LP II - 2015.2 / 1ª Lista de Exercícios

Em todos os exercícios o aluno deve atender aos requisitos enunciados. Métodos e variáveis auxiliares podem ser criadas e usadas, desde que pertinentes. O aluno deve necessariamente empregar e explorar as características de orientação a objetos do Java:

- Encapsulamento (incluindo modificadores de acesso),
- Herança (de classe e interface) e polimorfismo; Classes e métodos abstratos.
- Sobrecarga de métodos;
- Tratamento e geração de Exceções;
- Uso das classes básicas (Object, por exemplo);
- Classes / pacotes

A lista deve ser entregue, deixando os arquivos com código fonte e os executáveis (class) na máquina virtual. O professor vai avaliar o código fonte, cumprimento dos requisitos e o programa em execução, na própria máquina.

O aluno deve criar um diretório L1, dentro do seu diretório home. Dentro de L1, criar um diretório para cada exercício com os seguintes nomes: E1, E2, E3, etc. O nome da classe com o método *main()* de cada exercício deve ser Ex1, Ex2, Ex3. Mesmo que no enunciado esteja sendo solicitado um outro nome (mude o nome solicitado por estes). A correção vai ser, o máximo possível, automatizada.

Todos os exercícios devem pertencer ao pacote default. O aluno não deve colocar as classes desenvolvidas dentro de pacotes. Isso só deve ser feito quando explicitamente solicitado no exercício.

Para não termos problemas, não use caracteres acentuados, nem no código nem nos comentários.

Para todos os exercícios devem ser tratadas as exceções numéricas, de conversão, número de argumentos, de input/output, etc., além das exceções discutidas no enunciado.

"Tratamento" de exceções no estilo abaixo não serão considerados, muito mentos o *throws* sem razão (ou melhor, para se livrar das exceções).

```
try{
    // codigo sem tratamento
}catch (Exception e)
{
    System.out.println("Erro")
}
```

Exercício 1

O DNA humano tem 3 bilhões de bases pareadas, então laboratórios recortam esse DNA em pedaços menores e mais fáceis de ser computados, com menos bases. Esses recortes não são feitos no DNA original, são feitos em copias, por isso às vezes parte do DNA se encontra em dois

recortes, o seu trabalho é criar um programa em Java que vai concatenar dois ou mais recortes de forma a não repetir as partes iguais, essa aplicação é encontrada em muitas outras áreas, então faça seu código funcionar com qualquer par de *strings* (N recortes)

Suas *strings* serão passadas por argumento na linha de comando.

Para este primeiro exercício, pode colocar o programa todo dentro do método *main*.

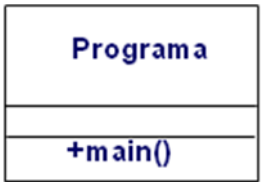

Figura 1. Diagrama de classe - Ex1

Exemplo:

```
java Ex1 ATCGAT GATTCTAGA
ATCGATTCTAGA

java Ex1 "N rec" "ortes c" "s com qualqu" "alquer par" "ar de Strings"
N recortes com qualquer par de Strings
```

Exercício 2:

Fazer um programa que calcule a área de 3 tipos de figuras, círculos, retângulos e triângulos dependendo da entrada recebida, e imprima seus valores de acordo.

- Caso a figura for um círculo é passado o raio do circulo
- Caso a figura for um retângulo é passado base e altura
- Caso for um triângulo o comprimento dos lados do Triângulo serão passadas

Os valores devem ser passados como argumentos de linha de comando, valores com tipo *real*.

Você vai identificar o tipo de figura com base no número de argumentos passados.

O cálculo da área deve ser feito em um método de classe, chamado *calcula*.

```
private static real calcula(real r)
private static real calcula(real b, double a)
private static real calcula(real l1, real l2, real l3)
```

Chame estes métodos para fazer os cálculos (tornando o programa mais modular).

No caso de ser um triângulo, classifique se ele é: equilátero, isósceles ou escaleno. Para isso crie um outro método estático e use o mesmo.

Utilize técnicas de criticas de dados e/ou tratamento de exceções para capturar problemas de entrada: número insuficiente de argumentos, número de excessivo de argumentos, argumentos que não sejam convertíveis para real [argumento(s) inválido(s)], argumentos que não formam um triângulo.

