



UNIVERSIDADE FEDERAL RURAL DO SEMI-ÁRIDO
CENTRO MULTIDISCIPLINAR DE PAU DOS FERROS
DEPARTAMENTO DE CIÊNCIAS EXATAS E NATURAIS
DISCIPLINA: **PROJETO DETALHADO DE SOFTWARE**
PROFESSOR: **HULIANE MEDEIROS DA SILVA**
DISCENTES: **ARTHUR ALVES DE ABREU**
BRUNO VICTOR PAIVA DA SILVA
VINICIUS ANACLETO D ALMEIDA

Link: <https://github.com/ArtthurAbreu/Gerenciamento-Zoologico>

RELATÓRIO DE SISTEMA GERENCIAMENTO DE ZOOLOGICO

1. DESCRIÇÃO DO SISTEMA

O nosso sistema consiste em um sistema para gerenciar um zoológico, onde teremos que gerenciar os animais e os funcionários que estão presentes nesse zoológico. Com isso, na parte de gerenciamento dos animais é onde vamos controlar todos os animais que vão ser adicionados, separando-os por espécies pertencentes de cada um, pois cada animal tem características específicas que os diferencia um do outro. Além disso, apresentamos particularidades importantes sobre as espécies de cada animal, temos nas informações sobre: as suas principais características, a sua reprodução, qual o meio de locomoção de cada espécie, a sua respiração e o seu habitat. Podemos ainda, dentro do nosso sistema verificar se algum animal está doente ou não, fazendo assim com que saibamos quando esse animal precisa ou não de visita do veterinário. Além de tudo, o nosso sistema consiste também na parte de gerenciamento dos funcionários, onde controlamos o total de funcionários, os salários de cada funcionário e a função que cada um deve cumprir, tendo como funcionários: o Veterinário que tem como função cuidar dos animais, Gerente e o zelador. É importante ressaltar, que além de gerenciarmos a parte do salário, as funções dos funcionários, os animais que foram adicionados e qual a espécie de cada, também gerenciamos o local que cada funcionário irá ocupar e a jaula dos animais, onde cada animal terá sua jaula não podendo ser ocupada por mais de um animal em cada.

Portanto, nosso sistema teria como objetivo administrar um zoológico em todas as suas áreas e que a partir disso buscamos facilitar o gerenciamento de um dono de um zoológico, onde ele teria todo o controle por apenas um sistema, o que facilitaria tanto na parte financeira para os funcionários quando na parte de controle dos animais que estão ou que saíram do zoológico.

2. ESPECIFICAÇÃO DO SISTEMA

A implementação realizada neste projeto tem como objetivo principal o gerenciamento de todos os componentes constituintes de um zoológico, desde equipe dos funcionários até o alocamento de animais, o programa é constituído por quatorze classes e uma interface.

A interface é constituída pelos métodos abstratos “remove”, “imprimir” e “imprimirTodos”, e possui como descendente as classes “GerenciadorFuncionarios” e “GerenciadorAnimal”, as mesmas nos permite implementar funcionalidades como adicionar um funcionário ou animal no sistema, removê-los, buscá-los e até mesmo exibir suas informações na tela, em outras palavras, são responsáveis por administrar os ArrayList constituídos por funcionários e animais.

As classes abstratas contém o código que é comum a todas suas classes filhas. A classe “Animal” através das suas descendentes e dos seus construtores herdados, possibilita ao usuário declarar características como nome, peso, idade, quantidade de membros, alocar o animal em uma jaula específica, tendo seu setor já pré-definido dependendo da sua

classificação e por fim podemos declarar um ID único para cada animal adicionado. Esta classe também proporciona informar a ausência de saúde de um animal, relacionando o mesmo com o veterinário apropriado para realizar a consulta.

“Funcionario” permite por meio das classes filhas e dos construtores herdados a determinação do nome do funcionário, o salário base, a sala que pertence ao profissional e uma matrícula específica.

O método “InfoAnimais” permite ao funcionário ou até mesmo clientes do zoológico obter uma mensagem com informações gerais características de cada classe dos animais. O método “Analisedagestacao” tem como objetivo informar ao utente se baseado no tempo de gestação de cada animal, o filhote já está próximo de nascer, ou se está no tempo necessário para realizar a ultrassom permitindo averiguar a saúde do bebê.

Nas classes filhas de “Funcionario”, existe a possibilidade de calcular o salário de cada funcionário através do método “calculaSalario”, exibindo uma mensagem na tela com o valor exato a ser recebido.

O sistema foi desenvolvido utilizando a linguagem de programação Java, umas das linguagens mais utilizadas nos dias atuais voltado à programação orientada a objeto. Em java faz utilização de classes e objetos. Contudo, dentro das classes poderá possuir atributos e métodos, e nelas são possíveis as criações de vários objetos, no qual poderão utilizar os atributos e métodos gerenciados por uma determinada classe. Além disso, Java é composto por bibliotecas, e nesse sistema foi necessário o uso de algumas, como ArrayList e a Interface Gráfica. Sendo, o ArrayList utilizado para implementação de listas a partir de uma Interface List, facilitando na manipulação dos conjuntos de dados; e a biblioteca de interface Gráfica torna-se responsável pela interação com o usuário , utilizando dispositivos que servem para a entrada de dados, como mouse e teclado.

