1º Teste de Introdução à Programação

Duração: 2 horas

Departamento de Informática, FCT NOVA
20 de Novembro de 2021

(O teste tem **18** páginas e **5** grupos)

Número: _____ **Nome:** _____

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).
- Não pode ter folhas de rascunho. Use os espaços para rascunhos no enunciado.
- Em cima da mesa, só pode ter o enunciado, o documento de identificação e material de escrita (caneta, lápis, borracha).
- As respostas podem ser escritas a lápis.
- Não pode desagrafar o enunciado.
- 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 numa zona livre do enunciado, **numa folha do mesmo grupo da pergunta**, indicando na caixa da pergunta a página onde a resposta continua.
- No início do teste, identifique todas as folhas do enunciado no local apropriado, com o seu número de aluno e o seu nome.
- 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.
- Só pode sair da sala quando o teste terminar (2 horas após ter começado).
- Se pretender que o seu teste não seja avaliado, escreva "Desisto" na zona de identificação desta página.

No Final do Teste

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

Grupo 1 [2 valores]

Considere a classe `Grupo1`:

```
public class Grupo1 {
  private int v1;
  private int v2;

  public Grupo1( int v1, int v2 ) {
    if ( v1 <= v2 ) {
      this.v1 = v1;
      this.v2 = v2;
    }
    else {
      this.v1 = v2;
      this.v2 = v1;
    }
  }

  public int pass( int n ) {
    int res = n;
    if ( res < v1 )
      res = v1;
    else if ( res > v2 )
      res = v2;
    return res;
  }

}
```

Indique nas caixas abaixo o que seria escrito na consola após a execução de cada grupo de instruções, executadas em sequência pela ordem indicada. Caso não seja escrito nada, escreva "Não escreve nada".

```
Grupo1 s1 = new Grupo1(18, 6);
Grupo1 s2 = new Grupo1(4, 20);
Grupo1 s3 = new Grupo1(5, 2);
int x = 3;
```

```
x = s1.pass(x);
System.out.println(x);
```

```
x = s2.pass(x);
System.out.println(x);
```

```
x = s3.pass(x);
System.out.println(x);
```

Número: Nome:

Notas Aplicáveis aos Grupos de 2 a 5:

- O que se pede concretamente para resolver está bem identificado (por caixas) no enunciado.
- Pode preencher qualquer caixa, mesmo que não tenha preenchido outras.
- Todas as zonas fora das caixas podem ser usadas para rascunho.

Grupo 2 [5 valores]

A classe `CoinPurse` vai ajudar a gerir as compras de Natal. Um objeto `CoinPurse` contém informação sobre o dinheiro disponível (`double`), o preço máximo permitido para qualquer presente, o número máximo de presentes que podem ser comprados e informação estatística sobre os presentes já comprados.

A classe `CoinPurse` deve incluir os seguintes métodos:

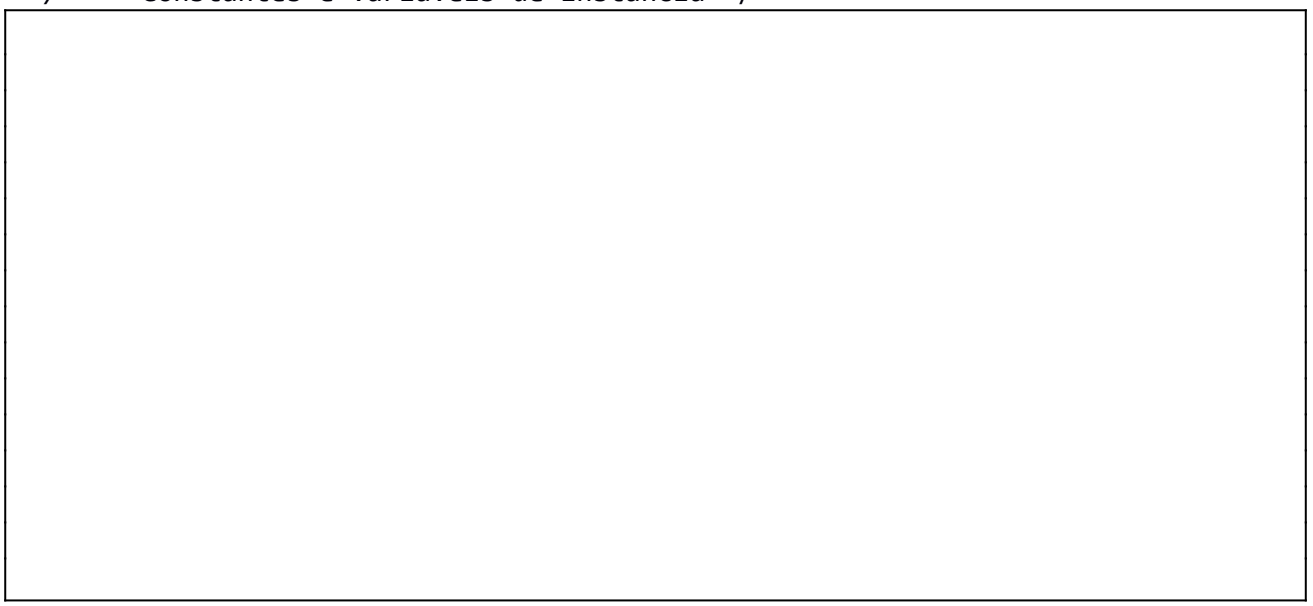
- `getBalance()`: consulta o saldo (o montante disponível) no porta-moedas;
- `getMaxAmountAllowed()`: consulta o preço máximo permitido para qualquer presente;
- `getMaxNrGiftsAllowed()`: consulta o número máximo de presentes que podem ser comprados;
- `getNrGiftsBought()`: consulta o número de presentes já comprados;
- `getAverageGiftCost()`: consulta o preço médio dos presentes já comprados;
- `getHighestGiftPrice()`: consulta o preço do(s) presente(s) mais caro(s) já comprado(s);
- `registerBuy(price)`: regista a compra de mais um presente, pressupondo que é possível efetuar essa compra. A compra pode ser realizada, se (1) o preço do presente for positivo, se (2) houver saldo disponível, se (3) o preço do presente for inferior ou igual ao preço máximo permitido para qualquer presente e se (4) o número de presentes comprados for inferior ao número máximo de presentes que podem ser comprados.

