Número: **Nome:**

Teste 2 de Introdução à Programação

Duração: 2 horas

Departamento de Informática, FCT NOVA

Sábado, 8 de janeiro de 2022

Este teste tem 5 grupos e 15 folhas.
Pode usar as costas das folhas para rascunho.

Regras do Teste

- O teste é sem consulta. Não pode consultar quaisquer elementos para além deste enunciado.
- Não pode usar dispositivos eletrónicos (como, por exemplo, calculadoras, telemóveis, *tablets*, *smartwatches* e portáteis). Não pode ter telemóveis junto ao corpo (por exemplo, nos bolsos).
- O teste é resolvido no enunciado.
- Não pode usar folhas de rascunho separadas. Use as costas das folhas. Elas serão ignoradas durante a correção.
- Em cima da mesa, só pode ter o enunciado, o documento de identificação, material de escrita (caneta, lápis, borracha) e, eventualmente, uma garrafinha de água.
- As respostas podem ser escritas a lápis.
- O enunciado está agrafado e não se permite que seja desagrafado.
- Antes de começar a resolver cada grupo, leia o enunciado das perguntas do grupo com atenção, do princípio até ao fim.
- Responda a cada pergunta na caixa atribuída para o efeito. **Se faltar espaço, coloque a continuação nas costas da mesma folha, indicando na caixa da pergunta que a resposta continua nas costas.**
- No início do teste, **identifique todas as folhas do enunciado com o seu número de aluno e o seu nome abreviado**.
- A interpretação do enunciado faz parte da avaliação. Por esse motivo, não há esclarecimento de dúvidas. Se suspeitar que o enunciado tem algum erro, deve avisar o docente.
- Se pretender que o seu teste não seja avaliado, escreva “Desisto” na zona de identificação desta página de rosto.
- Só pode sair da sala quando o teste terminar (2 horas após ter começado).

No Final do Teste

- Verifique que **todas as folhas** estão identificadas com o seu número e o seu nome.

Número: **Nome:**

Enunciado comum aos Grupos 1 a 4

Pretende-se implementar uma aplicação para dar apoio a uma loja de venda de automóveis. A aplicação regista o número de exemplares vendidos de cada modelo de automóvel, calcula os valores monetários correspondentes às vendas e produz listagens.

A aplicação mantém informação sobre os automóveis disponíveis para entrega imediata e sobre os automóveis já vendidos. A loja apenas pode vender um exemplar de determinado modelo quando esse exemplar existir em stock.

Cada modelo de automóvel tem um preço específico, que nunca muda, e não são feitos descontos.

A aplicação é constituída pelas classes **CarModel**, **CarOutlet**, **MinPriceCarModelIterator** e **Main**.

Cada objeto da classe CarModel descreve um modelo de automóvel usando os seguintes atributos: nome da marca (*brandName*), nome do modelo (*modelName*), preço de venda (*price*), número de exemplares disponíveis na loja (*stock*); contagem de exemplares já vendidos (*sold*).

Note que um objeto CarModel não representa um automóvel específico mas sim uma categoria de automóveis, todos com as mesmas propriedades, por exemplo “Bugatti Chiron Sport” ou “Fiat Abarth”. É por esse motivo que um objeto CarModel inclui o número de exemplares disponíveis para entrega. A aplicação só usa um objeto CarModel por cada modelo registado na loja.

A classe **CarOutlet** guarda a informação sobre os automóveis que a loja dispõe para entrega, bem como os automóveis já vendidos. Disponibiliza diversas operações, nomeadamente a inserção de um novo modelo, o incremento do *stock*, a consulta do stock, cálculo das vendas totais (soma dos preços dos exemplares vendidos), um mecanismo de navegação nos modelos registados (método **iterator**).

A classe **MinPriceCarModelIterator** implementa um iterador sobre objetos CarModel com preço maior ou igual a determinado valor.

A classe **Main** manda carregar os dados iniciais de um ficheiro de texto e produz listagens. O interpretador de comandos está fora do âmbito deste enunciado.