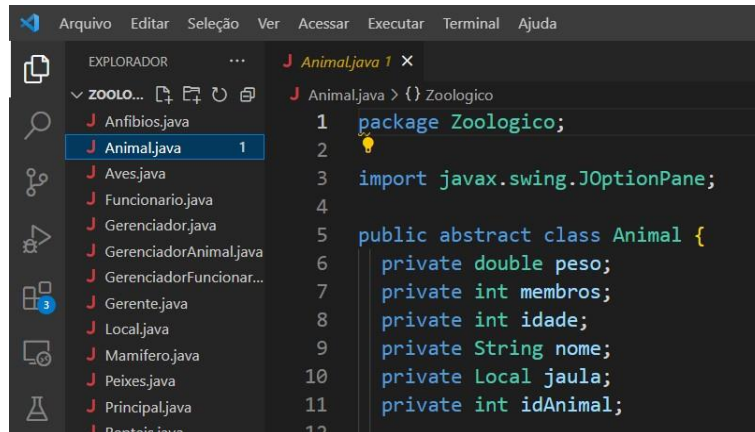
Futuramente, dando uma continuidade ao sistema de gerenciamento do zoológico, pode ser adicionado um esquema de senhas, para a privacidade do profissional, aumentar as classes de funcionários, implementar métodos específicos para cada classificação de animais, e incluir um sistema de controle de clientes.

3. CONTEÚDOS USADOS NO PROJETO

3.1. CLASSE

Uma classe serve para definir tanto estrutura como comportamentos do que deseja representar, é uma forma de definir um tipo de dado e utilizamos classes para representar objetos. Além disso, é dentro dela onde atribuímos atributos e métodos onde irá representar as características desse objeto representado. Nesse projeto utilizamos as classes para representar algumas espécies de animais existentes, foram elas: Anfíbios, Aves, Mamífero , Peixes e Répteis, onde nelas declaramos alguns atributos e métodos que vão representar suas características, e também temos a classe “GerenciadorAnimal” para podermos manipular os

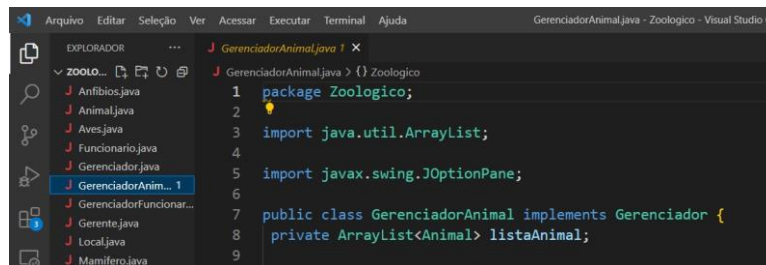
animais, ou seja, criamos uma lista de animais para podermos adicionar, remover e entre outros. Exemplos no sistema:



```
Arquivo  Editar  Seleção  Ver  Acessar  Executar  Terminal  Ajuda
EXPLORADOR  ...  J Animal.java 1 X
ZOOL...  J Anfibios.java
J Animal.java 1
J Aves.java
J Funcionario.java
J Gerenciador.java
J GerenciadorAnimal.java
J GerenciadorFuncionar...
J Gerente.java
J Local.java
J Mamifero.java
J Peixes.java
J Principal.java
J Renteis.java

J Animal.java > {} Zoologico
1 package Zoologico;
2
3 import javax.swing.JOptionPane;
4
5 public abstract class Animal {
6     private double peso;
7     private int membros;
8     private int idade;
9     private String nome;
10    private Local jaula;
11    private int idAnimal;
12
```

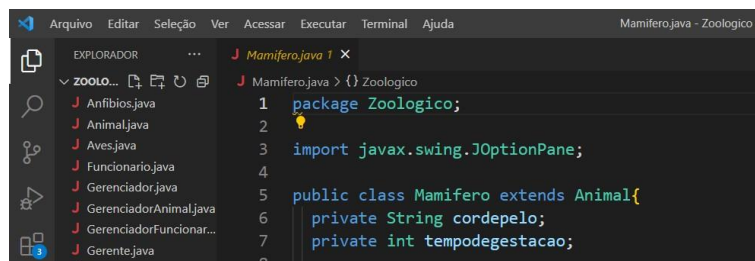
Figura 1. Exemplo de uma classe chamada “Animal”.



```
Arquivo  Editar  Seleção  Ver  Acessar  Executar  Terminal  Ajuda
EXPLORADOR  ...  J GerenciadorAnimal.java 1 X
ZOOL...  J Anfibios.java
J Animal.java
J Aves.java
J Funcionario.java
J Gerenciador.java
J GerenciadorAnimal.java 1
J GerenciadorFuncionar...
J Gerente.java
J Local.java
J Mamifero.java
J Peixes.java
J Principal.java
J Renteis.java

J GerenciadorAnimal.java > {} Zoologico
1 package Zoologico;
2
3 import java.util.ArrayList;
4
5 import javax.swing.JOptionPane;
6
7 public class GerenciadorAnimal implements Gerenciador {
8     private ArrayList<Animal> listaAnimal;
9
```