Exemplos de execução:

```
>java Ex1 1.3
A area do circulo e': XXX unidades de area.

>java Ex1 3.0 4.0 5.0
A area do triangulo e': XXX unidades de área.
O triangulo e' isosceles.

>java Ex1
Número de argumentos insuficiente

>java Ex1 0 1 3.7 9.5
Número de argumentos excessivo

>java Ex1 a 0 0xD
1o argumento, "a", nao é numero
3o argumento, "0xD", nao é numero
```

Para esta experiência, o programa principal também pode ser codificado todo no método *main* (sabendo que isso não é incentivado), como na Figura 1. Mas outros métodos serão chamados, a partir do *main*, como já colocado no enunciado.

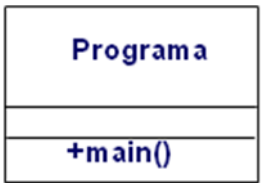

Figura 2. Diagrama de classe – Ex2

Exercício 3:

a) Crie uma classe *Produto*, que contenha os campos privados:

Codigo:	int ou String, identificador do produto
Nome:	Nome do produto
Preco:	Preço normal do produto
Estoque:	Quantidade do produto em estoque
Vendidos:	Quantidade de produtos vendidos
Promocao:	Percentual abatido do preço, 0 se não houver promoção no produto

O construtor vai receber todos os atributos, de forma a ser criado com todos eles com algum valor.

O método *toString*() da classe *Produto* devera mostrar todos os dados mas com o preço corrigido com o campo promoção.

b) Crie uma classe chamada *MinhaListaOrdenavel*.

1) Esta classe vai conter um objeto da classe *ArrayList*, do pacote java.util, que implementa a interface *Collection*. Este objeto vai colecionar objetos da classe *Produto* e, em cima dele, vamos fazer as várias ordenações.

2) Crie métodos *add (Produto p)* e *Produto get (index i)* para adicionar e resgatar, respectivamente, objetos da Classe produto do *ArrayList* interno.

3) Crie múltiplas formas de ordenar sua lista de *Produto* (maior preço, menor preço, mais vendido, A-Z, Z-A, etc.).

Para isso vamos usar a interface *Comparator* e um truque de programação em Java: definir uma classe dentro de outra e criar uma instância dela – tudo embutido. Para cada tipo de ordenação que queremos ter, ou seja comparando valores inteiros ou Strings, temos que ter uma classe que herda da *Comparator* diferente. A que compara inteiros vai pegar um campo, digamos o Código ou o Preço, de dois objetos da classe Produto, subtrair um do outro e isso será uma indicação se um é menor ou maior que o outro. Se for da classe *String*, podem usar os métodos da própria classe *String* para fazer a comparação.

Exemplo:

```
public Comparator precoC = new Comparator () {
   public int compare (Object p1, Object p2){ // recebe objetos Produto como Object
      double pf1, pf2;
      pf2 = (Produto) p2.preco*(1-(double) (Produto) p2.promo/100);
      pf1 = (Produto) p1.preco*(1-(double) (Produto) p1.promo/100);
      return (int)Math.round (pf2 – pf1);
   }
};
```

Observe que vamos ter um objeto chamado *precoC* "dentro" do (objeto) *MinhaListaOrdenavel*. Da mesma forma vamos ter vários outros comparadores encapsulados.

4) Crie um método *ordena*, que vai receber uma constante, de uma "tabela" de constantes que você vai criar dentro da classe *MinhaListaOrdenavel* e vai devolver um objeto da classe *ArrayList*, com os objetos Produto ordenados segundo o critério da constante.

Exemplo:

```
1.Alfabetica (A-Z) – nome do produto
2.Alfabetica (Z-A) – nome do produto
3.Menor Preco - crescente
4.Maior Preco - decrescente
5.Mais Vendidos – decrescente pelo mais vendido
```

Para efetivar a ordenação você vai usar o método de classe *sort* da classe *Collections*. Observe que a classe *Collections* só tem métodos de classe (*static*) para ser aplicado a objetos de coleção – mais um exemplo de utilidade dos métodos de classe.