Quando se cria um porta-moedas pode-se fornecer, em alternativa:

- O dinheiro disponível para todos os presentes, o montante máximo a despender por compra e o número máximo de presentes que podem ser comprados.
- O dinheiro disponível para todos os presentes e o número máximo de presentes que podem ser comprados. Neste caso, o valor máximo a despender por presente é o quociente entre o dinheiro disponível e o número máximo de compras.

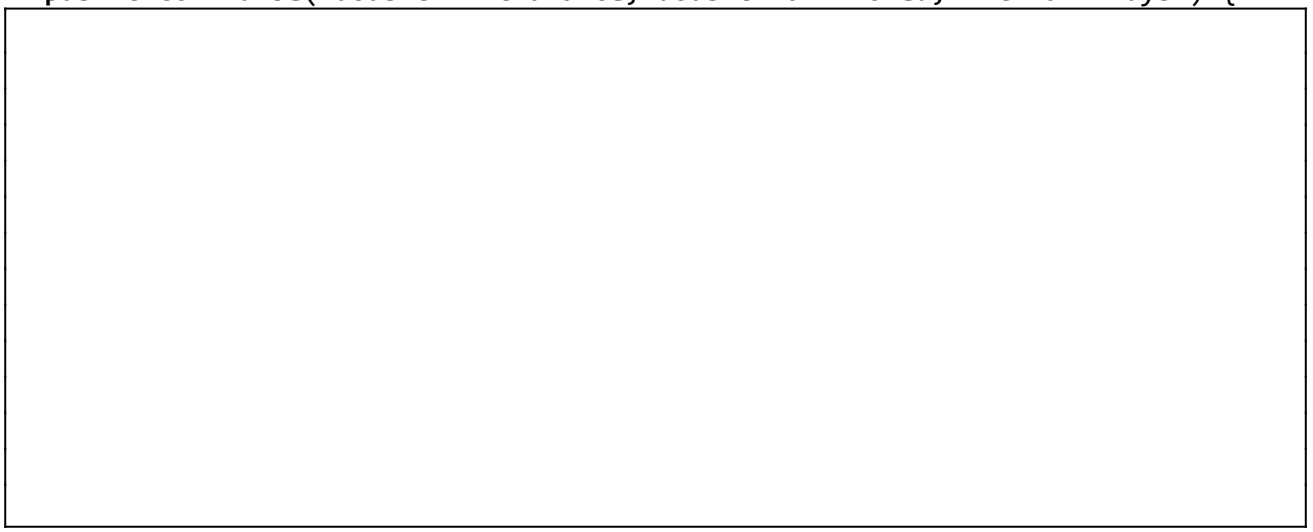
Preencha as caixas em branco na definição da classe `CoinPurse` com as constantes, variáveis de instância, precondições, corpos de construtores e corpos de métodos de modo a que a classe implemente a funcionalidade descrita em cima.

```
public class CoinPurse {
  /**   Constantes e variáveis de instância */
```



```
  /**
   * Construtor: inicializa o objeto com base nos valores recebidos.
   * @param initBalance: o dinheiro disponível para todos os presentes.
   * @param maxAllowed: o valor máximo a gastar em qualquer presente.
   * @param maxNrBuys: o número máximo de presentes que podem ser comprados.
   * @pre:
```

```
   */
  public CoinPurse( double initBalance, double maxAllowed, int maxNrBuys ) {
```