Número: **Nome:**

Grupo 1 [1.5 valores]

O código da classe CarModel é mostrado abaixo, quase completo. Escreva a parte que falta, ou seja, apenas o corpo do método **compareTo**.

```
public class CarModel {
    private String brandName;     // nome da marca (e.g., Honda, Renault, etc.)
    private String modelName;     // nome do modelo (e.g., Clio, Uno, etc.)
    private int price;            // preço do modelo (valor fixo; não há descontos)
    private int stock;            // número de exemplares disponíveis na loja
    private int sold;             // número de exemplares já vendidos

    /**
     * construtor: Inicializa um modelo com os valores indicados
     *                            e zero automóveis vendidos.
     * @pre brandName != null && modelName != null && price > 0 && stock >= 0
     */
    public CarModel(String brandName, String modelName, int price, int stock) {
       this.brandName = brandName;
       this.modelName = modelName;
       this.price = price;
       this.stock = stock;
       this.sold = 0;
    }

    /**
     * construtor: Semelhante ao construtor anterior, mas recebendo mais
     * um argumento indicando o número de automóveis já vendidos.
     * @pre brandName != null && modelName != null && price > 0
     *       && stock >= 0 && sold >= 0
     */
    public CarModel(String brandName, String modelName, int price, int stock, int sold){
       ... // não programar
    }

    public String getBrandName() { return brandName; }
    public String getModelName() { return modelName; }
    public int getPrice() { return price; }
    public int getStock() { return stock; }
    public int getSold() { return sold; }

    /**
     * Aumenta o stock.
     * @pre: quantity > 0
     */
    public void addToStock(int quantity) {
       stock += quantity;
    }
```

Número: **Nome:**

```
    /**
     * Regista a venda de mais um exemplar do modelo.
     * Portanto, decrementa as existências e incrementa as vendas.
     * @pre: getStock() > 0
     */
    public void sell() {
       this.stock--;
       this.sold++;
    }

/**
 * Define uma relação de ordem no conjunto dos modelos. A comparação entre dois
 * modelos (designados por this e por other) é feita:
 * 1) com base no nome da marca, por ordem lexicográfica crescente;
 * 2) se marca for igual, segundo nome do modelo, por ordem lexicográfica crescente;
 * 3) se os nomes da marca e modelo forem iguais, segundo o preço (crescente).
 *
 * @return o resultado segue a convenção dos métodos compareTo do Java, ou seja:
 *              0, se this for igual a other;
 *              um valor menor que 0, se this for menor que other;
 *              um valor maior que 0, se this for maior que other
 */
public int compareTo(CarModel other) {
```

```
}
}
```

Número: **Nome:**

Grupo 2 [7 valores]

Para a classe CarOutlet, defina as constantes, variáveis de instância, corpo do construtor e corpo dos métodos pedidos. A capacidade (número de modelos suportados) é especificada no construtor.

Evite repetição de código. Na última página deste grupo, dispõe de uma caixa própria para incluir os métodos auxiliares que achar convenientes.

```
public class CarOutlet {

// Coloque aqui as constantes e variáveis de instância.
```

```
/**
 * construtor: Inicializa o objeto com a capacidade especificada e
 *                                                zero modelos registados.
 * @pre: capacity > 0
 */
public CarOutlet(int capacity) {
```

```
}

/**
 * @return número de modelos registados.
 */
public int size() {
```

