Fundamentos da Programação Orientada a Objetos

UNIDADE 04

Interação entre Objetos

Olá! Nesta Unidade, vamos explorar a associação entre objetos, com o objetivo de verificar como o ***trabalho conjunto das classes*** *traz os benefícios mais significativos. Faremos isso explorando com maior detalhamento a interação entre os objetos, que é proporcionada pela* ***associação*** *entre classes.*

Associação entre classes e ligação entre objetos

No dia a dia, verificamos que existem vários objetos diferentes no mundo real que se associam e trabalham em conjunto. Veja os exemplos apresentados na Figura 1.

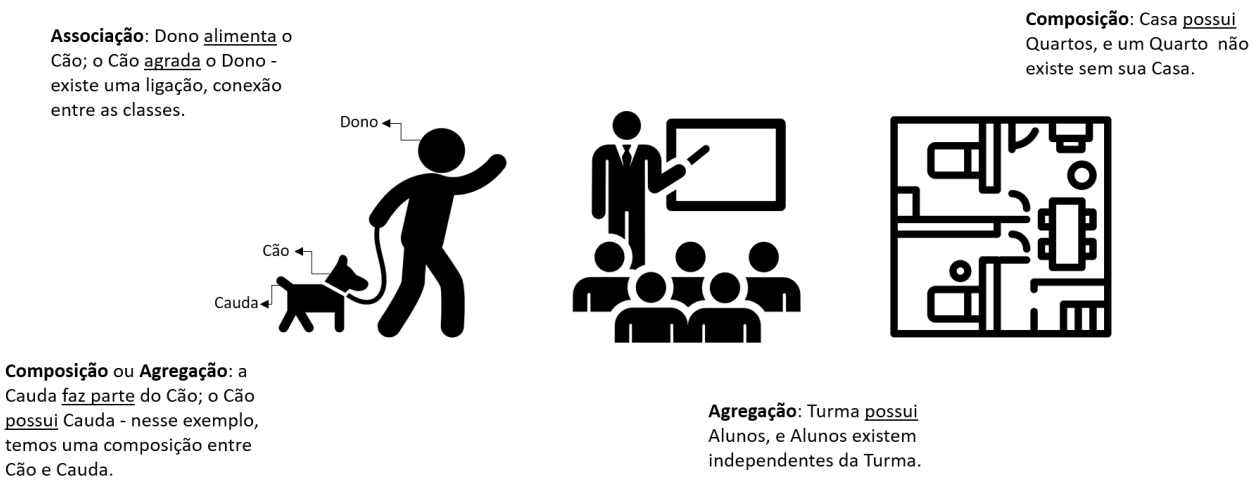
Considere as diferenças e similaridades entre os objetos da classe dono, cão e cauda. Conseguimos identificar as seguintes relações:

- **Donos** alimentam seus **cães** e **cães** agradam seus **donos** (**associação**).
- **Caudas** são parte de **cão**, ou o **cão** possui **cauda** (**agregação/composição**).

Explicando os conceitos de associação, agregação e composição:

- **Associação**: se duas classes precisam se comunicar, deve haver uma ligação entre elas, e isso pode ser representado por uma associação (conector).
- **Agregação** e **composição** são casos específicos da **associação**; em ambas, o objeto de uma classe "**possui**" objeto de outra classe, em um relacionamento **parte-todo**. Mas há uma diferença sutil.
  - **Agregação** é um relacionamento em que a parte pode existir independentemente do todo, como no exemplo: **turma** (todo) e **aluno** (parte da **turma**). Exclua a turma e os alunos ainda existirão.
  - **Composição** é um relacionamento em que a parte não existe sem o todo. Exemplo: **casa** (todo) e **quarto** (parte). Os quartos não existem sem sua casa.

Figura 1 – Tipos de relação entre objetos



Fonte: Autores (2020).

EXERCÍCIO

Associação

Vamos trabalhar nos exemplos passados, para ver como esses conceitos são traduzidas em código Java.

1. Crie um projeto com as classes **dono**, **cão** e **cauda**, no mesmo pacote default, como indicado.
2. Execute o projeto a partir da classe **dono**.
3. Observe que **dono** e **cão** têm uma relação de **associação**: a classe **dono** tem um **atributo** da classe **cão**: **pet**; a classe **dono cão** tem um **atributo** da classe **dono**: **meuDono**
4. Observe que **cão** têm uma relação de **composição** com a classe **cauda**: uma cauda não existe sem seu cão.
5. Altere a classe **dono** para ter mais um **cão**: **pet2**. Garanta que **pet2** seja alimentado e agrade sua dona **Maria**.
    a. Será necessário alterar apenas a classe **dono**: mais um atributo **pet2**.
    b. Duplique os métodos ***getter*** e ***setter***, para **pet2**.
    c. No método **alimentarCao**(), acrescente a linha de código para alimentar **pet2**.
    d. Altere também o método ***main*** da classe Dono, para criar mais uma instância de **cão.**