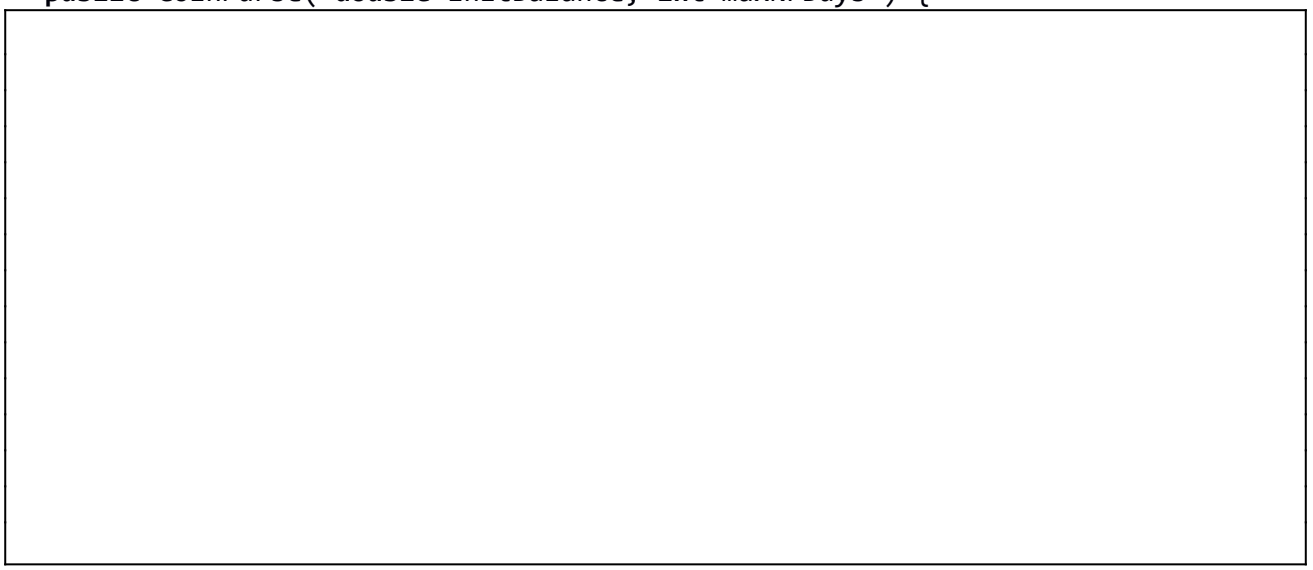


```
  }
```

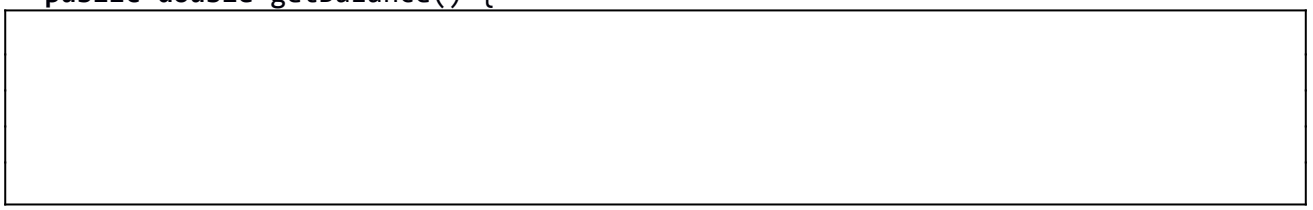
Número: Nome:

```
/**
 * Construtor: inicializa o objeto com base nos valores recebidos.
 * @param initBalance: o dinheiro disponível para todos os presentes.
 * @param maxNrBuys: o número máximo de presentes que podem ser comprados.
 * @pre:
```

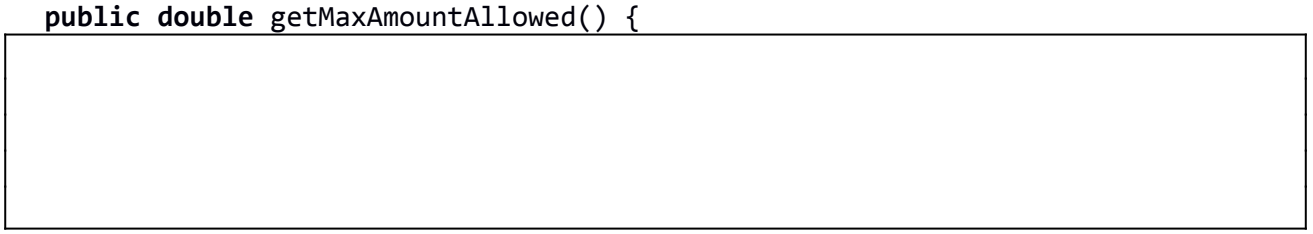
```
 */
public CoinPurse( double initBalance, int maxNrBuys ) {
```



```
}

/**
 * @return o montante disponível no porta-moedas.
 */
public double getBalance() {
```

