries;
}

2 usages  Diogo
private <T extends Zone> T newZone(TreeSet<T> zones, String item){
    for(T zone : zones){
        if(zone.getName().equals(item)) return zone;
    }
    return null;
}
```

Figura 2. Código do Exercício 1

Neste código, percorremos a lista de carregadores dada pelo método que lê o ficheiro *Excel*. Se o país de um dado carregador não estiver na nova lista, adicionamos, se já estiver, vemos se a dada cidade já está nesse objeto de país ou não, adicionando a cidade se não, e incrementando o número de carregadores dessa cidade se sim. Isto dá-nos o número de carregadores de cada cidade, cada uma contida no seu país.

Decidimos usar um *TreeSet* para armazenar esta lista. Por mais que sejam adicionados itens aos vários objetos depois de serem adicionados à lista, estes valores não alteram a sua posição na lista ordenada (mas sim o nome, que é dado no início e não muda). A ordenação é decidida pelo método *compareTo()* declarada na classe *Country* que implementa *Comparable*.

Exercício 2

Neste exercício, pretende-se conhecer a evolução do número de veículos elétricos dos vários países. Para isso determinamos a taxa de crescimento entre 2 anos do número de veículos elétricos nos vários países.

Como não foi especificado os anos decidimos escolher como ano inicial 2011 e ano final 2012. A taxa é dada por (último ano - primeiro ano) / primeiro ano.

.

```java
4 usages  ± 1221948 +1
public class Ex2 {
    1 usage
    static final int startYear = 2011;
    1 usage
    static final int finalYear = 2012;

    1 usage  ± 1221948 +1
    public static List<String[]> getCountriesYearsVehicles(String filePath) {
        List<String[]> data = new ArrayList<>();

        try (BufferedReader br = new BufferedReader(new FileReader(filePath))) {
            String line;
            boolean firstLine = true;

            while ((line = br.readLine()) != null) {
                if (firstLine) {
                    firstLine = false;
                    continue;
                }

                String[] parts = line.split( regex: ",");
                if (parts.length >= 4) {
                    String country = parts[0];
                    String year = parts[2];
                    String numVehicles = parts[3];
                    String[] extractedData = {country, year, numVehicles};
                    data.add(extractedData);
                }
            }
        } catch (IOException e) {
            e.printStackTrace();
        }

        return data;
    }

    1 usage  ± Tomas +1
    public static List<String[]> getFilterByYears(List<String[]> data, int startYear, int finalYear) {
        List<String[]> filteredData = new ArrayList<>();
        for (String[] entry : data) {
            int entryYear = Integer.parseInt(entry[1]);
            if (entryYear == startYear) {
                filteredData.add(entry);
            }
            if (entryYear == finalYear) {
                filteredData.add(entry);
            }
        }
        return filteredData;
    }
```

```
    1 usage   ± 1221948 +1
    public static List<String[]> getSumVehiclesByCountryAndYear(List<String[]> data) {
        List<String[]> summedData = new ArrayList<>();
        List<String[]> dataV2 = new ArrayList<>(data);

        for (int i = 0; i < dataV2.size(); i++) {
            String[] currentRow = dataV2.get(i);
            String country = currentRow[0];
            String year = currentRow[1];
            int sum = Integer.parseInt(currentRow[2]);

            for (int j = i + 1; j < dataV2.size(); j++) {
                String[] nextRow = dataV2.get(j);
                if (country.equals(nextRow[0]) && year.equals(nextRow[1])) {
                    sum += Integer.parseInt(nextRow[2]);
                    dataV2.remove(j);
                    j--;
                }
            }

            String[] summedRow = {country, year, String.valueOf(sum)};
            summedData.add(summedRow);
        }

        return Collections.unmodifiableList(summedData);
    }

    1 usage   ± 1221948
    public static List<String[]> getGrowthRate(List<String[]> data) {
        List<String[]> transformedData = new ArrayList<>();

        for (int i = 0; i < data.size(); i++) {
            String[] currentRow = data.get(i);
            String country = currentRow[0];
            String year = currentRow[1];
            int numVehicles2011 = Integer.parseInt(currentRow[2]);

            if (Integer.parseInt(year) == startYear) {
                for (int j = i + 1; j < data.size(); j++) {
                    String[] nextRow = data.get(j);
                    if (nextRow[0].equals(country) && Integer.parseInt(nextRow[1]) == finalYear) {
                        int numVehicles2012 = Integer.parseInt(nextRow[2]);
                        int difference = numVehicles2012 - numVehicles2011;
                        double division = (double) difference / numVehicles2011;