Figura 2 – Exercício para associação das classes **dono**, **cão** e **cauda**

```
</>
```

```
 1  public class Cao {             //
Associação: meuDono é uma referência da
 2      private Dono   meuDono;  // classe
Dono e também atributo da classe Cão
 3      private String nomeCao;
 4      private String raca;
 5      private String genero;
 6      private int    idade;
 7      private Cauda  minhaCauda; //
Composição: Cão possui Cauda
 8
 9      public Cao(String nome, String raca,
String genero, int idade,
10                     String forma, String
tipoPelo) {
11          this.nomeCao = nome;
12          this.raca = raca;
13          this.genero = genero;
14          this.idade = idade;     //
Composição: a cauda faz parte do cão
15          this.minhaCauda = new Cauda
(forma, tipoPelo);
16      }
17      public void setMeuDono(Dono meuDono) {
18          this.meuDono = meuDono;
19      }
20      public String getNomeCao() {
21          return nomeCao;
22      }
23      public void printCao() {
24          System.out.println("  Nome:   " +
this.nomeCao);
25          System.out.println("  Raça:   " +
this.raca);
26          System.out.println("  Gênero: " +
this.genero);
27          System.out.println("  Idade:  " +
this.idade);
28          minhaCauda.printCauda();
29          System.out.println();
30      }
31      public void realizarRefeicao() {
32          System.out.println(this.nomeCao +
" fazendo sua refeição.");
33      }
34      public void agradarDono() {
35          this.meuDono.receberFesta(); //
```

```
Invoca método da classe Dono
 36     }
 37 }
```

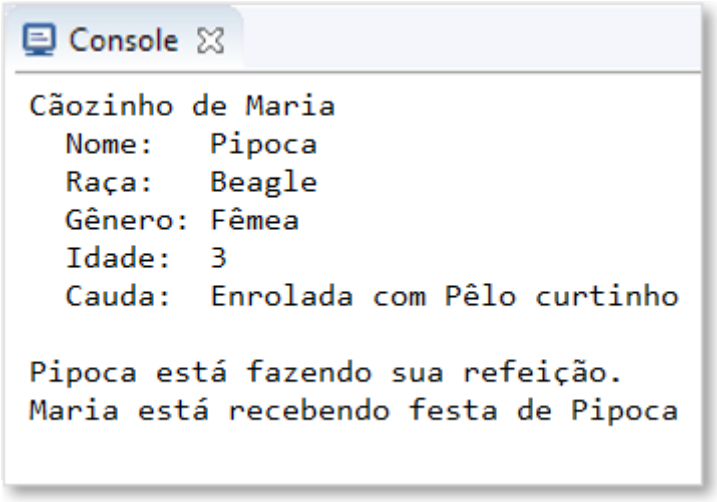
</>

```
  1 public class Cauda {
  2     private String forma;
  3     private String tipoPelo;
  4
  5     public Cauda(String forma, String
tipoPelo) {
  6         this.forma = forma;
  7         this.tipoPelo = tipoPelo;
  8     }
  9     public void printCauda() {
 10         System.out.println("  Cauda:  " +
this.forma + " com " +
 11
this.tipoPelo);
 12     }
 13 }
```

</>

```
 1  public class Dono {
 2      private String nome;
 3      private Cao    pet;  //
Dono está associado com seu Cão
 4
 5      public Dono(String nome)
{
 6          this.nome = nome;
 7      }
 8      public void setPet (Cao
pet) {
 9          this.pet = pet;
10      }
11      public Cao getPet () {
12          return this.pet;
13      }
14      public void
alimentarCao() {
15
pet.realizarRefeicao();
16      }
17      public void
receberFesta() {
18
System.out.println(this.nome + "
está recebendo festa de " +
19
this.pet.getNomeCao());
20      }
21
22      public static void
main(String[] args) {
23          Dono  maria  = new
Dono ("Maria");
24          Cao   pipoca = new
Cao  ("Pipoca", "Beagle",
"Fêmea", 3,
25
"Enrolada", "Pêlo curtinho");
26          maria.setPet
(pipoca); // associa Maria com
Pipoca
27          pipoca.setMeuDono
(maria);  // associa Pipoca com
Maria
28
29
System.out.println("Cãozinho de
" + maria.nome);
30
```

```
maria.getPet().printCao();
 31
 32
maria.alimentarCao();  // Maria
alimenta o seu cão Pipoca
 33
pipoca.agradarDono();  // Pipoca
agrada sua dona Maria
 34     }
 35  }
```



Fonte: Autores (2020).