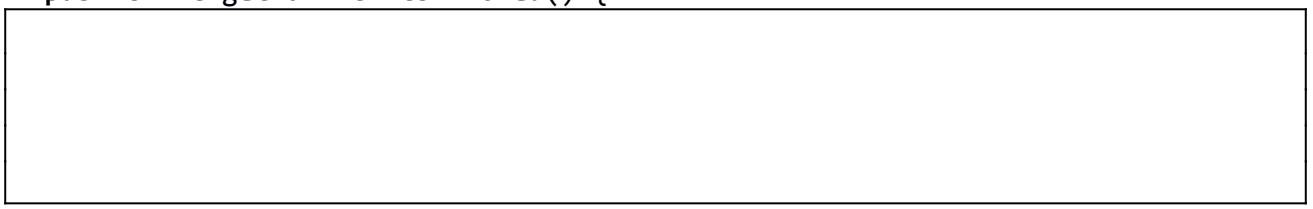


```
}

/**
 * @return o preço máximo permitido para qualquer compra.
 */
public double getMaxAmountAllowed() {
```

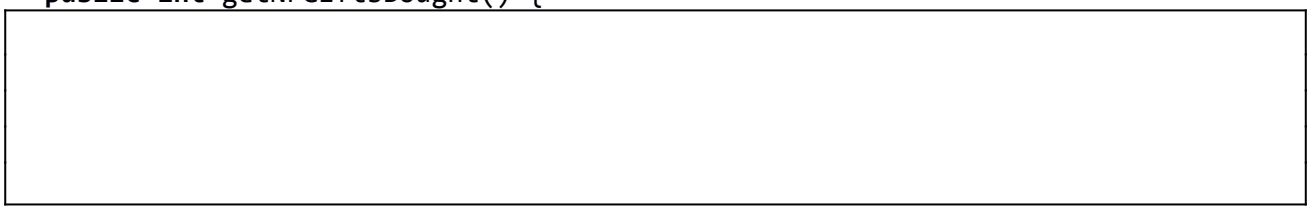


```
}
```

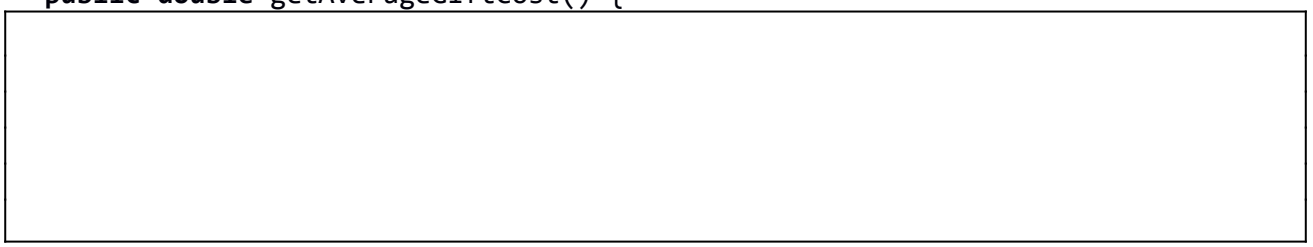
```
/**
 * @return o número máximo de presentes que podem ser comprados.
 */
public int getMaxNrGiftsAllowed() {
```



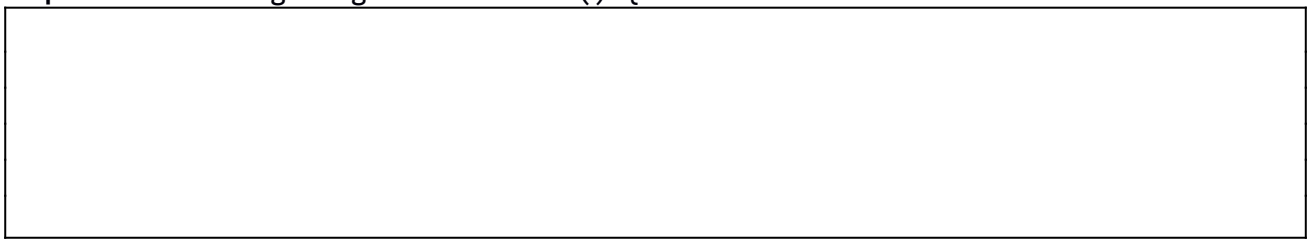
```
}

/**
 * @return o número de presentes já comprados.
 */
public int getNrGiftsBought() {
```



```
}

/**
 * @return o preço médio dos presentes já comprados.
 * @pre: this.getNrGiftsBought() > 0
 */
public double getAverageGiftCost() {
```



```
}

/**
 * @return o preço do(s) presente(s) mais caro(s) já comprado(s).
 * @pre: this.getNrGiftsBought() > 0
 */
public double getHighestGiftPrice() {
```



```
}
```

Número: Nome:

```
/**
 * Regista a compra de mais um presente, assumindo que é possível efetuar
 * essa compra. A compra pode ser efetuada caso se verifiquem as condições
 * descritas em cima (i.e., na primeira página deste Grupo).
 * @param price: o preço do presente.
 * @pre:
 */
public void registerBuy( double price ) {
```

```
    }

}
```

Grupo 3 [4 valores]

A classe `Main` cria e usa um objeto instância da classe `CoinPurse`, definida no grupo anterior. Note que pode resolver este grupo mesmo que não tenha respondido às questões do Grupo 2.