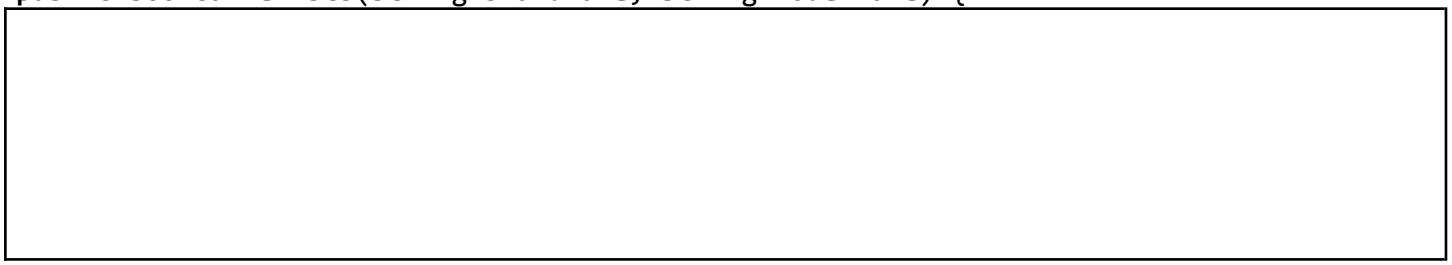
```
}
```

Número: **Nome:**

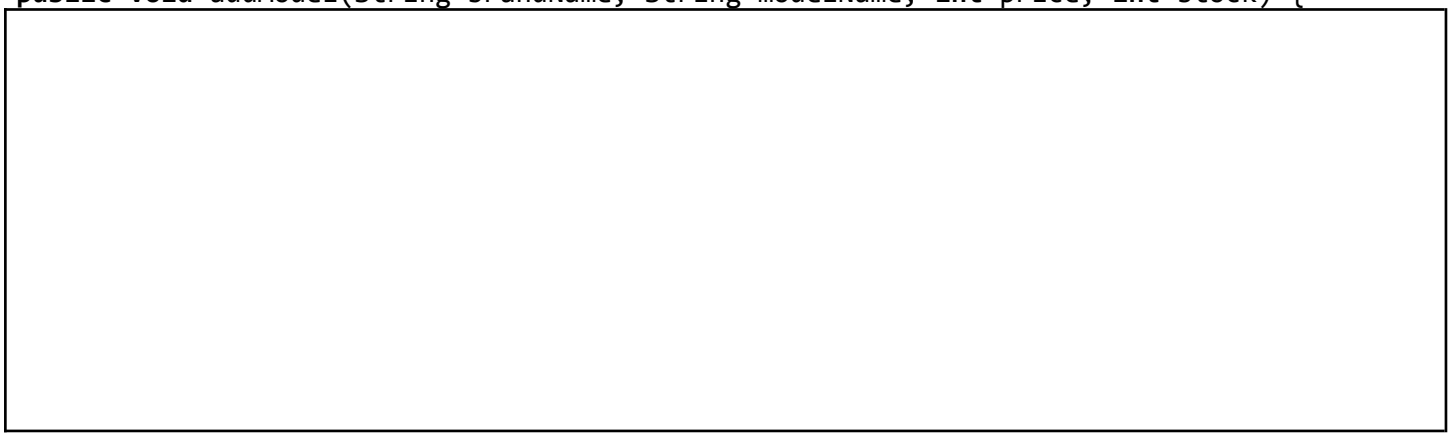
```
/**
 * @return true se a capacidade estiver esgotada; false caso contrário.
 */
public boolean isFull() {
```

```
}

/**
 * @return true se não houver qualquer modelo registado; false caso contrário
 */
public boolean isEmpty() {
```

```
}

/**
 * @pre: brandName != null && modelName != null
 * @return true se existir o modelo identificado pelo nome da marca e pelo nome do
 *              modelo (mesmo que o seu stock seja zero); false caso contrário
 */
public boolean exists(String brandName, String modelName) {
```