Resolução do exercício

veja as alterações e acréscimos ao código da questão 4:

```
</>
```

```
  1     private String nome;
  2     private Cao    pet;    // Dono está
associado com seu Cão
  3     private Cao    pet2;  // Dono está
associado com seu Cão
  4  ...
  5  public void setPet (Cao pet) {  this.pet
= pet;  }
  6  public Cao getPet () { return this.pet; }
  7
  8  public void setPet2 (Cao pet) {
this.pet2 = pet;  }
  9  public Cao getPet2 () { return this.pet2;
}
 10  …
 11  public void alimentarCao() {
pet.realizarRefeicao();
pet2.realizarRefeicao();  }
 12  public void receberFesta() {
 13     System.out.println(this.nome + " está
recebendo festa de " + this.pet.getNomeCao());
 14     System.out.println(this.nome + " está
recebendo festa de " +
this.pet2.getNomeCao());
 15  }
 16  …
 17  Cao   bruce  = new Cao  ("Bruce", "Pug",
"Macho", 2, "Caracol", "Pêlo curtinho");
 18  ...
 19  bruce.setMeuDono    (maria);  // associa
Bruce com Maria
 20  ...
 21  maria.getPet2().printCao();
 22  ...
 23  bruce.agradarDono();   // Bruce agrada
sua dona Maria
```

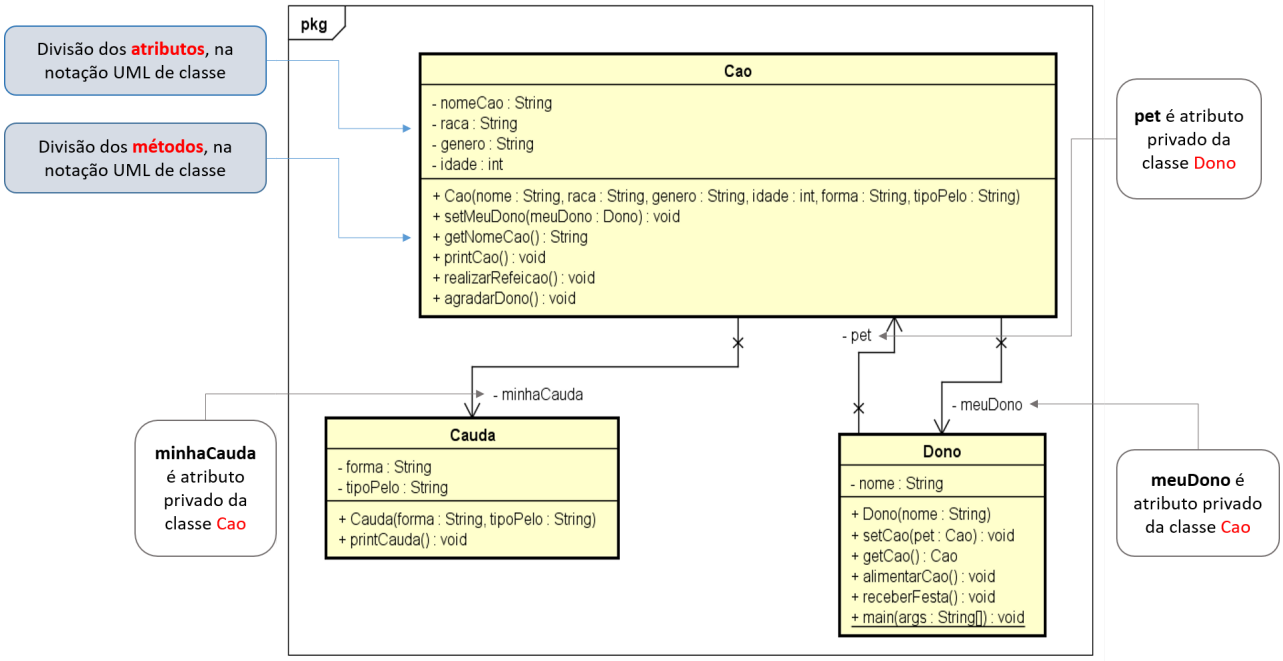
Diagrama de classes da associação

O código-exemplo da Figura 2 pode ser representado em um modelo específico, dado por seu diagrama de classes, conforme apresentado na Figura 3.

A especificação em *Unified Modeling Language*[1] (UML), das classes do Exercício 1, **dono**, **cão** e **cauda**, é apresentada a seguir:

a. Todos os atributos são privados (encapsulamento, com modificador -).
b. Todos os métodos são públicos (com modificador **+**, são a interface das classes).
c. Os **atributos de outras classes** aparecem nas **ligações** (conectores), também como privados (modificador -), na extremidade com flecha do conector.
d. O "x" no conector significa que não há visibilidade entre as classes no sentido oposto ao da flecha.
e. Este diagrama foi gerado pela ferramenta ASTAH[2], a partir do código Java do exercício 1.