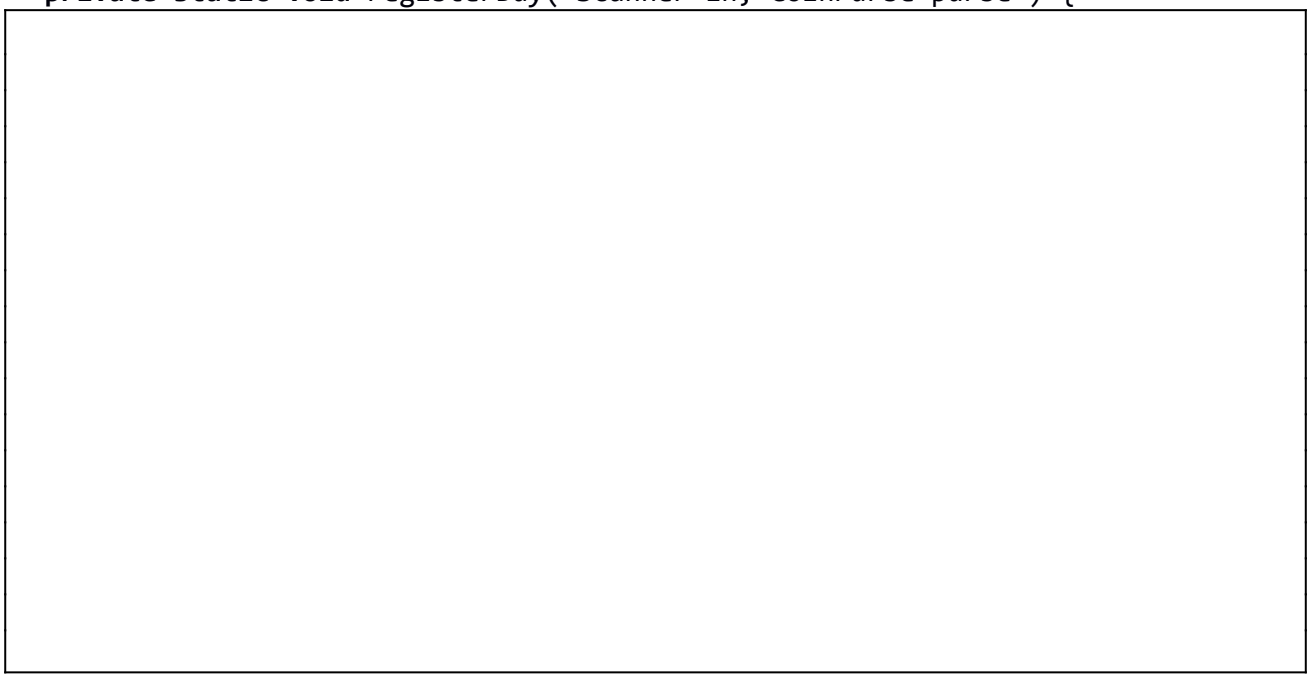
Complete o corpo dos dois métodos seguintes, cujas funcionalidades estão descritas nos respetivos comentários.

```
public class Main {

  /**
   * Cria e retorna um objeto CoinPurse, assumindo que os dados de entrada
   * satisfazem as precondições da criação do objeto.
   * @param balance:   o dinheiro disponível para todos os presentes.
   * @param maxPrice:  o valor máximo a gastar em qualquer presente.
   * @param maxBuys:   o número máximo de presentes que podem ser comprados.
   * @return o objeto criado de acordo com os dados recebidos.
   * @pre: (omitidas deliberadamente)
   */
  private static CoinPurse createPurse( double balance, double maxPrice,
                                        int maxBuys ) {
```

```
  }
```

Número: Nome:

```
  /**
   * Lê do input o preço do presente e, se for possível efetuar a compra,
   * regista-a no porta-moedas. Caso não seja possível efetuar a compra,
   * o método não produz qualquer efeito.
   * @param in:     o input.
   * @param purse:  o objeto porta-moedas.
   * @pre: in != null && purse != null
   */
  private static void registerBuy( Scanner in, CoinPurse purse ) {
```



```
  }

}
```

(Esta página é para rascunho)

Número: Nome:

Grupo 4 [7 valores]

A classe `Pilgrim` representa um peregrino que faz caminhadas diárias durante um **período** (i.e., durante um dado número de dias consecutivos). Quando se regista a distância percorrida num dia, esse dia corresponde ao seguinte do último já registado. Ou seja, o primeiro registo corresponde ao primeiro dia do período, o segundo registo corresponde ao segundo dia do período e assim sucessivamente. Conhece-se a distância máxima (em km) que o peregrino deve percorrer por dia, chamada a **distância máxima diária recomendada**.