                        String[] transformedRow = {country, String.format("%.2f", division)};
                        transformedData.add(transformedRow);
                    }
                }
            }
        }

        return transformedData;
    }
}
```

Figuras 3 e 3(1). Código do Exercício 2

"*static final int startYear = 2011;*" e" static *final int finalYear = 2012;*" são constantes que definem o ano inicial e final para os quais os dados serão filtrados e processados.

"*getCountriesYearsVehicles*", método que lê um arquivo de dados *CSV* (especificado pelo caminho *filePath*) que contém informações sobre veículos em diferentes países e anos. Ele armazena esses dados em uma lista de *arrays* de *strings*, onde cada *array* contém informações sobre o país, o ano e o número de veículos.

"*getFilterByYears*", método que filtra os dados com base nos anos de início e fim especificados. Ele retorna uma lista com os dados que correspondem aos anos fornecidos.

"*getSumVehiclesByCountryAndYear*", método que calcula a soma do número de veículos para cada país em cada ano. Ele combina dados para o mesmo país e ano e retorna a soma em uma lista.

"*getGrowthRate*", método que calcula a taxa de crescimento dos veículos de 2011 para 2012 para cada país que possui dados para ambos os anos. Ele calcula a diferença no número de veículos entre 2012 e 2011 e divide pela quantidade de veículos em 2011. Os resultados são retornados em uma lista.

O código usa estruturas de repetição, como *loops for* e *while*, para percorrer os dados e realizar as operações necessárias. Ele também faz uso de manipulação de *strings* e conversões de tipos de dados para realizar cálculos e formatação de saída.

Exercício 3

Neste exercício pretende-se obter um conjunto de países nos quais **não houve aumento de vendas** de um ano para o outro associados aos anos em que tal sucedeu, e, adicionalmente duas diferenças de vendas para cada tipo de *Powertrain* nesses mesmo anos.

Para a resolução deste exercício, foram utilizados três métodos:

-***readData*** : método que lê e armazena os ficheiros em uma Lista do tipo *Powertrain*.

-***type_to_num***: pequeno método que suporta a implementação de *sortCountriesWithNoVehicleDifference*.

-***sortCounrtriesWithNoVehicleDifference***: contem a implementação do exercício.

```
public static String[][] sortCountriesWithNoCoVehicleDifference() throws FileNotFoundException, ClassCastException {
    ArrayList<Powertrain> data = readData();
    //mapping countries -> years -> array with the sales of each type (check type_to_num)
    HashMap<String, HashMap<Integer, Integer[]>> map = new HashMap<>();
    for (Powertrain element : data) {
        if (!map.containsKey(element.getCountryName())) {
            HashMap<Integer, Integer[]> minimap = new HashMap<>();
            Integer[] temp = new Integer[2];                            // sales for bev and sales for phev
            temp[type_to_num(element.getType())] = element.getNumberOfSales();
            temp[((type_to_num(element.getType())) + 1) % 2] = 0;
            minimap.put(element.getYearOfSale(), temp);
            map.put(element.getCountryName(), minimap);
        } else {
            HashMap<Integer, Integer[]> minimap = map.get(element.getCountryName());
            if (!minimap.containsKey(element.getYearOfSale())) {
                Integer[] temp = new Integer[2];
                temp[type_to_num(element.getType())] = element.getNumberOfSales();
                temp[((type_to_num(element.getType())) + 1) % 2] = 0;
                minimap.put(element.getYearOfSale(), temp);
            } else {
                minimap.get(element.getYearOfSale())[type_to_num(element.getType())] = element.getNumberOfSales();
            }
        }
    }
    int count = 0;                                      //number of countries with no sales increase
    String[][] declines = new String[data.size()][4];
    for (String key : map.keySet()) {                   // key = country
        HashMap<Integer, Integer[]> minimap = map.get(key);  // equals minimap from map to this minimap
        for (Integer year : minimap.keySet()) {
            int lastBev;
            int lastPhev;
            int lastyear = year - 1;
            if (lastyear == -1) {
                lastyear = year;
                continue;
            }
            if (minimap.containsKey(lastyear)) {
                lastBev = minimap.get(lastyear)[0];
                lastPhev = minimap.get(lastyear)[1];
            } else {
                lastBev = 0;
                lastPhev = 0;
            }
            if ((lastBev + lastPhev) >= (minimap.get(year)[0] + minimap.get(year)[1])) {
                declines[count][0] = key;
                declines[count][1] = lastyear + "/" + year;
                declines[count][2] = "BEV: " + (minimap.get(year)[0] - lastBev);
                declines[count][3] = "PHEV: " + (minimap.get(year)[1] - lastPhev);
                count += 1;
            }
            lastyear = year;
        }
    }
    String[][] final_declines = new String[count][4];