Figura 3 – Diagrama das classes **dono**, **cão** e **cauda**



Fonte: Autores (2020).

[1]Linguagem Unificada de Modelagem, que modela aplicações orientada a objetos

[2]http://astah.net é uma ferramenta de diagramação e modelagem de aplicações

Observe que na Figura 3, o diagrama da UML coloca os atributos de outras classes nas extremidades da ligação (conector) entre as classes, e não dentro da notação de retângulo que representa a classe, em que há uma divisão apenas para os **atributos**. Essa é a forma que a UML usa para destacar a **associação** entre os objetos de classes, e é uma **boa prática de modelagem**.

Observe o detalhe a seguir de como o diagrama da Figura 3 é traduzido para Java, da Figura 2:

</>

```
1  public class Cao {             // Associação: meuDono é uma
referência da
2      private Dono   meuDono;  // classe Dono e atributo da
classe Cão
3      private String nomeCao;
4      private String raca;
5      private String genero;
6      private int    idade;
7      private Cauda  minhaCauda; // Associação do tipo
Composição: Cão
8                                    // possui Cauda
```

Utilização de objeto e referência de objeto

Recapitulando: um **objeto é uma instância de uma classe** e uma **classe pode ter vários objetos**, ou várias **instâncias**. Quando precisamos acessar esses objetos, utilizamos sua **referência**, que é o **nome da variável do objeto que instanciamos**, nos nossos códigos.

Para **instanciar uma classe**, ou **criar um objeto**, em Java, usamos a palavra-chave ***new***, como demostrado a seguir, em que o objeto da classe **dono** tem o **nome de variável**, ou **referência**, igual a `maria`:

```
1  Dono  maria  = new Dono ("Maria"); // maria é referência à
instância de Dono
```

> Em Java, um **objeto existe** apenas quando é **instanciado** com a palavra-chave ***new***.

Podemos e devemos utilizar os objetos e suas referências de diferentes formas. Vejam como fazemos isso, com base nos exemplos já passados nas Figura 2 e Figura 3.

Objeto utilizado como atributo

Começando na Figura 2, na classe **cao**, observamos que na Linha 2 temos o atributo `meuDono`, do tipo classe **dono**. Da mesma forma, na linha 7, temos o atributo `minhaCauda`, do tipo classe **cauda**.

O nome dos atributos `meuDono` e `minhaCauda` também são referências a esses objetos. Isso significa que um objeto da classe **cao** faz referência a objetos das classes **dono** e **cauda**:

```
 1  public class Cao {
 2        private Dono   meuDono;     // atributo = objeto da classe Dono
...       ...
 7        private Cauda  minhaCauda;  // atributo = objeto da classe Cauda
...       ...
```

Objeto utilizado como variável local

Para que um objeto de **cao** fique completo, precisamos instanciar **dono** e **cauda**. Isso é feito na classe **dono**, nas linhas 22 a 27, da classe **dono** na Figura 2:

```
22  public static void main(String[] args) {
23        Dono  maria  = new Dono ("Maria");
24        Cao   pipoca = new Cao  ("Pipoca", "Beagle", "Fêmea", 3,
25                                           "Enrolada", "Pêlo curtinho");
26        maria.setPet        (pipoca); // associa Maria com Pipoca
27        pipoca.setMeuDono   (maria);  // associa Pipoca com Maria
...       ...
34  }
```

Observe que as variáveis `maria` e `pipoca`, que são **referências a objetos** das classes **dono** e **cao**, respectivamente, também são **variáveis locais**, pois existem apenas dentro do método *`main`*, que vai das linhas 22 a 34 na classe **dono**.

> **Variáveis locais** apenas são **acessíveis**, **ou visíveis, dentro do escopo** em que foram criados, como dentro dos limitadores { e } de um método.

Objeto utilizado como parâmetro (argumento)

Novamente, na Figura 2, classe **dono**, vemos o método `Dono.setPet` que espera receber um objeto da classe **cao** como **argumento ou parâmetro**:

```
</>

  1  public void setPet (Cao pet) {  // argumento = pet
  2          this.pet = pet;    // atributo recebe o argumento
  3      }
```

Da mesma forma, na Figura 2, classe **cao**, o método `Cao.setMeuDono` espera receber um objeto da classe **dono** como **argumento ou parâmetro**:

```
</>

  1  public void setMeuDono(Dono meuDono) { // argumento =
meuDono
  2          this.meuDono = meuDono;    // atributo recebe o
argumento
  3      }
```