Quando se cria um objeto `Pilgrim`, fornece-se:

- a distância máxima diária recomendada;
- o número de dias do período (o número previsto de dias de caminhada).

A classe `Pilgrim` tem os seguintes métodos:

- `getAdvisableDist()`: consulta a distância máxima diária recomendada;
- `getDaysOfWalk()`: consulta o número de dias do período;
- `getWalkingDays()`: consulta o número de caminhadas registadas;
- `endReached()`: indica se a caminhada chegou ao fim (se o número de dias registados é o número de dias previstos);
- `registerDay(dist)`: regista a distância percorrida (em km) em mais um dia de caminhada;
- `getDaysAboveAdvisable()`: consulta o número de dias em que distância percorrida excedeu a recomendada;
- `getDistInterval(dayI, dayF)`: consulta a distância total percorrida entre dois dias dados;
- `getDistsFromStart()`: retorna as distâncias totais percorridas desde o primeiro dia.

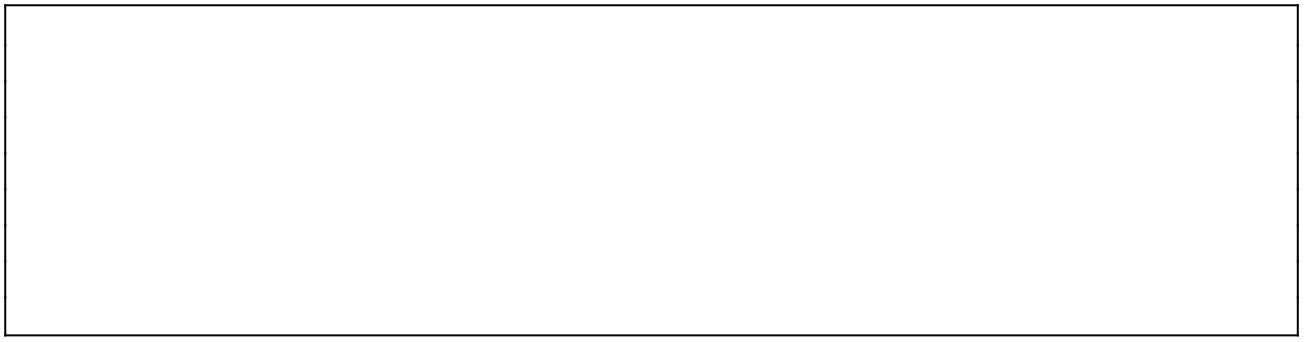
Sempre que sejam especificados dias, considere que 0 (zero) denota o primeiro dia, 1 (um) representa o segundo dia, etc. Para exemplificar os dois últimos métodos, considere que o peregrino `pilg` caminhou durante 5 dias, tendo percorrido:

- 5 km no primeiro dia,
- 7 km no segundo dia,
- 3 km no terceiro dia,
- 6 km no quarto dia e
- 4 km no quinto dia.

Neste caso:

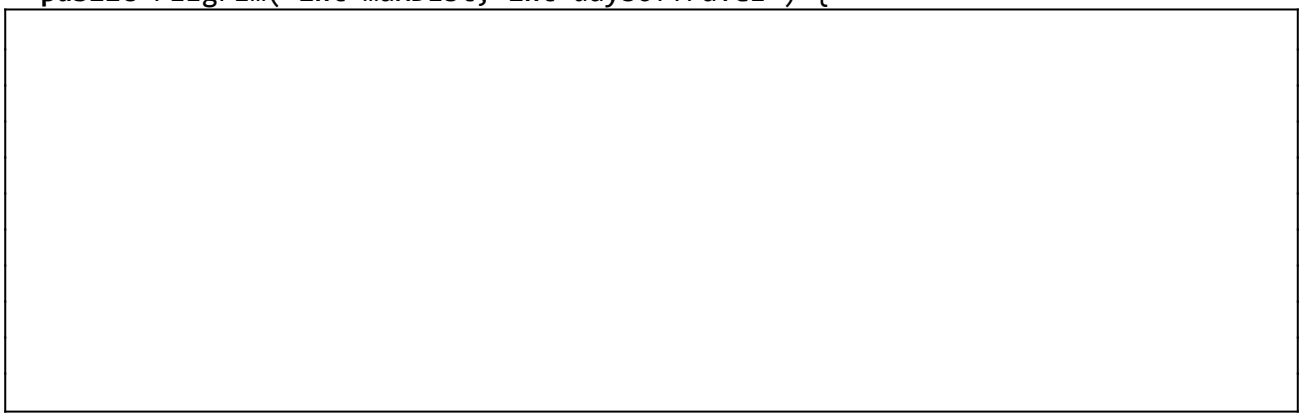
- `pilg.getDistInterval(1, 3)` corresponde à distância total percorrida desde o segundo dia até ao quarto dia (inclusive), que é 16 (16 = 7 + 3 + 6);
- `pilg.getDistsFromStart()` é um vetor (chamado `vetExemplo` neste exemplo) com 5 posições, que contém a distância total percorrida desde o primeiro dia até ao dia correspondente:
  - `vetExemplo[0]` tem 5
  - `vetExemplo[1]` tem 12 (12 = 5 + 7)
  - `vetExemplo[2]` tem 15 (15 = 5 + 7 + 3)
  - `vetExemplo[3]` tem 21 (21 = 5 + 7 + 3 + 6)
  - `vetExemplo[4]` tem 25 (25 = 5 + 7 + 3 + 6 + 4)

```
public class Pilgrim {
    /** Constantes e variáveis de instância */
```



```
  /**
   * @param maxDist:      a distância máxima diária recomendada.
   * @param daysOfTravel: o número de dias do período.
   * @pre:
```

```
   */
  public Pilgrim( int maxDist, int daysOfTravel ) {
```



```
  }

  /**
   * @return a distância máxima diária recomendada.
   */
  public int getAdvisableDist() {
```

```
  }
```

Número: Nome:

```
/**
 * @return o número de dias do período.
 */
public int getDaysOfWalk() {
```

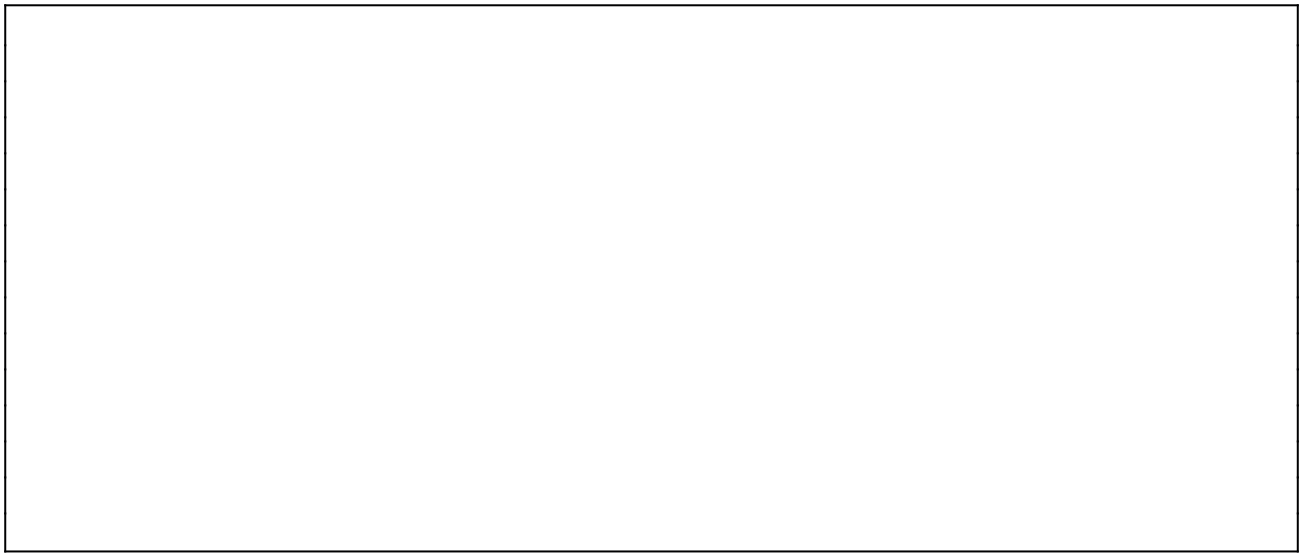
```
}

/**
 * @return o número de caminhadas registadas.
 */
public int getWalkingDays() {
```

```
}

/**
 * Indica se a caminhada chegou ao fim.
 * @return true, se o número de dias registados é o número de dias
 *         previstos; false, no caso contrário.
 */
public boolean endReached() {
```

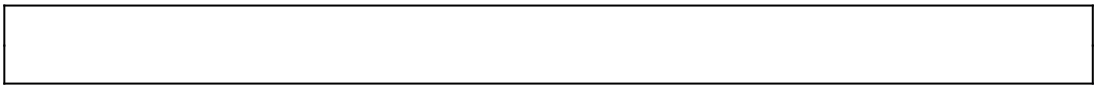
```
}

/**
 * Regista a distância percorrida em mais um dia de caminhada.
 * @param dist: a distância percorrida
 * @pre: !this.endReached() && dist >= 0
 */
public void registerDay( int dist ) {
```

```
}
```