Exemplo:

```
public ArrayList ordena (int critério) {

...switch (critério) {
      case PRECO:
         Collections.sort(this.[ArrayList encapsulado] , precoC);
         // passamos o próprio ArrayList encapusado dentro de MinhaListaOrdenavel
         // e o Comparator correspondente ao critério
     case PRECO-REVERSO:
        Collections.sort(this.[ArrayList encapsulado] , precoC. reversed());
        // observe que a única diferença é a chamada a reversed()
...
   return this.[ArrayList encapsulado];
}
```

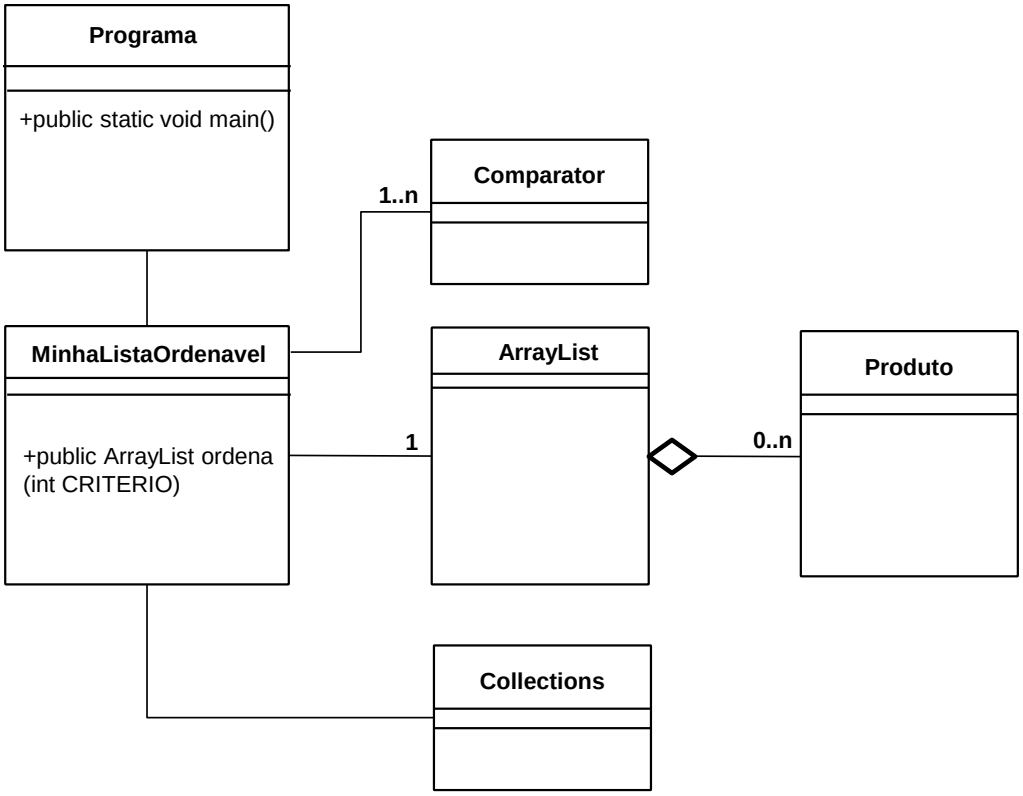

Figura 3. Diagrama de classe – Ex3

b) O programa principal vai conter um objeto da classe *MinhaListaOrdenavel* e partir dela algumas operações serão efetuadas.

- Crie um objeto da classe *MinhaListaOrdenavel*.

- Em seguida, crie "na mão" 10 objetos da classe *Produto* e insira no objeto *MinhaListaOrdenavel*.
- Na sequencia crie um menu para o usuário imprimir a lista de produtos ou sair do programa. Se o usuário optar por listar, pergunte qual o critério e liste os produtos.

PS.: Para simplificar as coisas, pode colocar estas etapas dentro do método *main*. No próximo exercício vamos organizar melhor as coisas.

Exemplo:

```
      1.Imprimir Lista
      2.Sair
Digite sua opcao: 2
            Escolha seu modo de ordenacao
      1.Alfabetica (A-Z)
      2.Alfabetica (Z-A)
      3.Menor Preco
      4.Maior Preco
      5.Mais Vendidos
Digite sua opcao: 5
2     Violao      750.0 6 21
8     Flauta      450.0 5 12 10% off!
3     Baixo 935.0 5 8 15% off!
5     Bateria     2500.0 3 4
13    Sax   4000.0 2 4

      1.Imprimir Lista
      2.Sair
Digite sua opcao: 2
            Escolha seu modo de ordenacao
      1.Alfabetica (A-Z)
      2.Alfabetica (Z-A)
      3.Menor Preco
      4.Maior Preco
      5.Mais Vendidos
Digite sua opcao: 3
8     Flauta      450.0 5 12 10% off!
2     Violao      750.0 6 21
3     Baixo 935.0 5 8 15% off!
5     Bateria     2500.0 3 4
13    Sax   4000.0 2 4
```