Esses métodos `Dono.setPet` e `Cao.setMeuDono` apenas podem ser invocados se existirem os objetos que serão usados como parâmetros. Vejam suas **invocações** no exemplo da classe **dono** na Figura 2, nas linhas 26 e 27, em que os objetos de classes **dono** e **cao** são usados como **parâmetros** (ou **argumentos**) de método:

```
22  public static void main(String[] args) {
23        Dono  maria  = new Dono ("Maria");
24        Cao   pipoca = new Cao  ("Pipoca", "Beagle", "Fêmea", 3,
25                                 "Enrolada", "Pêlo curtinho");
26        maria.setPet       (pipoca); // associa Maria com Pipoca
27        pipoca.setMeuDono  (maria);  // associa Pipoca com Maria
...       ...
34  }
```

Observação: primeiramente, instanciamos `maria` e `pipoca`, para depois utilizá-los como parâmetros.

Objeto utilizado como retorno de método

Mais um exemplo da Figura 2, classe **cao:** vemos o método `Cao.sgetNomeCao` **retorna** o **valor do atributo** nomeCao, que é do tipo *`string`* – recordando que *`string`* no Java é uma **classe**, logo, nomeCao faz referência a um objeto da classe *`string`*.

</>

```
1  public String getNomeCao() {
2      return nomeCao;
```

> **Recapitulando**: um método tem a sua **assinatura** definida como (mais detalhes na Unidade 2).
>
> ```
> tipo nomeMetodo (tipo parametro1, tipo parametro2,
> ..., tipo parametroN) {
>     // corpo do método
> }
> ```

Antes no nome do método, temos o tipo de dado que o método retorna, que neste exemplo é uma **instância** (objeto) da classe *`string`*. Logo, o método tem que ter obrigatoriamente a palavra-chave ***`return`***, seguido do valor que será retornado pela invocação (chamada do método).

Coleção de objetos: vetores e listas

Em algumas soluções computacionais, precisamos que **um objeto mantenha várias instâncias de outros objetos**. Fizemos isso no Exercício 1 - Associação, no qual solicitamos a criação de outro *pet* para Maria, de forma que ele tivesse dois cães, ou **dois objetos** da classe **cao:**

</>

```
1  public class Dono {
2      private String nome;
3      private Cao    pet;  // Dono está associado com seu Cão
1
4  private Cao    pet2; // Dono está associado com seu Cão 2
5  ...
6  }
```

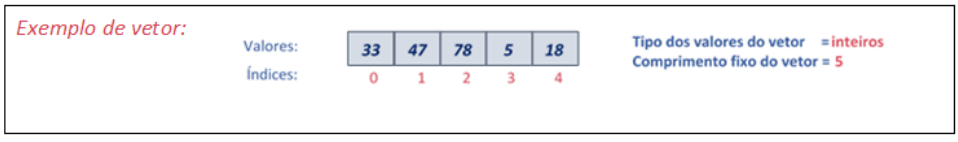
Contudo, essa **não é a melhor forma de manter vários objetos da mesma classe**! Uma solução mais **flexível** deve ser usada para resolver os casos em que precisamos de uma **coleção de objetos**. O Java oferece algumas alternativas para criar essas coleções, como veremos a seguir com as soluções dadas por **vetores** e **lista de objetos**.

EXERCÍCIO

Vetores

Em Java, um **vetor**, ou ***array***, é um **objeto** que mantém um **número fixo** de valores de um único tipo de dado. Portanto, comprimento do vetor (***array length***) é **fixo**, pois é definido no momento de sua criação, ou instanciação (precisamos usar a palavra-chave ***new***) e não pode ser alterado.

Para acessar cada elemento do vetor precisamos usar um **índice**, como apresentado na Figura 4 a seguir.

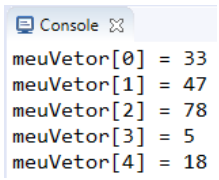


Código Java correspondente:

Figura 4 – Exercício com **vetor** (*array*)

```
</>
```

```
 1  int[] meuVetor = new int
[5];
 2
 3      meuVetor[0] = 33;
 4      meuVetor[1] = 47;
 5      meuVetor[2] = 78;
 6      meuVetor[3] = 5;
 7      meuVetor[4] = 18;
 8
 9      for(int i=0; i<
meuVetor.length; i++)
10
System.out.println("meuVetor["
+ i + "] = " + meuVetor[i]);
11
12  // meuVetor.length =
comprimento do vetor = 5
inteiros mantidos no vetor
```



Fonte: Autores (2020).

Na Figura 5 a seguir, apresentamos a alteração necessária para que o exemplo da Figura 2 possa manter um vetor de objetos da classe **cao**.