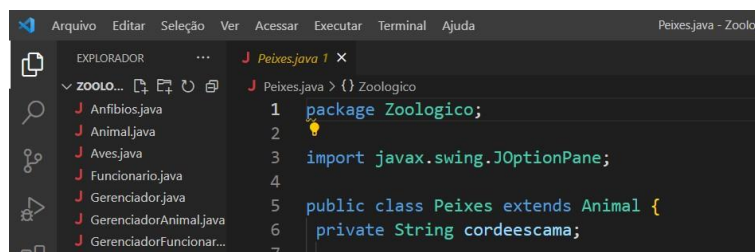
Figura 2. Exemplo de uma classe chamada “GerenciadorAnimal”.



```
Arquivo  Editar  Seleção  Ver  Acessar  Executar  Terminal  Ajuda
EXPLORADOR  ...  J Mamifero.java 1 X
ZOOL...  J Anfibios.java
J Animal.java
J Aves.java
J Funcionario.java
J Gerenciador.java
J GerenciadorAnimal.java
J GerenciadorFuncionar...
J Gerente.java
J Local.java
J Mamifero.java 1
J Peixes.java
J Principal.java
J Renteis.java

J Mamifero.java > {} Zoologico
1 package Zoologico;
2
3 import javax.swing.JOptionPane;
4
5 public class Mamifero extends Animal{
6     private String cordepelo;
7     private int tempodegestacao;
8
```

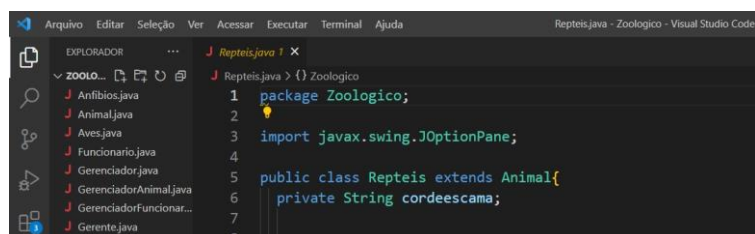
Figura 3. Exemplo de uma classe chamada Mamífero.



```
Arquivo  Editar  Seleção  Ver  Acessar  Executar  Terminal  Ajuda
EXPLORADOR  ...  J Peixes.java 1 X
ZOOL...  J Anfibios.java
J Animal.java
J Aves.java
J Funcionario.java
J Gerenciador.java
J GerenciadorAnimal.java
J GerenciadorFuncionar...
J Gerente.java
J Local.java
J Mamifero.java
J Peixes.java 1
J Principal.java
J Renteis.java

J Peixes.java > {} Zoologico
1 package Zoologico;
2
3 import javax.swing.JOptionPane;
4
5 public class Peixes extends Animal {
6     private String cordeescama;
7
```

Figura 4. Exemplo de uma classe chamada Peixes.



```
Arquivo  Editar  Seleção  Ver  Acessar  Executar  Terminal  Ajuda
EXPLORADOR  ...  J Repteis.java 1 X
ZOOL...  J Anfibios.java
J Animal.java
J Aves.java
J Funcionario.java
J Gerenciador.java
J GerenciadorAnimal.java
J GerenciadorFuncionar...
J Gerente.java
J Local.java
J Mamifero.java
J Peixes.java
J Principal.java
J Repteis.java 1

J Repteis.java > {} Zoologico
1 package Zoologico;
2
3 import javax.swing.JOptionPane;
4
5 public class Repteis extends Animal{
6     private String cordeescama;
7
```

Figura 5. Exemplo de uma classe chamada Répteis.

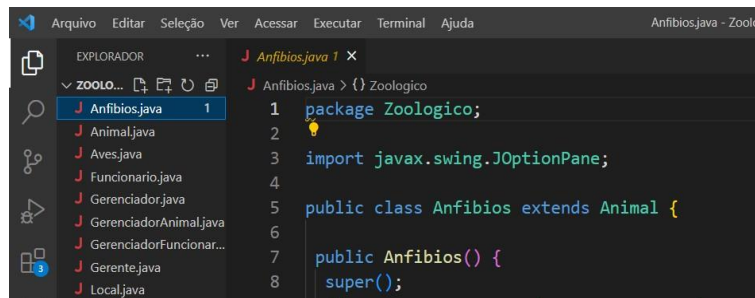


Figura 6. Exemplo de uma classe chamada Anfibios.

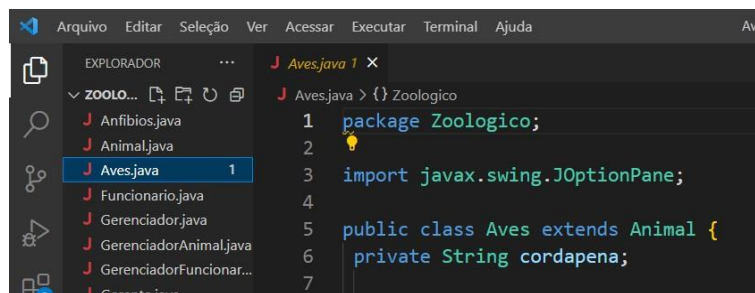


Figura 7. Exemplo de uma classe chamada Aves.