    System.arraycopy(declines,  srcPos: 0, final_declines,  destPos: 0, count);
    Arrays.deepToString(final_declines)
            .replace( target: "], ",  replacement: "],\n")
            .replace( target: "[[",  replacement: "[")
            .replace( target: "]]",  replacement: "]");


    return final_declines;
}
```

Figura 4. Código do Exercício 3

***ReadData**:*

Lê dados do ficheiro de tipo *csv* declarado como constante *csvFile*:

```
final class Ex3 {
    1 usage
    final static File csvFile = new File( pathname: "ev_sales.csv");
```

Figura 4(1). Código do Exercício 3

O método itera por todas linhas e colunas guardando cada linha como um objeto *Powertrain*, sabendo que os atributos da classe *Powertain* simbolizam os mesmos atributos das colunas do ficheiro (País, ano, tipo de *Powertrain*, saldos), respetivamente.

```
    public static ArrayList<Powertrain> readData() throws FileNotFoundException {
        ArrayList<Powertrain> powertrains = new ArrayList<>();
        Scanner csvScanner = new Scanner(csvFile);
        csvScanner.nextLine(); // Ignore header line
        while (csvScanner.hasNext() && !csvScanner.nextLine().equals(" ")) { //using iterator method hasnext

            String currentLine = csvScanner.nextLine();

            String[] currentLinePartitioned = currentLine.split( regex: ",");
            Powertrain savedPowerTrain = new Powertrain(currentLinePartitioned[0], currentLinePartitioned[1],
                    Integer.parseInt(currentLinePartitioned[2]), Integer.parseInt(currentLinePartitioned[3]));
            powertrains.add(savedPowerTrain);
        }
        return powertrains;
    }

}
```

Figura 4(2). Código do Exercício 3

***Type_to_num**:*

Como o próprio nome indica, este método retorna um número específico *0* caso a palavra recebida como parâmetro seja *BEV* e *1* caso seja outra palavra qualquer, no entanto no ficheiro existem apenas dois tipos de palavras para esta coluna, sendo estes *BEV* ou *PHEV* Portanto se o método retornar o número *1*, este representara somente a palavra "*PHEV*".

```
public static int type_to_num(String type) {
    if (type.equals("BEV")) {
        return 0;
    }

    return 1;
}
```

Figura 4(3). Código do Exercício 3

sortCounrtriesWithNoVehicleDifference:

Este método contém a logística e é responsável pela resolução do problema apresentado. O método começa por usar o método antes descrito acima: ***ReadData*** para poder obter os dados já lidos e armazenados em uma lista de *Powertrains* em uma variável: *data*.

```
ArrayList<Powertrain> data = readData();
```

Figura 4(4). Código do Exercício3

De seguida é criado um mapa (*map*) para futuramente poder ser armazenada a informação, onde as chaves são os **nomes dos países** e os valores são mapas internos. (*mínimap*)

```
HashMap<String, HashMap<Integer, Integer[]>> map = new HashMap<>();
```

Figura 4(5). Código do Exercício 3

O *minimap* usa anos como chaves e armazenam *arrays* de valores inteiros com as vendas de cada tipo de *powertrain* como valores (*BEV* e *PHEV*). O método percorre cada linha de dados do ficheiro e verifica se o país já existe no mapa criado (*map*).
Se não existir, ele cria um mapa interno (*minimap*) para esse país e inicializa as vendas para *BEV* com os valores da linha atual e as vendas para *PHEV* com zero.
Se o país já existir, verifica-se se o ano já existe no *minimap* desse país, se não existir, ele cria um novo *array* de vendas com os valores da linha atual e as vendas para *PHEV* com zero.
Se o ano já existir, o método atualiza as vendas correspondentes com os valores da linha atual.