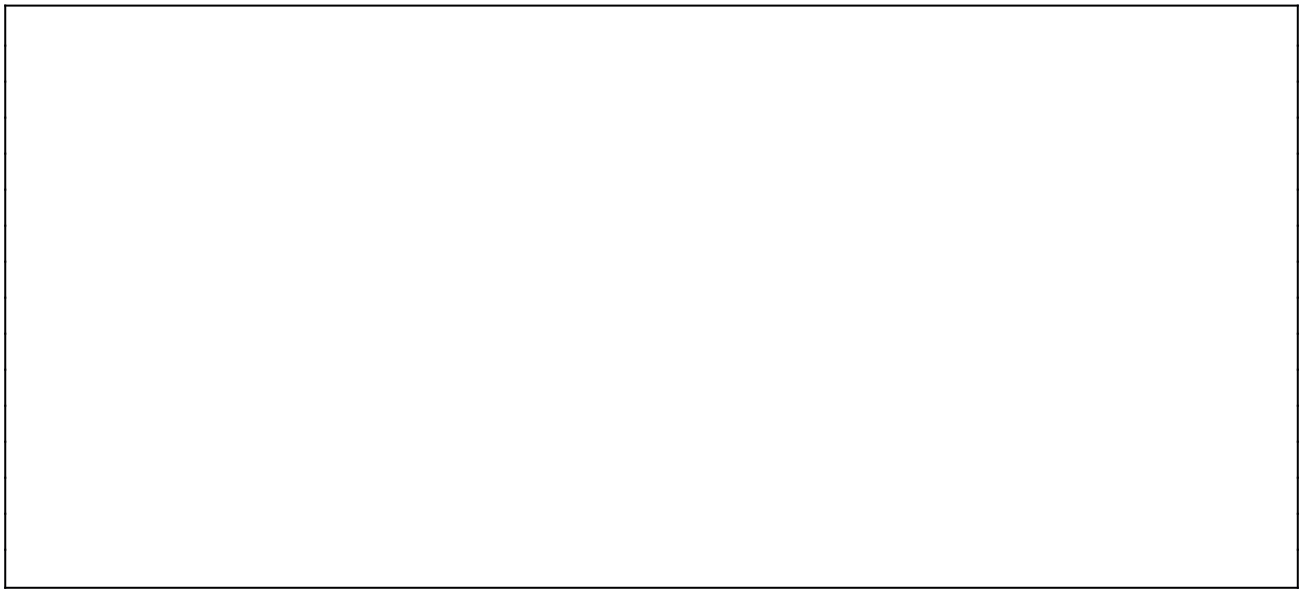
```
/**
 * @return o número de dias em que distância percorrida excedeu a
 *         recomendada para o peregrino.
 */
public int getDaysAboveAdvisable() {
```



```
}

/**
 * Calcula a distância total percorrida entre dois dias dados
 * (que inclui a distância percorrida nesses dias).
 * @param dayI: o dia inicial.
 * @param dayF: o dia final.
 * @return a distância total percorrida entre os dias dayI e dayF.
 * @pre:
```



```
 */
public int getDistInterval( int dayI, int dayF ) {
```



```
}
```

Número: Nome:

```
  /**
   * Retorna um vetor com as distâncias totais percorridas desde o primeiro
   * dia até cada um dos dias em que foi registada uma caminhada.
   * O vetor tem tantas posições quantas as caminhadas registadas.
   * @return o vetor com as distâncias
   * @pre: this.getWalkingDays() > 0
   */
  public int[] getDistsFromStart() {
```

```
  }

}
```

(Esta página é para rascunho)

Grupo 5 [2 valores]

Considere a seguinte definição recursiva da potência, $\mathcal{P}(b, n)$, onde a base b é um número real diferente de zero e o expoente n é um número inteiro não negativo:

$$\mathcal{P}(b,n) = \begin{cases} 1, & \text{se } n = 0; \\ \mathcal{P}(b, \frac{n}{2}) \times \mathcal{P}(b, \frac{n}{2}), & \text{se } n \geq 1 \text{ e } n \text{ é par}; \\ b \times \mathcal{P}(b, n-1), & \text{se } n \geq 1 \text{ e } n \text{ é ímpar}. \end{cases}$$

Implemente em Java um **método recursivo**, chamado `power`, que recebe a base b e o expoente n e retorna o valor de b^n. Recorde que, se x e y forem dois números inteiros (com $y \neq 0$), a expressão $x\%y$ calcula o resto da divisão inteira de x por y.

```
/**
 * Calcula a potência de base real e expoente natural.
 * @param base:     a base da potência.
 * @param exponent: o expoente da potência.
 * @return a base elevada ao expoente.
 * @pre:  base != 0 && exponent >= 0
 */
public static double power( double base, int exponent ) {









}
```

(Esta página é para rascunho)