Usamos também classes para podermos representar funcionários que trabalhavam no zoológico, são eles: Gerente, Veterinário e o Zelador, onde cada classe possui um método para calcular seu salário dependendo do seu cargo e da sua função. E também, fizemos a classe chamada “GerenciadorFuncionários” e criamos uma lista de funcionários para podermos manipular os funcionários, ou seja, adicionar, remover, imprimir, entre outros. Exemplos no sistema:

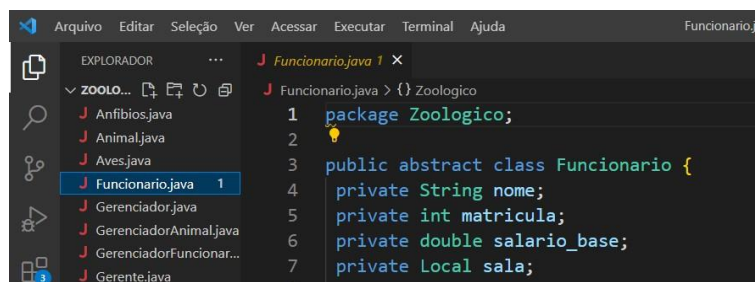


Figura 8. Exemplo de classe chamada “Funcionario”.

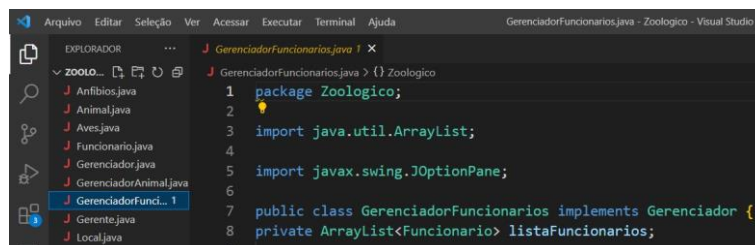


Figura 9. Exemplo de classe chamada “GerenciadorFuncionarios”.

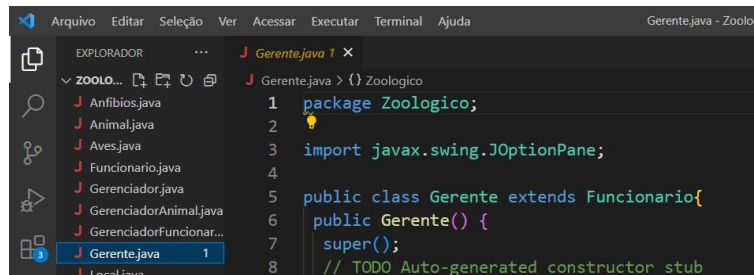


Figura 10. Exemplo de classe chamada “Gerente”.

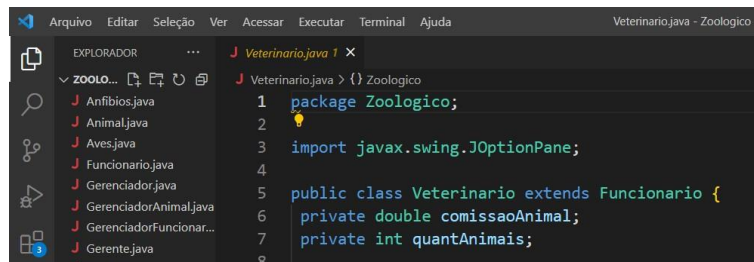


Figura 11. Exemplo de classe chamada “Veterinario”.

Temos também a classe Local que seria para definir jaulas para animais e salas para funcionários. Exemplo:

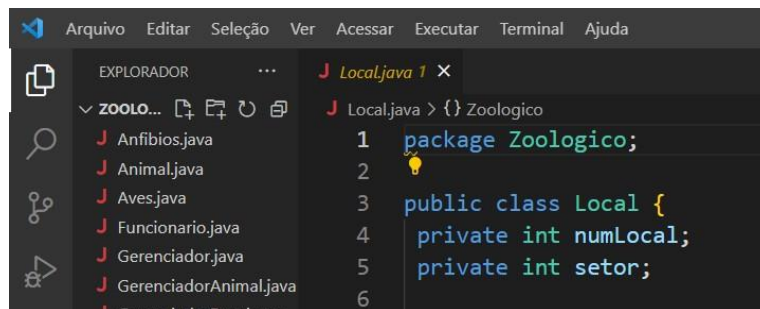


Figura 12. Exemplo de classe chamada Local.