```
for (Powertrain element : data) {
    if (!map.containsKey(element.getCountryName())) {
        HashMap<Integer, Integer[]> minimap = new HashMap<>();
        Integer[] temp = new Integer[2];                          // sales for bev and sales for phev
        temp[type_to_num(element.getType())] = element.getNumberOfSales();
        temp[((type_to_num(element.getType())) + 1) % 2] = 0;
        minimap.put(element.getYearOfSale(), temp);
        map.put(element.getCountryName(), minimap);
    } else {
        HashMap<Integer, Integer[]> minimap = map.get(element.getCountryName());
        if (!minimap.containsKey(element.getYearOfSale())) {
            Integer[] temp = new Integer[2];
            temp[type_to_num(element.getType())] = element.getNumberOfSales();
            temp[((type_to_num(element.getType())) + 1) % 2] = 0;
            minimap.put(element.getYearOfSale(), temp);
        } else {
            minimap.get(element.getYearOfSale())[type_to_num(element.getType())] = element.getNumberOfSales();
        }
    }
```

Figura 4(6). Código do Exercício 3

Após processar todos os dados, o método começa a contar quantos países não tiveram um aumento nas vendas, e, cria uma matriz de *strings* declarada como "*declines*" para guardar os resultados, onde cada linha representa um país que não teve um aumento nas vendas. A matriz tem a quatro colunas: nome do país, par de anos, diferença nas vendas de *BEV* diferença nas vendas de *PHEV*.

```
int count = 0;
String[][] declines = new String[data.size()][4];
```

O método percorre o mapa global de países e os respetivos dados internos, calcula as diferenças nas vendas entre os anos, e, verifica se houve um aumento nas vendas. No caso de tal não se verificar, os dados desse país serão adicionados à matriz "*declines*".

```
for (String key : map.keySet()) {                    // key = country
    HashMap<Integer, Integer[]> minimap = map.get(key);  // equals minimap from map to this minimap
    for (Integer year : minimap.keySet()) {
        int lastBev;
        int lastPhev;
        int lastyear = year - 1;
        if (lastyear == -1) {
            lastyear = year;
            continue;
        }
        if (minimap.containsKey(lastyear)) {
            lastBev = minimap.get(lastyear)[0];
            lastPhev = minimap.get(lastyear)[1];
        } else {
            lastBev = 0;
            lastPhev = 0;
        }
        if ((lastBev + lastPhev) >= (minimap.get(year)[0] + minimap.get(year)[1])) {
            declines[count][0] = key;
            declines[count][1] = lastyear + "/" + year;
            declines[count][2] = "BEV: " + (minimap.get(year)[0] - lastBev);
            declines[count][3] = "PHEV: " + (minimap.get(year)[1] - lastPhev);
            count += 1;
        }
        lastyear = year;
    }
}
```

Figura 4(7). Código do Exercício 3

Por fim, o método cria uma nova matriz chamada "*final_declines*", que representa a resposta do exercício:
onde copia os dados da matriz anterior “declines" e formata a representação em *string* da matriz "final declines" para que seja mais fácil na testagem.
Assim, esta matriz está preenchida com o formato pedido pelo exercício:
Países que não tiveram um aumento nas vendas e as diferenças nas vendas de *BEV* e *PHEV* em dois anos seguidos, e, para tal é então retornada.

```
    String[][] final_declines = new String[count][4];
    System.arraycopy(declines, srcPos: 0, final_declines, destPos: 0, count);
    Arrays.deepToString(final_declines)
            .replace( target: "], ", replacement: "],\n")
            .replace( target: "[[", replacement: "[")
            .replace( target: "]]", replacement: "]");

    return final_declines;
}
```

Figura 4(8). Código do Exercício 3

Exercício 4

Neste exercício, queremos uma lista com os países e os números de carregadores desse país com um número maior que *x*, menor ou igual *x*, e o total dos dois. Também queremos apresentar essa lista por ordem decrescente do número total de carregadores do determinado país.