EXERCÍCIO

1. Mantendo os arquivos das classes **cao** e **cauda**, da Figura 2, crie uma classe **dono** conforme o indicado.

2. Observe que **dono** agora tem um atributo do tipo vetor de objetos da classe **cao**, atributo chamado de ***pets***.

3. Observe que para manipular o vetor **pets** e acessar todos os seus objetos, usamos uma variável inteira **index** que aponta o objeto que queremos de ***pets***, como no exemplo:
this.pets[index] = pet;

4. Execute o projeto a partir da classe **dono**.

5. Responda:

a. O que acontece quando alteramos linha 9 e acrescentamos a linha 52?

```
linha 9    pets = new Cao[4];
linha 52   maria.getPet(3).printCao();
```

b. O que acontece quando alteramos na linha 38 o valor do índice que **pipoca** deve ter no vetor **pets**, para 20:

```
maria.addPet        (20, pipoca); ?
```

Figura 5 – Exercício com **vetor** de objetos da classe **cão**

</>

```
 1  public class Dono {
 2      private String nome;
 3      private Cao[]  pets;  // Dono está
associado a um vetor pets de cães
 4
 5      public Dono(String nome) {
 6          this.nome = nome;   // Cria uma
instância do vetor;
 7          pets = new Cao[3];  // vetor
preparado para receber até 3 cães
 8      }
 9      public void addPet (int index, Cao
pet) { // inclui um objeto de cão
10          this.pets[index] = pet;
// no vetor pets
11      }
12      public Cao getPet (int index) { //
obtém uma instância de Cao
13          return this.pets[index];  // na
posição index do vetor pets
14      }
15      public void alimentarCaes() {
16          for(int i=0; i< 3; i++)
// Invoca realizarRefeicao
17              pets[i].realizarRefeicao();
// de cada objeto de pets
18      }
19      public void receberFesta() {        //
Invoca o método receberFesta
20          for(int i=0; i< 3; i++)       //
de cada objeto de pets
21              System.out.println(this.nome +
" está recebendo festa de "
22                                          +
this.pets[i].getNomeCao());
23      }
24      public void listarCaes() {      //
lista todos os cães do vetor pets
25          for(int i=0; i< 3; i++)
26              this.getPet(i).printCao();
27      }
28
29      public static void main(String[] args)
{
30          Dono  maria  = new Dono ("Maria");
31          Cao   pipoca = new Cao  ("Pipoca",
"Beagle", "Fêmea", 3,
32
"Enrolada", "Pêlo curtinho");
```

```
 33          Cao   bruce  = new Cao  ("Bruce",
"Pug", "Macho", 2, "Caracol",
 34
"Pêlo curtinho");
 35          Cao   jujuba = new Cao  ("Jujuba",
"Maltês", "Fêmea", 1,
 36
"Enrolada", "Pêlo longo");
 37
 38          maria.addPet   (0, pipoca); //
Pipoca está na posição 0 de pets
 39          maria.addPet   (1, bruce);  //
Bruce  está na posição 1 de pets
 40          maria.addPet   (2, jujuba); //
jujuba está na posição 2 de pets
 41
 42          pipoca.setMeuDono (maria);     //
associa Pipoca com Maria
 43          bruce.setMeuDono  (maria);     //
associa Bruce com Maria
 44          jujuba.setMeuDono (maria);     //
associa Jujuba com Maria
 45
 46          //Lista todos os cães de Maria
 47          System.out.println("Cãozinhos de "
+ maria.nome);
 48          maria.listarCaes();
 49
 50          maria.alimentarCaes();  // Maria
alimenta todos os  cães
 51          maria.receberFesta();   // Maria
recebe festa de todos os cães
 52      }
 53  }
 54
```

Fonte: Autores (2020).

Resolução do exercício

Veja as alterações e acréscimos ao código:

5. Teste de alterações:

a. Índice do vetor é válido, contudo, não foi incluído um objeto da classe cao na posição do índice; **exceção**, ou erro, de tipo “o valor de retorno na posição é nulo, ou a posição do vetor é válida, mas

está vazia”: `Exception in thread "main" java.lang.NullPointerException: Cannot invoke "Vetor.Cao.printCao()" because the return value of "Vetor.Dono.getPet(int)" is null`

b. Índice = 20 está fora dos limites do vetor pets, que vai de 0 a 2 (3 objetos de cao) ; **exceção**, ou erro, do tipo “índice fora dos limites do vetor”:

4.