Exercício 4

a) Desenvolva uma classe de exceção confirmada chamada *NaoPrimoEx*. Ela vai ser usada pela classe *Primo*, adiante.

b) Desenvolva a classe *Numero*. Esta classe vai ter um campo para armazenar um número inteiro. Além disso, vai ter os seguintes métodos:

- o construtor, que recebe um inteiro para inicializar o campo que armazena o número inteiro;
- métodos *get* e *set;*
- métodos *soma*, *sub*, *mult* e *div*, que recebem um número inteiro como argumento de entrada e devolvem um **inteiro** como resultado da soma, subtração, multiplicação e divisão, respectivamente;
- O método toString.

c) Desenvolva a classe *Primo*, que encapsula um número primo. Esta classe herda da classe *Numero*.

Redefina o construtor e os métodos que executam as operações de soma, subtração, multiplicação e divisão usando *wrapping* das funções da classe pai. Porém, se o número resultante não for primo, a exceção *NaoPrimoEx* deve ser jogada.

Acrescente um método, que só a própria classe deve usar, *boolean isPrimo*(), que retorna *true* quando o número encapsulado por primo e *false* caso *contrário*.

Como sugestão de tratamento de exceção, você pode solicitar uma entrada de um outro número ou oferecer um arredondamento para o número primo mais próximo. O objeto que chamou o método decide (este objeto pode ser o cliente que interage com o usuário humano).

Obs.1: Considere 1, -1 e versões negativas dos primos como primos para o programa.
Obs.2: aceite números de ponto flutuante para as operações, mas inteiros no objeto.

Especialize também o método *toString*.

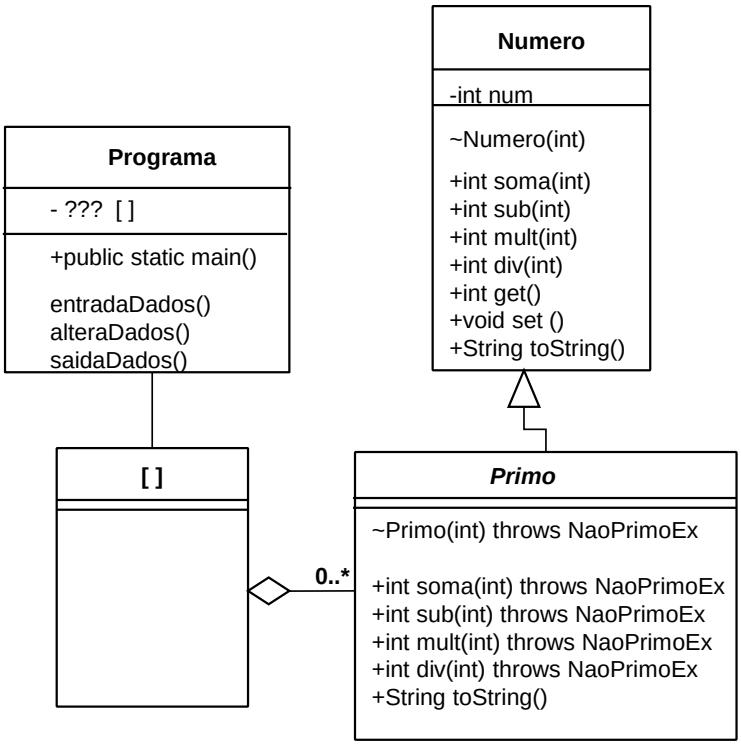

Figura 4. Diagrama de classe – Ex4

d) Agora desenvolva o programa principal em uma classe separada.

Crie um campo de instância, para um *array* de objetos da classe Numero e Primo.

Crie um construtor que inicialize o *array e* outro que não inicializa o *array*.

Crie um objeto da própria classe do programa
Chame os métodos deste objeto, listados a seguir, na sequência.

Crie 3 métodos: *entradaDados, alteraDados, saidaDados*.