E por último temos a classe principal que é para podermos instanciar as nossas classes e testarmos o nosso sistema. Exemplo:

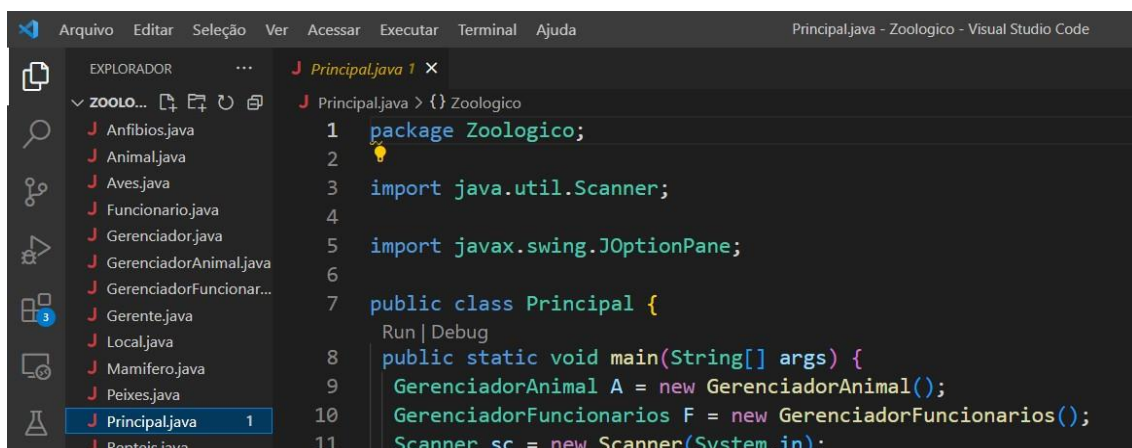


Figura 13. Exemplo de classe chamada Principal.

3.1.1. OBJETOS

O objeto é um elemento que irá representar alguma entidade seja ela abstrata ou concreta. Usamos os objetos para criarmos uma instância de Anfíbios, assim como os outros animais (Mamífero, Aves, Répteis e Peixes) e assim o usuário irá passar valores para essas instâncias. Exemplo no sistema:

```
case 1:
    A.AddAnimal(new Anfibios(
        Double.parseDouble(JOptionPane.showInputDialog(parentComponent: null, message: "INFORME O PESO DO ANFÍBIO: ", title: "ANFÍBIO", JOptionPane.INFORMATION_MESSAGE)),
        Integer
```

Figura 14. Exemplo de uma instância de um anfíbio.

```
case 1:
    F.addFuncionario(new Gerente(
        JOptionPane.showInputDialog(parentComponent: null, message: "NOME DO GERENTE: ", title: "GERENTE", JOptionPane.INFORMATION_MESSAGE),
        Integer.parseInt(JOptionPane.showInputDialog(parentComponent: null, message: "MATRÍCULA DO GERENTE: ", title: "GERENTE",
```

Figura 15. Exemplo de uma instância de um gerente.

Assim como em animais, utilizamos os objetos também para criarmos instâncias do “Gerente”, “Veterinario” e “Zelador” que são funcionários e assim o usuário irá passar os valores para essas instâncias.

3.1.2. MÉTODOS ASSESSORES

Os métodos de acesso e modificação são utilizados apenas em classes que possuam características próprias, para que possamos modificar seu valor e para pegarmos esse valor. Com isso, utilizamos os métodos de acesso para guardar ou mudar o valor que o usuário irá passar, como no caso do peso dos animais que podemos mudar através do método de acesso “setPeso” que pega o valor que o usuário passou e guarda. Foram utilizados nas classes: “Animal”, “Aves”, “Funcionario”, “Local”, “Mamifero”, “Peixes”, “Repteis” e “Veterinario”. Observe o exemplo abaixo:

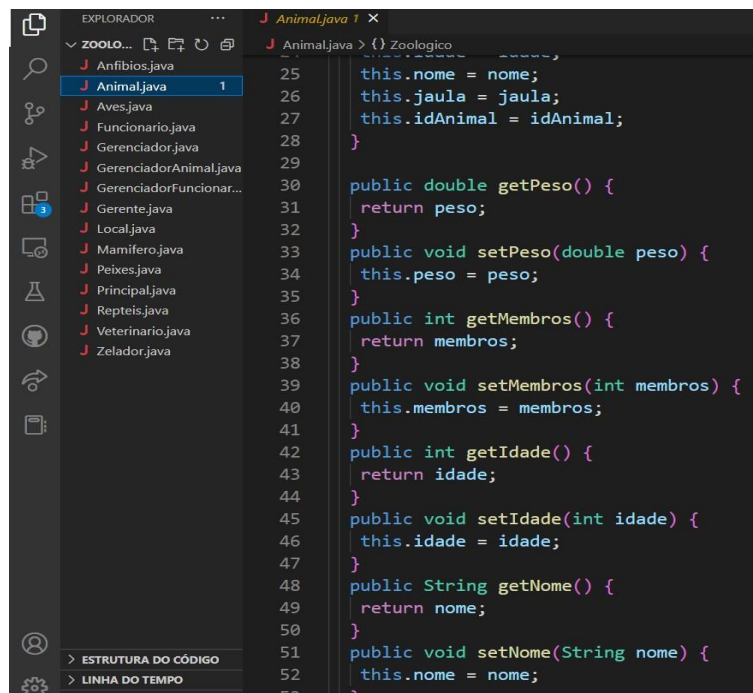


Figura 16. Exemplo de métodos assessores (get e set).