```
5 usages
public static final int DEFINED_THRESHOLD = 150;
1 usage   Diogo
public List<Country> deliverCountriesByNumChargers() {
    List<Supercharger> superchargers = ReadFile.readSuperChargers();
    List<Country> countries = new ArrayList<>();
    ArrayList<Integer> kwsForFirstCountry = new ArrayList<>();
    kwsForFirstCountry.add(superchargers.get(0).getKilowatts());
    countries.add(new Country(superchargers.get(0).getCountry().getName(), kwsForFirstCountry, DEFINED_THRESHOLD));
    for (Supercharger charger : superchargers) {
        Country country = newCountry(countries, charger.getCountry().getName());
        if (country != null) {
            country.addNewCharger(charger.getKilowatts(), DEFINED_THRESHOLD);
        } else {
            ArrayList<Integer> kwsForFollowingCountries = new ArrayList<>();
            kwsForFollowingCountries.add(charger.getKilowatts());
            countries.add(new Country(charger.getCountry().getName(), kwsForFollowingCountries, DEFINED_THRESHOLD));
        }
    }
    Comparator<Country> comparator = new CountryComparator();
    countries.sort(comparator);
    return countries;
}
1 usage   Diogo
private Country newCountry(List<Country> countries, String item){
    for(Country country : countries){
        if(country.getName().equals(item)) return country;
    }
    return null;
}
```

Figura 5. Código do Exercício 4

Assim como no exercício 1, percorremos a lista de carregadores dada pelo método que lê o ficheiro *Excel* dado, adicionando um país à nova lista se este for novo.

Para cada entrada, vamos incrementar a lista com o nº de carregadores e, dependendo do valor da potência desse carregador, incrementa na lista dos carregadores com valores mais baixos ou iguais a *x* kWs ou na lista dos carregadores com valores mais altos que *x* kWs (x é definido no código pela constante *DEFINED_THRESHOLD*, neste caso é 150). Estas 3 listas mencionadas anteriormente são adjacentes ao determinado país, daí levando ao resultado pretendido.

Como temos de adicionar itens aos vários países depois de os adicionar à lista que vamos devolver, alterando o valor que decide a ordenação, decidimos não usar uma estrutura como um *TreeSet*, mas sim usar um *ArrayList* e orderná-lo no fim do processo de acordo com um critério de comparação estabelecido anteriormente numa classe *Comparator*.

Exercício 5

Neste exercício, pretende-se criar uma funcionalidade que devolva uma estrutura com os vários países e a sua respetiva mínima autonomia para circular nesse país, ordenada de forma decrescente pela mínima autonomia ou em caso de igualdade, por ordem alfabética do nome do país.

A mínima autonomia é a distância entre dois pontos de coordenadas de GPS.

```
4 usages  ± 1221948 +1 *
public class Ex5 {

    1 usage
    public static final double EARTH_RADIUS_KM = 6371;

    1 usage  ± 1221948
    public static List<String[]> getCountriesGPS(String filePath) {
        List<String[]> filteredData = new ArrayList<>();

        try (FileInputStream fis = new FileInputStream(filePath);
             Workbook workbook = new XSSFWorkbook(fis)) {
            Sheet sheet = workbook.getSheetAt( i: 0);

            boolean firstRow = true;

            for (Row row : sheet) {
                if (firstRow) {
                    firstRow = false;
                    continue;
                }

                Cell countryCell = row.getCell( i: 5);
                Cell gpsCell = row.getCell( i: 8);

                if (countryCell != null && gpsCell != null) {
                    String country = countryCell.getStringCellValue();
                    String gps = gpsCell.getStringCellValue();
                    filteredData.add(new String[]{country, gps});
                }
            }
        } catch (IOException e) {
            e.printStackTrace();
        }

        return filteredData;
    }

    1 usage  ± 1221948 +1
    public static List<String[]> getPairs(List<String[]> data) {
        boolean duplicate;
        List<String[]> duplicateTest = new ArrayList<>();
        List<String[]> result = new ArrayList<>();

        for (int i = 0; i < data.size() - 1; i++) {
            String[] entry1 = data.get(i);
            String gps1 = entry1[1];
            String[] parts1 = entry1[1].split( regex: ", ");
            String country1 = entry1[0];
            String lat1 = parts1[0];
            String lon1 = parts1[1];

            for (int j = i + 1; j < data.size(); j++) {
                duplicate = true;
                String[] entry2 = data.get(j);
                String gps2 = entry1[1];
                String[] parts2 = entry2[1].split( regex: ", ");
                String country2 = entry2[0];
                String lat2 = parts2[0];
                String lon2 = parts2[1];
```