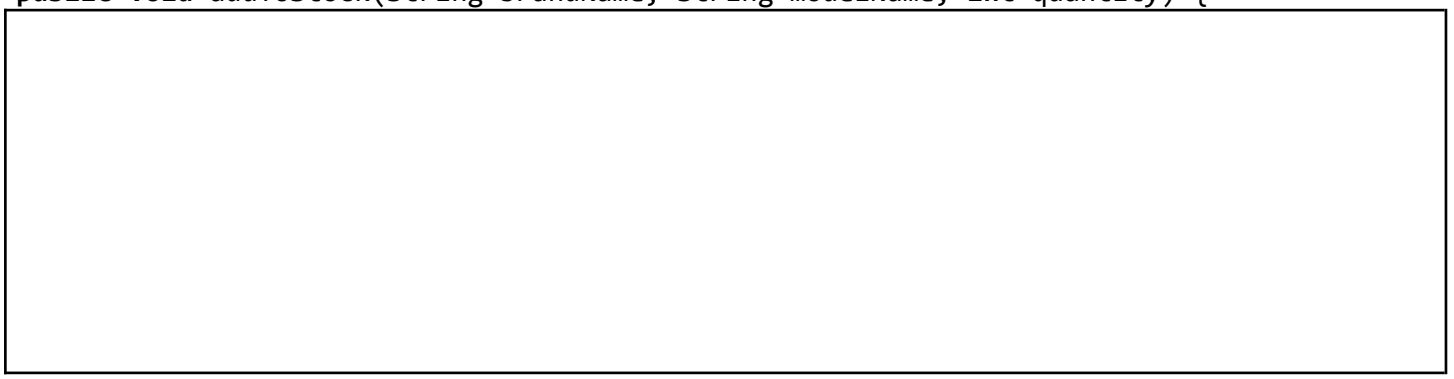
```
}

/**
 * Regista um novo modelo de automóvel.
 * @pre brandName != null && modelName != null && price > 0
 *              && !exists(brandName, modelName) && !isFull() && stock >= 0
 */
public void addModel(String brandName, String modelName, int price, int stock) {
```