3.1.3. TOSTRING

O toString é uma representação em texto do seu objeto, ou seja, é utilizado para apresentar os atributos de determinadas classes. E também, o toString pode ser utilizado em todas as classes para definir a string de saída quando for imprimir seu objeto, caso não seja declarada no programa o método ele será sobrescrito da sua classe Object. Além disso, o método foi usado em nosso sistema para apresentar as características de determinadas classes de um determinado objeto, como por exemplo, a classe animal que está para retornar o nome do animal e suas características. E também, em cada classe, o método retornará suas características específicas. Como segue o exemplo:

```

66 }
67 @Override
68 public String toString() {
69     return nome + " - INFORMAÇÕES: \n" + " PESO: " + peso + "\n QUANTIDADE DE MEMBROS:" + membros + "\n IDADE:" +
70     idade + "\n" + jaula + "\n ID DO ANIMAL:" + idAnimal;
71 }

```

Figura 17. Exemplo de toString de animal.

3.2. ENCAPSULAMENTO

O Encapsulamento é utilizado em todas as classes que possuem características ou atributos próprios, para proteger os dados daquela determinada classe, como por exemplo na classe Funcionario onde temos o atributo “salario_base” que utilizamos o private para ter uma maior segurança e para que o usuário não possa definir o salário que ele quiser para os funcionários. Temos em nosso sistema as seguintes classes que utilizamos encapsulamento : “Animal”, “Aves”, “Funcionario” , “GerenciadorAnimal”, “GerenciadorFuncionario” , “Local”, “Mamifero”, “Peixes”, “Repteis” e “Veterinario”. Como segue o exemplo abaixo:

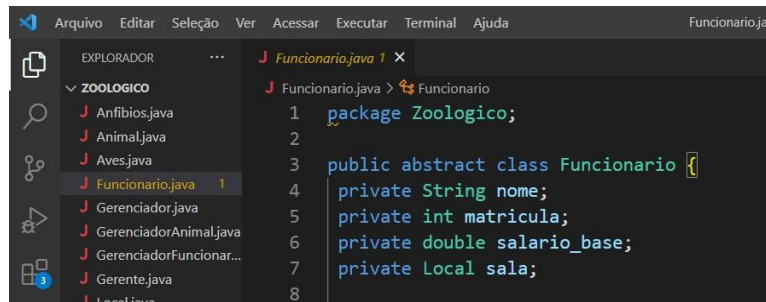


Figura 18. Exemplo de um encapsulamento de atributos ou variáveis na classe Funcionário.

E também, utilizamos encapsulamento na classe “Veterinario” no atributo “comissaoAnimal” para que possamos ter segurança na variável e para que não seja passado para todos os veterinários a mesma comissão.

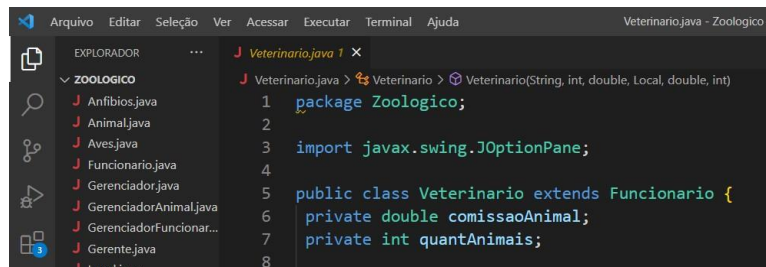


Figura 19. Exemplo de um encapsulamento de atributos ou variáveis na classe Veterinário.

3.3. USO DE CONSTRUTORES E OBJETOS

Os construtores são usados dentro das classes chamadas pelo mesmo nome da classe, onde não indicamos seu retorno. Sendo assim, temos dois tipos de construtores que são eles: o construtor preenchido onde o usuário instancia já passando seus valores e temos o construtor vazio onde se o usuário instanciar e não atribui os valores, conterà um valor atribuído para esses atributos. Com isso, em nosso sistema usamos os dois tipos de construtores para caso o usuário não preencher, ele assim não terá problema. Também foram usados construtores nas classes: “Animal”, “Anfibios”, “Mamifero”, “Aves”, “Repteis”, “Peixes”, “Funcionarios”, “GerenciadorAnimal”, “GerenciadorFuncionarios”, “Gerente”, “Local”, “Veterinario”, “Zelador”. Como no exemplos abaixo:

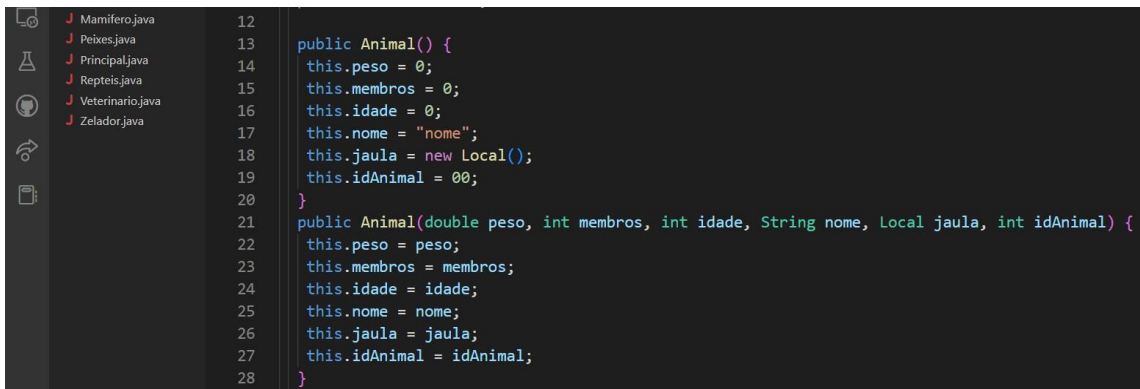
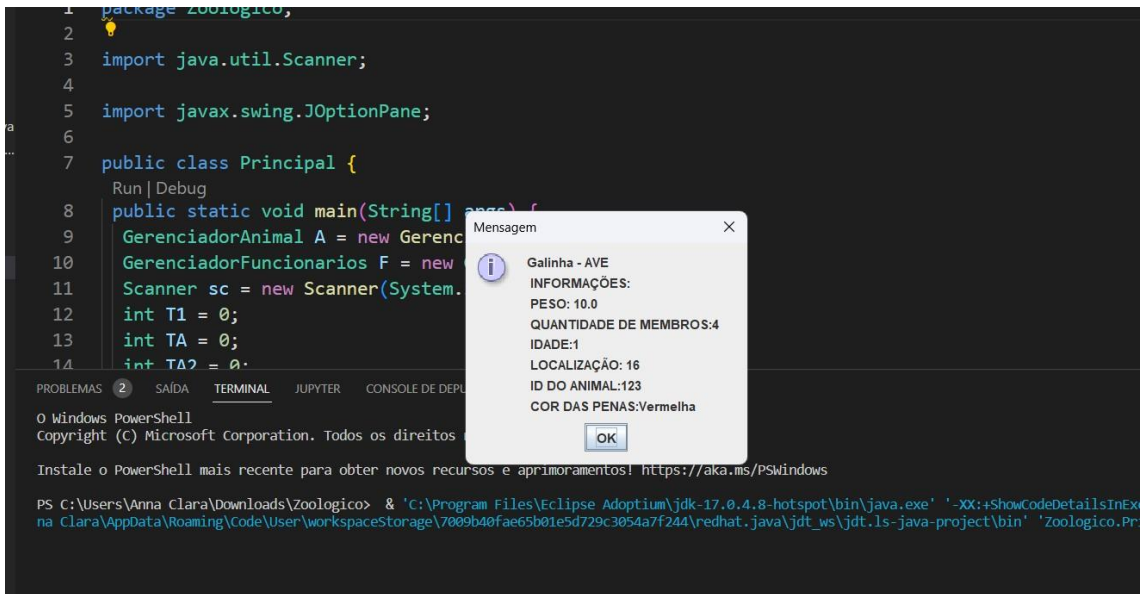


Figura 20. Exemplo de um construtor vazio e um construtor preenchido na classe Animal.

Além disso, criei uma ave em que tem valores dados pelo o usuário e que armazena esses valores nos atributos. Exemplo:



```
1 package Zoologico;
2
3 import java.util.Scanner;
4
5 import javax.swing.JOptionPane;
6
7 public class Principal {
8     Run | Debug
9     public static void main(String[] args) {
10         GerenciadorAnimal A = new GerenciadorAnimal();
11         GerenciadorFuncionarios F = new GerenciadorFuncionarios();
12         Scanner sc = new Scanner(System.in);
13         int T1 = 0;
14         int TA = 0;
15         int TA2 = 0;
16     }
17 }
```

Mensagem

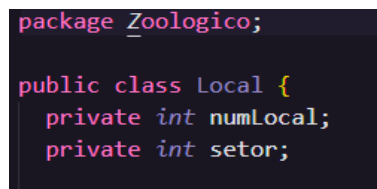
Galinha - AVE
INFORMAÇÕES:
PESO: 10.0
QUANTIDADE DE MEMBROS: 4
IDADE: 1
LOCALIZAÇÃO: 16
ID DO ANIMAL: 123
COR DAS PENAS: Vermelha

OK

Figura 21. Exemplo de Objeto preenchido pelo o usuário.