```
            for (int k = 0; k < duplicateTest.size() - 1; k++) {
                String[] duplicateTestEntry = duplicateTest.get(k);
                if (gps1.equals(duplicateTestEntry[2]) && Objects.equals(gps2, duplicateTestEntry[1]) && country1.equals(country2)) {
                    duplicate = false;
                    break;
                }
            }
            if (duplicate && country1.equals(country2)) {
                duplicateTest.add(new String[]{country1, gps1, gps2});
                double distance = haversineDistance(Double.parseDouble(lat1), Double.parseDouble(lon1), Double.parseDouble(lat2), Double.parseDouble(lon2));
                result.add(new String[]{country1, lat1, lon1, lat2, lon2, String.valueOf(distance)});
            }
        }
    }

    return result;
}

2 usages  ± 1221948 +1
public static double haversineDistance(double lat1, double lon1, double lat2, double lon2) {
    double dLat = Math.toRadians(lat2 - lat1);
    double dLon = Math.toRadians(lon2 - lon1);
    double a = Math.sin(dLat / 2) * Math.sin(dLat / 2) +
            Math.cos(Math.toRadians(lat1)) * Math.cos(Math.toRadians(lat2)) *
                    Math.sin(dLon / 2) * Math.sin(dLon / 2);
    double c = 2 * Math.atan2(Math.sqrt(a), Math.sqrt(1 - a));
    double distance = EARTH_RADIUS_KM * c;
    String formattedDistance = String.format("%.2f", distance);
    formattedDistance = formattedDistance.replace( target: ",",  replacement: ".");
    return Double.parseDouble(formattedDistance);
}

1 usage  ± Tomas +1 *
public static List<String[]> getSorted(List<String[]> data) {
    List<String[]> sortedTestData = new ArrayList<>(data);

    sortedTestData.sort((o1, o2) -> {
        double value1 = Double.parseDouble(o1[o1.length - 1]);
        double value2 = Double.parseDouble(o2[o2.length - 1]);
        int valueComparison = Double.compare(value2, value1);

        if (valueComparison != 0) {
            return valueComparison;
        } else {
            return o1[0].compareTo(o2[0]);
        }


    });

    return sortedTestData;
}

}
```

Figuras 6 e 6(1). Código do Exercício 5

"*EARTH_RADIUS_KM*" é uma constante definida com um valor de 6371, que representa o raio médio da Terra em quilômetros.

"*etCountriesGPS*", método que recebe como argumento o caminho de um arquivo do *Excel* (XLSX) e retorna uma lista de *arrays* de *strings*. O objetivo deste método é ler o arquivo do *Excel*, extrair dados de determinadas colunas (país e coordenadas *GPS*) e armazená-los na lista *filteredData*.

"*getPairs*", método que recebe uma lista de *arrays* de *strings* (data) como entrada e procura pares de localizações geográficas dentro da lista que pertençam ao mesmo país e não sejam duplicados. Para cada par de localizações geográficas não duplicadas, calcula a distância entre elas usando a fórmula da distância *haversine* e as adiciona à lista *result*. Ele também usa uma lista *duplicateTest* para rastrear os pares já processados e evitar duplicatas.

"*haversineDistance*", método que calcula a distância entre dois pontos na superfície da Terra usando a fórmula da distância *haversine*. Recebe como entrada as latitudes e longitudes de dois locais e retorna a distância entre eles em quilômetros.

"*getSorted*", método que recebe uma lista de *arrays* de *strings* (data) que contém informações sobre locais geográficos e suas distâncias, e retorna uma lista ordenada com base nas distâncias e nos nomes dos países. Ele usa uma função de comparação personalizada para ordenar os dados em ordem decrescente de distância e, em caso de empate, em ordem crescente de nomes de países.

Exercício 6

O objetivo do exercício 6 é obter a lista dos pontos GPS que se encontram mais próximos de um dado ponto de interesse. Por outras palavras, fornece-se uma lista de pontos de interesse e para cada um deles é retornada a lista dos pontos, ordenada do que se encontra mais próximo para o que se encontra mais afastado. Para tal elaborei o seguinte método:

```
24  public static HashMap<CoordinatePoint, ArrayList<CoordinatePoint>> obtainClusters(Set<CoordinatePoint> pointsOfInterest) {
25      HashMap<CoordinatePoint, ArrayList<CoordinatePoint>> clusters = new HashMap<>();
26      ArrayList<CoordinatePoint> coordinatePointsInFile = new ArrayList<>();
27      List<Supercharger> superchargerList = ReadFile.readSuperChargers();
28      superchargerList.forEach(supercharger -> coordinatePointsInFile.add(supercharger.getCoordinatePoint()));
29      for (CoordinatePoint poi: pointsOfInterest) {
30          // Comparing based on the distance between the poi and each point in the Excel file.
31          coordinatePointsInFile.sort(Comparator.comparingDouble(poi::getDistanceBetweenPoint));
32          ArrayList<CoordinatePoint> coordinatePointsToPoi = new ArrayList<>(coordinatePointsInFile);
33          clusters.put(poi, coordinatePointsToPoi);
34      }
35      return clusters;
36  }
```

Figura 7. Código do Exercício 6

A primeira observação a fazer é o tipo de estrutura escolhida para armazenar a informação, como se pode observar pelo tipo de retorno do método escolhi armazenar a informação num HashMap que mapeia a coordenada de um ponto de interesse a uma lista de pontos, isto é, para cada ponto de interesse temos um valor que é a lista dos pontos no ficheiro ordenados do mais próximo ao ponto de interesse ao mais afastado. Depois também é importante notar que o único parâmetro do método é um set com os pontos de interesse para os quais queremos calcular a lista ordenada. O método funciona da seguinte forma:

1. Leem-se os Superchargers do ficheiro Excel (linha 27);
2. Para cada Supercharger obtém-se o ponto correspondente e armazena-se num ArrayList (linha28);
3. Finalmente para cada ponto de interesse no Set que é passado como parâmetro ordena-se a ArrayList com os pontos do ficheiro de acordo com a distância ao ponto de interesse (linha 31), salva-se essa informação numa nova ArrayList (linha 32), finalmente adiciona-se a entrada no HashMap (linha 33). Este processo é repetido para cada ponto de interesse é de notar que a lista na linha 31 é sempre atualizada de acordo com o ponto de interesse que está a ser iterado.

Exercício 7

Para o exercício 7 era pedido que fosse calculada uma quota para cada país num determinado ano, sendo que a quota é dada pela expressão:

$$\frac{Número\ de\ Stalls}{Número\ de\ veículos} \times 1000$$

Para alcançar os objetivos do exercício elaborei o seguinte método:

```
27 @    public static SortedMap<Country, Double> calculateQuotas(int year) throws FileNotFoundException {
28          SortedMap<Country, Double> quotas = new TreeMap<>();
29          List<Supercharger> superchargers = ReadFile.readSuperChargers();
30          SortedSet<Country> countries = ReadFile.getData();
31          for (Country country : countries) {
32              double quota = 0;
33              List<Powertrain> powertrains = country.getPowertrains();
34              for (Powertrain powertrain : powertrains) {
35                  if (powertrain.getYearOfSale() == year) {
36                      quota += (double) country.getStalls(superchargers) / powertrain.getNumberOfSales();
37                  }
38              }
39              if (quota != 0) {
40                  quotas.put(country, quota * 1000);
41              }
42          }
43          return quotas;
44      }
```

Figura 8. Código do Exercício 7

Decidi guardar a informação num SortedMap que mapeia um país à sua quota. A decisão de usar um SortedMap advém da maneira como foi lida e guardada a informação do ficheiro CSV: Estabeleceu-se que o critério de comparação para cada país seria o seu próprio nome e foi assim lida a informação do ficheiro. Como tal, de modo que a leitura fosse correta, este método também usa um SortedMap cuja ordenação é alfabética relativa ao nome de cada país. O único parâmetro aceite pelo método é o ano para o qual queremos calcular as quotas de cada país. Eis como o método funciona:

1. Leem-se as informações necessárias dos ficheiros fornecidos (linhas 29/30);
2. Depois para cada país efetuam-se as seguintes operações:
3. Obtêm-se os Powertrains desse país (linha 33);
4. Para cada Powertrain do país específico verifica-se se o seu ano é correspondente ao mesmo passado como parâmetro (linha 35) se isto for verdade calcula-se a quota do mesmo (linha 36). É de notar que a quota teve de ser inicializada porque existem diferentes tipos de Powertrains, mas com o mesmo ano, de modo que o cálculo correto é somar a quota desses diferentes tipos.
5. Finalmente guarda-se a quota se for diferente de zero (linha 39) no SortedMap criado na linha 28 (linha 40).

Exercício 8

Neste exercício, queremos fazer um top *n* de estados, sendo o valor *n* e a lista de estados ou de países o input para esta função. O top é organizado por quais estados têm mais potência elétrica total, e o objeto que devolvemos contém os nomes dos estados, a potência total (calculada por *Número de Stalls x Potência*) e os nomes das cidades que participam neste valor obtido.

```java
public static StatesAndCharges startProcess(int n, List<String> names){
    if(isItACountry(names)){
        names = getAllStatesFromCountryList(names);
    }
    return new StatesAndCharges(topStates(n, names));
}

private static boolean isItACountry(List<String> names){
    for(Supercharger charger : superchargers)
        for(String name : names)
            if (charger.getCountry().getName().equals(name))
                return true;
    return false;
}

private static List<String> getAllStatesFromCountryList(List<String> names){
    List<String> statesNames = new ArrayList<>();
    for(Supercharger charger : superchargers) {
        for(String name : names)
            if (charger.getCountry().getName().equals(name) && !statesNames.contains(charger.getState()))
                statesNames.add(charger.getState());
    }
    return statesNames;
}

private static TreeSet<State> topStates(int n, List<String> stateNames){
    TreeSet<State> topNStates = new TreeSet<>();
    List<State> allStates = new ArrayList<>();
    for(String name : stateNames)
        allStates.add(deliverAState(name));
    Collections.sort(allStates);
    for(int i = 0; i < n; i++)
        topNStates.add(allStates.get(i));
    return topNStates;
}

private static State deliverAState(String name){
    TreeSet<City> cities = new TreeSet<>();
    double totalValue = 0;
    for(Supercharger charger : superchargers){
        if(charger.getState().equals(name) && charger.getStatus().equals("Open")){
            City city = newCity(cities, charger.getCity().getName());
            if(city != null)
                city.incrementNumOfChargers();
            else
                cities.add(new City(charger.getCity().getName(),  1));
            totalValue += charger.getKilowatts() * charger.getStalls();
        }
    }
    return new State(name, totalValue, cities);
}

private static City newCity(TreeSet<City> cities, String item){
    for(City city : cities){
        if(city.getName().equals(item)) return city;
    }
    return null;
}
```

Figura 9. Código do Exercício 8

O código vai pela lista de carregadores e obtém todos os estados pedidos (se o utilizador inseriu países, o algoritmo vê quais estados estão nesses países) lá presentes, assim como as suas cidades e as suas cargas. Devolve um objeto que contém os estados e as cidades por ordem alfabética, assim como a carga somada de todos os estados presentes neste novo objeto.

Recursos

Informação Global do Projeto:

- ISEP Moodle Project description File:
  - *ESINF23-24-TP1.pdfESINF23-24-TP1.pdf*
- Other Moodle Data files:
  - *carregadores_europa.csv*
  - *ev_sales.csv*
  - *carregadores_europa.xlsx*

Recursos adicionais usados para implementações de métodos Java:

- Sharan, K. (2018). *Java language features: With modules, streams, threads, I/O, and Lambda Expressions*.
- YouTube Video: " **java Unit Testing with JUnit - Tutorial - How to Create And Use Unit Tests**" by *Coding with John* Link: https://youtu.be/gN29Y600k5g