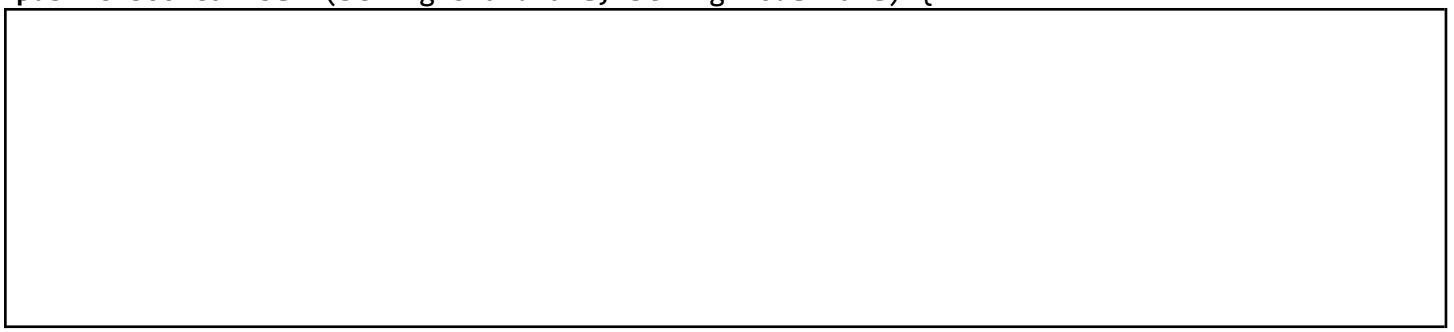
```
}
```

Número: **Nome:**

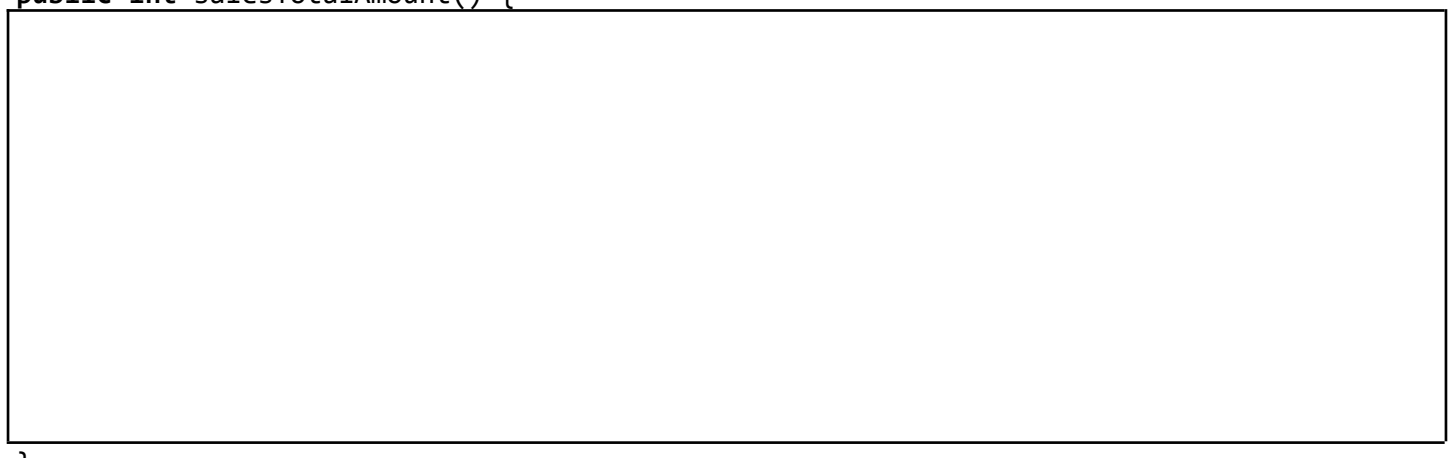
```
/**
 * Aumenta o stock de um modelo já registado. Não regista novos modelos.
 * @pre: brandName != null && modelName != null
                && exists(brandName, modelName) && quantity > 0
 */
public void addToStock(String brandName, String modelName, int quantity) {
```



```
}

/**
 * Regista a venda de mais um exemplar do modelo especificado, caso exista stock.
 * @return true, caso a operação tenha sucesso; false, caso o modelo possua
 *         stock zero. Nesse segundo caso, o método não altera nada.
 * @pre: brandName != null && modelName != null && exists(brandName, modelName)
 */
public boolean sell(String brandName, String modelName) {
```



```
}

/**
 * @return valor total das vendas (soma dos preços dos exemplares vendidos)
 *                                                          até ao momento.
 */
public int salesTotalAmount() {
```



```
}
```

Número: **Nome:**

```
/**
 * @return um iterador com filtro para objetos da classe CarModel.
 * @pre: minimumPrice > 0
 */
public MinPriceCarModelIterator iterator(int minimumPrice) {
          return new MinPriceCarModelIterator(..., minimumPrice);
}

// Métodos auxiliares que entenda conveniente acrescentar na classe CarOutlet.
```

Número: **Nome:**

Grupo 3 [4 valores]

Implemente, na classe MinPriceCarModelIterator, um iterador com filtro para objetos CarModel. Pretende-se um iterador, que devolva, pela ordem original, a sequência de elementos que cumpram a seguinte condição: o preço é igual ou superior a um dado valor mínimo. A implementação do iterador deve estar completamente contida na classe MinPriceCarModelIterator e ser coerente com o código do método **iterator** da classe CarOutlet.

```
public class MinPriceCarModelIterator {

// Coloque aqui as constantes e as variáveis de instância.
```

```
/**
 * construtor: Inicializa o iterador.
 * @pre: models != null && size >= 0 && minimumPrice > 0
 */
public MinPriceCarModelIterator(CarModel[] models, int size, int minimumPrice) {
```

```
}

/**
 * @return true se ainda existir algum elemento da sequência para next() retornar;
 *         false caso contrário.
 */
public boolean hasNext() {
```

```
}
```

Número: **Nome:**

```
/**
 * @return o próximo elemento CarModel que cumpre a condição do filtro.
 * @pre:
 */
 public CarModel next() {
```

```
}
```

```
// Métodos auxiliares que entenda acrescentar na classe MinPriceCarModelIterator:
```

Número: **Nome:**

Grupo 4 [4 valores]

Parte 1. No contexto da classe Main, implemente o método privado **listing**, que recebe um objeto CarOutlet e apresenta na consola todos os modelos com um preço maior ou igual a um dado valor mínimo.