No método *entradaDados* você deve entrar num loop e solicitar perguntar se o usuário quer entrar um número primo ou um número "normal". Criar o objeto correspondente e inserir no *array*. Se o usuário apenas pressionar o ENTER, a loop termina. O programa passa para a próxima fase.

No método *alteraDados* você vai criar um menu para testar seus métodos. Exemplo:

> `Qual número quer testar?` <o usuário deve digitar um número de índice válido do *array*, que ainda deve contar um objeto (ou seja, testar se é NULL)>
>
> ```
> 1. Soma
> 2. Subtração
> 3. Multiplicação
> 4. Divisão
> 5. Outro número
> 6. Sair
> ```

No método *saidaDados* você vai listar os objetos criados e modificados e terminar o programa.

Exercício 5 (bônus):

O IMC de uma pessoa pode ser calculado através de uma fórmula que consiste em dividir o peso (em kg) pelo quadrado da altura (em m²). O número resultante é então verificado em uma escala que varia de acordo com o seu gênero e então se chega a um resultado. Sua tarefa, neste exercício, é realizar o cálculo do IMC usando dados fornecidos pelo(a) usuário(a) e analisar a escala a fim de mostrar se ele(a) está acima, na média ou abaixo do peso ideal

Crie a classe *Pessoa* com os campos protegidos (encapsulados), *nome* e *dataNascimento*, objetos da classe *String*, que vão representar o nome e data de nascimento. A classe *Pessoa* deve conter:

- Um construtor que recebe como parâmetros duas *strings* e inicializa os campos *nome* e *dataNascimento*.
- O método *toString*, que não recebe parâmetros e retorna um objeto da classe String na seguinte forma:

```
Nome: <nome da pessoa>
Data de Nascimento: <data de nascimento da pessoa>
```

Crie a classe abstrata *PessoaIMC* que herde da classe *Pessoa* e contenha tenha os campos protegidos *peso* e *altura*, ambos do tipo *double*. O construtor desta classe deve receber como parâmetros duas *strings* e dois valores do tipo *double* e inicializar os campos *nome*, *dataNascimento*, *peso* e *altura*. A classe *PessoaIMC* deve conter os seguintes métodos:

- *public double getPeso()* que retorna o peso;
- *public double getAltura()* que retorna a altura;
- *calculaIMC()* que recebe como parâmetros dois valores do tipo *double* que são a altura e o peso e retorna um valor do tipo *double* correspondente ao IMC (Índice de Massa Corporal = peso / altura ao quadrado) calculado.
- o método abstrato *resultIMC()* que não recebe parâmetros e retorna uma instância da classe String. (o método não é implementado nesta classe - ele é **abstrato**)
- O método *toString()* desta classe deve retornar uma string da seguinte forma (um bom lugar para você exercer o reuso de código por herança):

```
Nome: <nome da pessoa>
Data de Nascimento: <sua data de nascimento>
Peso: <seu peso>
Altura: <sua altura>
```

Crie as classes *Homem* e *Mulher*, herdeiras de *PessoaIMC*. Cada uma deve implementar o método abstrato *resultIMC()* para realizar o calculo do IMC. O método *toString()* retorna um objeto da classe String com o resultado acordo com o valor obtido e todos as demais informações da pessoa.

Para Homem:	Para Mulher:
IMC < 20.7 : Abaixo do peso ideal	IMC < 19 : Abaixo do peso ideal
20.7 < IMC < 26.4: Peso ideal	19 < IMC < 25.8: Peso ideal
IMC > 26.4 : Acima do peso ideal	IMC > 25.8 : Acima do peso ideal

Crie uma classe para o programa principal, com o método *main()*, que crie instâncias das classes *Homem* e *Mulher* e armazene essas instâncias em um objeto da classe *Vector*. O programa deve perguntar ao usuário o número de pessoas, que tipo de objeto (Homem ou Mulher) deseja criar e os dados referentes a cada objeto. A leitura de dados deve ser feita através de fluxo de entrada. Após o armazenamento de todos os objetos, o programa deve ler cada elemento do *Vector*, imprimindo os dados do objeto ali contido, com todos os dados.

Erros de entrada de dados devem ser criticados e tratados quando possível (por exemplo, solicitando nova entrada).