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException:

Index 20 out of bounds for length 3
```

EXERCÍCIO

Listas de objetos com *ArrayList*

A classe ***ArrayList*** provê um objeto do tipo **vetor**, porém com **comprimento variável**: diferentemente do **vetor** (*array*), o ***ArrayList* não tem um comprimento fixo**!

O ***ArrayList***, que está na biblioteca `java.util`, precisa que seja declarada a classe dos objetos que serão mantidos na sua estrutura. Veja o exemplo da Figura 6.

Código Java correspondente:

Figura 6 – Exercício com *ArrayList* de objetos da classe *string*

</>

```
 1  import java.util.ArrayList;
 2
 3  public class ListaCores {
 4      public static void
main(String[] args) {
 5          int i;
 6          // Declara a
instancia o ArrayList cores
 7          ArrayList<String>
cores = new ArrayList<String>();
 8          cores.add("Azul");
// Inclui elemento no ArrayList
 9          cores.add("Verde");
// Inclui elemento no ArrayList
10
cores.add("Vermelho"); // Inclui
elemento no ArrayList
11          cores.add("Amarelo");
// Inclui elemento no ArrayList
12
13          // Loop para varrer a
lista, elemento por elemento
14          for (i = 0; i <
cores.size(); i++) // imprime
cada elemento
15
System.out.println((i+1) + "º) "
+ cores.get(i));
16
17          // ALTERA elemento da
lista:
18          cores.set(1, "Pink");
// altera elemento na posição 1
para "Pink"
19
20          i=0;
21          System.out.println("-
---");
22          // Loop for-each para
varrer a lista, elemento por
elemento
23          for (String c :
cores) { // imprime cada
elemento
24
System.out.println((i+1) + "º) "
+ c);
25             i++;
26          }
27
28          // REMOVE elemento da
```

```
Console
1º) Azul
2º) Verde
3º) Vermelho
4º) Amarelo
----
1º) Azul
2º) Pink
3º) Vermelho
4º) Amarelo
----
1º) Azul
2º) Pink
3º) Vermelho
----
Tamanho da lista = 0
```

```
lista da posição 3: “Vermelho”
 29        cores.remove(3);
 30
 31        i=0;
 32        System.out.println("-
---");
 33        // Loop for-each para
varrer a lista, elemento por
elemento
 34        for (String c :
cores) { // imprime elemento por
elemento
 35
System.out.println((i+1) + "º) "
+ c);
 36           i++;
 37        }
 38
 39        // LIMPA a lista:
exclui todos os objetos de
String
 40        cores.clear();
 41
 42        System.out.println("-
---");
 43
System.out.println("Tamanho da
lista = " + cores.size());
 44     }
 45  }
```

Fonte: Autores (2020).

A Figura 7 apresenta outros métodos da classe ***ArrayList***, muito úteis e que tornam esse recurso poderoso, provendo muita flexibilidade para a manipulação de uma **coleção de objetos**.

Figura 7 – Métodos do ***ArrayList***

```
1.  add(Object obj)              // insere um objeto no fim da lista
2.  add(int index, Object obj)   // insere um objeto na posição especificada
3.  remove(Object obj)           // remove da lista o objeto especificado
4.  remove(int index)            // remove da lista o objeto na posição especificada
5.  set(int index, Object obj)   // atualiza o objeto na posição especificada
6.  int indexOf(Object obj)      // retorna à posição do objeto especificado
7.  Object get(int index)        // retorna o objeto na posição especificada
8.  int size( )                  // retorna o tamanho da lista
9.  boolean contains(Object obj) // verifica se o objeto passado está na lista
10. clear( )                     // remove todos os objetos da lista
```

Referência para consulta:https://www.w3schools.com/java/java_arraylist.asp.Fonte: Autores (2020).

Na Figura 8 a seguir, será possível realizar um exercício mais completo, em que criamos uma classe **dependente**, cujos objetos são mantidos pela classe **contribuinte** em um atributo de ***ArrayList***. Vamos praticar!

EXERCÍCIO

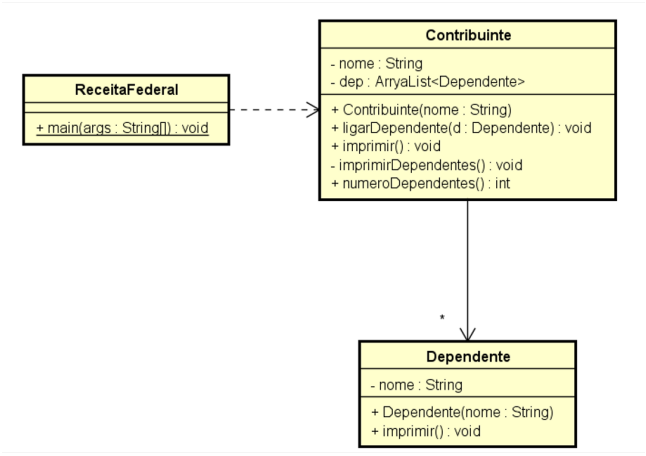


Diagrama de classes

A classe **ReceitaFederal** depende da classe contribuinte, pois utiliza seus objetos.

A classe **contribuinte** possui (agregação) objetos da classe **dependentes**, mantidos em um **ArrayList**.

O método privado **imprimirDpendente()** da classe contribuinte apenas é visível dentro da própria classe. Já o método **imprimir()** é **público**, logo visível na classe **ReceitaFederal**.

1. Crie os arquivos com os códigos fonte das classes dependente, contribuinte e ReceitaFederal, conforme o indicado.
2. Observe que **contribuinte** tem um atributo do tipo ***ArrayList*** de objetos da classe **dependentes**, atributo chamado de **dep**: `private ArrayList <Dependente> dep;`
3. Observe que a instância **dep** é criada no método **construtor** da classe **contribuinte**: `dep = new ArrayList <Dependente>();`
4. Observe que para manipular **pets** e acessar cada um dos seus objetos, usamos um iterador fo tipo *for-each*:

```
for (Dependente d : dep) {
        d.imprimir();
}
```

5. Execute o projeto a partir da classe **ReceitaFederal**.
6. Altere a classe **ReceitaFederal**:

    a. Acrescente um novo **contribuinte** Pedro, sem dependente. Imprima o total de dependentes de Pedro

(zero dependentes).

b. Acrescente um novo **contribuinte** João, com uma única dependente Ana. Imprima o total de dependentes de Pedro (uma dependente).

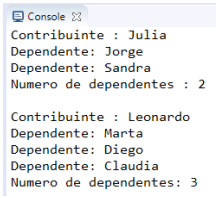
Figura 8 – Exercício com *ArrayList* de objetos da classe dependente

```
</>
```

```
 1  public class Dependente {
 2      private String nome;
 3
 4      public Dependente(String nome) {
 5          this.nome = nome;
 6      }
 7      public void imprimir() {
 8          System.out.println ("Dependente: "
+ nome);
 9      }
10  }
```

```
</>
```

```
1  import java.util.ArrayList;
2
3  public class Contribuinte {
4      private String nome;
5      private ArrayList
<Dependente> dep;
6      public Contribuinte
(String nome) {
7          this.nome = nome;
8          dep = new ArrayList
<Dependente>();
9      }
10     public void
ligarDependente (Dependente d) {
11         dep.add (d);
12     }
13     public void imprimir ()
{
14
System.out.println("Contribuinte
: " + this.nome);
15
imprimirDependentes();
16     }
17     private void
imprimirDependentes () {
18         for (Dependente d :
dep) {
19             d.imprimir();
20         }
21     }
22     public int
numeroDependentes () {
23         return dep.size();
24     }
25 }
```



</>

```
 1  public class ReceitaFederal {
 2
 3      public static void main(String[] args)
{
 4          Contribuinte julia = new
Contribuinte ( "Julia");
 5          Dependente jorge = new Dependente
( "Jorge");
 6          Dependente sandra = new Dependente
( "Sandra");
 7
 8          julia.ligarDependente(jorge);
 9          julia.ligarDependente(sandra);
10          julia.imprimir();
11          System.out.println("Numero de
dependentes : " +
12
julia.numeroDependentes ( ) + "\n");
13
14          Contribuinte leonardo = new
Contribuinte ("Leonardo");
15          Dependente marta = new Dependente
("Marta");
16          Dependente diego = new Dependente
("Diego");
17          Dependente claudia = new
Dependente ("Claudia");
18          leonardo.ligarDependente(marta);
19          leonardo.ligarDependente(diego);
20          leonardo.ligarDependente(claudia);
21          leonardo.imprimir();
22          System.out.println ("Numero de
dependentes: " +
23
leonardo.numeroDependentes ( ) + "\n");
24      }
25  }
```

Fonte: Autores (2020).

Resolução do exercício

veja as alterações e acréscimos ao código.

6. Acrescentando mais contribuintes

a. Pedro, sem dependentes:

```
1  Contribuinte pedro = new Contribuinte
("Pedro");
2  pedro.imprimir();
3  System.out.println ("Numero de
dependentes: " + pedro.numeroDependentes ( ) +
"\n");
```

b. João, com uma dependente:

```
1  Contribuinte joao = new Contribuinte
("João");
2  Dependente ana = new Dependente ("Ana");
3  joao.ligarDependente(ana);
4  joao.imprimir();
5  System.out.println ("Numero de
dependentes: " + joao.numeroDependentes ( ) +
"\n");
```

EXPERIMENTE

Referências

GODOY, V. **Programação orientada a objetos I**. Curitiba: IESDE, 2019.

HORSTMANN, C. S.; CORNELL, G. **Core Java – volume I**. 8. ed. São Paulo: Pearson, 2010.

SCHILDT, H. **Java para iniciantes**. Porto Alegre: Bookman, 2015.