O método **printCarModel** já está programado e lista na consola toda a informação sobre um modelo.

Não é preciso ter implementado o iterador do Grupo 3 para resolver este problema. É suficiente que saiba usar corretamente um iterador, seguindo os princípios estudados nas aulas.

```
public class Main {
   ...

// Escreve um modelo na consola
private static void printCarModel(CarModel car) {
    ... // não programe
}

// Lista todos os modelos registados na loja, que tenham um preço maior
//                                      ou igual a um dado valor mínimo.
private static void listing(CarOutlet outlet, int minimumPrice) {
```

```
}
```

Número: **Nome:**

Parte 2. No contexto da classe CarOutlet, adicione um segundo construtor que leia os dados a partir dum ficheiro de texto. Os dados no ficheiro são os seguintes:

1. A primeira linha contém o número de modelos de automóveis referidos nas linhas seguintes.
2. Depois, cada modelo é representado pelos seguintes cinco valores, um por linha: (1) nome da marca do automóvel; (2) nome do modelo; (3) preço do modelo; (4) número de exemplares disponíveis para entrega; (5) número de exemplares já vendidos.

Eis um curto exemplo que mostra o formato pretendido para o ficheiro:

```
2
Fiat
Abarth
64000
5
11
Peugeot
3008 Hybrid
44000
3
7
```

Garante-se que o ficheiro é válido, ou seja que todos os dados estão no formato correto e que cada modelo de automóvel ocorre apenas uma vez no ficheiro.

```
/**
 * construtor de CarOutlet: Lê os dados a partir do ficheiro com o nome indicado.
 */
public CarOutlet(String filename) throws FileNotFoundException {
```

```
}
```

Número: **Nome:**

Grupo 5 [3.5 valores]

Parte 1. O objetivo deste problema é acrescentar à classe DataAnalysis (usada nas aulas) um método para determinar quantas posições mini-max existem na matriz de dados. Diz-se que uma posição (i,j) da matriz é mini-max se o valor dessa posição for o mínimo da linha i e o máximo da coluna j.

Exemplo: A seguinte matriz tem três posições mini-max, correspondentes às três ocorrências do valor **8**:

```
0 1 2 3
4 5 6 7
8 8 8 9
```

Recorde as variáveis de instância da classe DataAnalysis:

```
public class DataAnalysis {
  private int[][] data;
  private int rows, cols;
}
```

Pedimos-lhe para organizar a solução usando os três métodos indicados abaixo

```
// Cria um vetor com os mínimos das várias linhas da matriz data.
private int[] getRowsMins() {
```

```
}
```

Número: **Nome:**

```
// Cria um vetor com os máximos das várias colunas da matriz data.
private int[] getColsMaxs() {
// Não programe este método, que tem uma estrutura semelhante à do método anterior.
}

/**
 * Conta o número de posições mini-max da matriz, com a ajuda dos métodos anteriores.
 */
public int countMinMax() {
    int[] rowsMins = getRowsMins();
    int[] colsMaxs = getColsMaxs();
}

// Métodos auxiliares que entenda acrescentar na classe DataAnalysis.
```

Número: **Nome:**

Parte 2. De acordo com uma convenção gráfica do português, nunca se escreve mais do que um espaço em branco a seguir a uma vírgula. Programe um método para eliminar de um texto todos os espaços em branco em excesso, isto é que transgridam a regra.

O texto é passado para o método através dum vetor de caracteres. O método altera o vetor diretamente. Como o tamanho do vetor não pode mudar, usam-se '-', no final, para indicar posições do vetor não usadas. Exemplo, com remoção de três espaços em branco:

Antes: **abc, ,,, a,b,c, z**

Depois: **abc, ,,, a,b,c, z---**

```
// Remove espaços em excesso a seguir a vírgulas.
private static void clean(char[] text) {
```

```
// Métodos auxiliares que entenda acrescentar.
```