Exemplo:

```
java Ex4
Digite o numero de pessoas:
2
Inserir homem (h) ou mulher(m)?
j
--- Opcao Invalida!!!
Inserir homem (h) ou mulher(m)?
h
Digite o nome:
Zezinho
Digite a data de nascimento:
01/01/1901
Digite o peso:
64.8
Digite a altura (em metros):
um m
--- A altura deve ser um numero real!!!
Digite a altura (em metros):
1.80
Inserir homem (h) ou mulher(m)?
m
Digite o nome:
Mariazinha
Digite a data de nascimento:
02/02/02/1902
Digite o peso:
64.8
Digite a altura (em metros):
1.8
--------
Nome: Zezinho
Data de Nascimento: 01/01/1901
Peso: 64.8
Altura: 1.8
IMC: 19.99  Abaixo do peso
--------
--------
Nome: Mariazinha
Data de Nascimento: 02/02/02/1902
Peso: 64.8
Altura: 1.8
IMC: 19.99  Peso ideal
--------
```

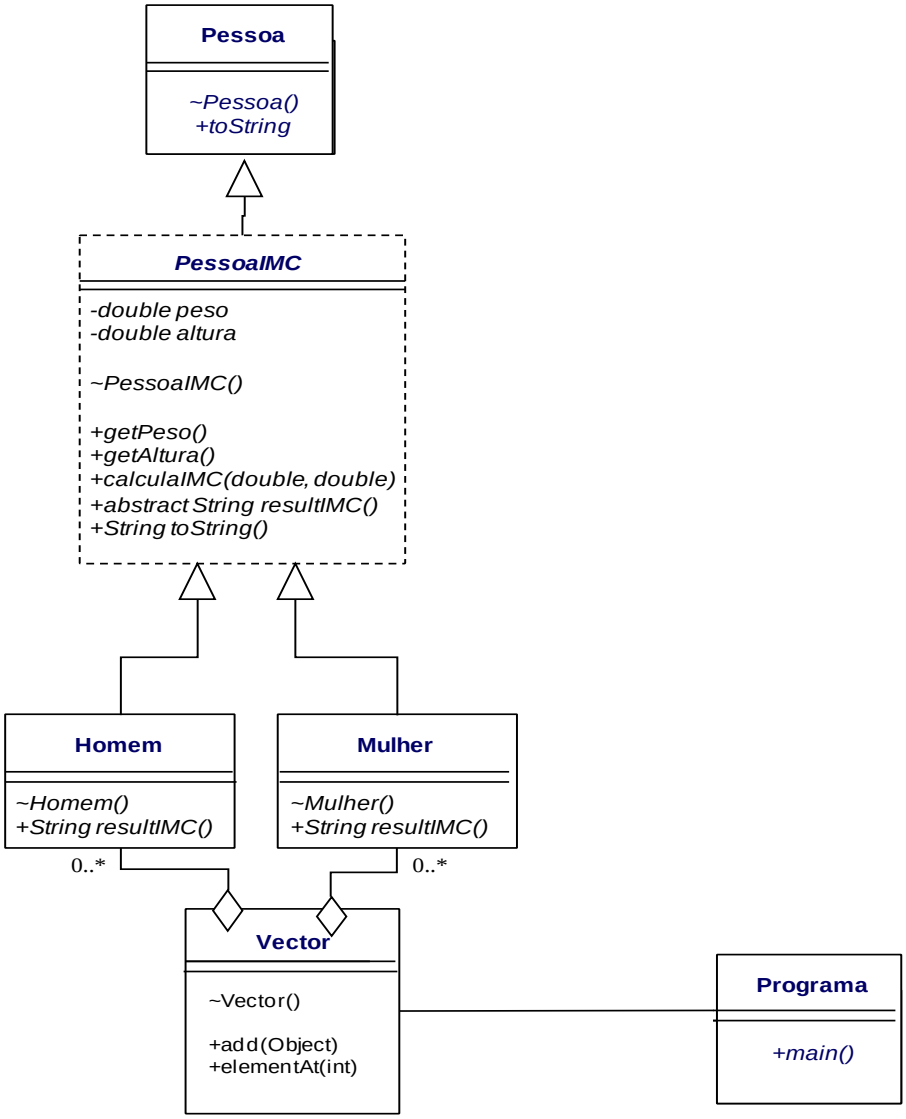

Figura4. Diagrama de classe - Ex5