3.4. COMPOSIÇÃO

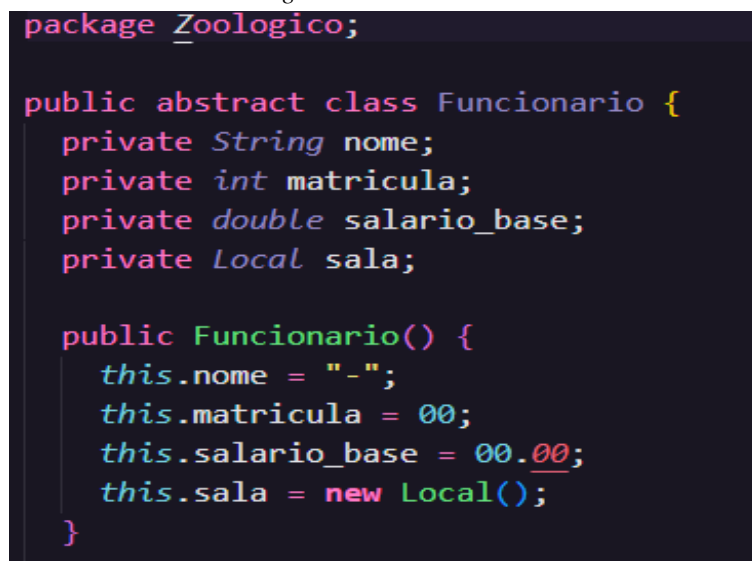
Composição é quando o objeto contém um ou mais objetos de outras classes como membros. Portanto, a composição foi aplicada no projeto abordando que a classe “Local” está implementada tanto na classe “Animal” quanto na classe “Funcionario“. Dessa forma, podemos dizer que o Animal e o Funcionário tem um local. Observe os exemplos abaixo:



```
package Zoologico;

public class Local {
    private int numLocal;
    private int setor;
}
```

Figura 22. Classe Local



```
package Zoologico;

public abstract class Funcionario {
    private String nome;
    private int matricula;
    private double salario_base;
    private Local sala;

    public Funcionario() {
        this.nome = "-";
        this.matricula = 00;
        this.salario_base = 00.00;
        this.sala = new Local();
    }
}
```

Figura 23. Classe “Funcionario” tem composição da Classe local

```

package Zoologico;

import javax.swing.JOptionPane;

public abstract class Animal {
    private double peso;
    private int membros;
    private int idade;
    private String nome;
    private Local jaula;
    private int idAnimal;

    public Animal() {
        this.peso = 0;
        this.membros = 0;
        this.idade = 0;
        this.nome = "nome";
        this.jaula = new Local();
        this.idAnimal = 00;
    }
}

```

Figura 24. Classe “Animal” tem composição da classe Local

3.5. HERANÇA

A herança ocorre quando o objeto da classe filha pode ser tratado como um objeto da classe mãe. Dessa maneira, a herança foi implementada no projeto abordando que as classes mães são “Funcionario” e “Animal”. Logo, as classes filhas de Funcionário são: classe Gerente, Veterinário e Zelador; e as classes filhas de Animal são: classe Anfíbios, Aves, Peixes, Mamífero e Répteis. Observe os exemplos abaixo:

- Classe mãe Funcionário

```

package Zoologico;

import javax.swing.JOptionPane;

public class Veterinario extends Funcionario {
    private double comissaoAnimal;
    private int quantAnimais;
}

```

Figura 24. Classe Veterinario com herança da Classe Funcionario

```
package _Zoologico;

import javax.swing.JOptionPane;

public class Zelador extends Funcionario {
    public Zelador() {
```

Figura 25. Classe Zelador com herança da Classe Funcionario

```
package _Zoologico;

import javax.swing.JOptionPane;

public class Gerente extends Funcionario{
    public Gerente() {
```

Figura 26. Classe Gerente com herança da Classe Funcionario

- Classe mãe Animal:

```
package _Zoologico;

import javax.swing.JOptionPane;

public class Repteis extends Animal{
    private String cordeescama;
```

Figura 27. Classe Repeteis com herança da Classe Animal

```
package _Zoologico;

import javax.swing.JOptionPane;

public class Anfibios extends Animal {
```

Figura 28. Classe Anfibios com herança da Classe Animal

```
package Zoologico;

import javax.swing.JOptionPane;

public class Aves extends Animal {
    private String cordapena;
}
```

Figura 29. Classe Aves com herança da Classe Animal

```
package Zoologico;

import javax.swing.JOptionPane;

public class Mamifero extends Animal{
    private String cordepelo;
    private int tempodegestacao;
}
```

Figura 30. Classe Mamifero com herança da Classe Animal

```
package Zoologico;

import javax.swing.JOptionPane;

public class Peixes extends Animal {
    private String cordeescama;
}
```

Figura 31. Classe Peixes com herança da Classe Animal

3.6. COLEÇÕES (ARRAYLIST)

ArrayList é uma implementação da interface List que utiliza um vetor para armazenar elementos. Dessa maneira, o ArrayList foi implementado nas classes de “GerenciadorFuncionario” e “GerenciadorAnimal” com o propósito de gerar lista para “Animal” e “Funcionario”. Observe exemplos abaixo:

```

package _Zoologico;

import java.util.ArrayList;

import javax.swing.JOptionPane;

public class GerenciadorFuncionarios implements Gerenciador {
    private ArrayList<Funcionario> listaFuncionarios;
}

```

Figura 32. Classe “GerenciadorFuncionario” possui uma ArrayList “listaFuncionario”.

```

package _Zoologico;

import java.util.ArrayList;

import javax.swing.JOptionPane;

public class GerenciadorAnimal implements Gerenciador {
    private ArrayList<Animal> listaAnimal;
}

```

Figura 33. Classe “GerenciadorAnimal” possui uma ArrayList “listaAnimal”.

3.7. CLASSES E MÉTODOS ABSTRATOS / INTERFACE

3.7.1. CLASSE ABSTRATA

As classes abstratas são usadas na Programação Orientada a Objeto para serem ancestrais de outras classes e não para serem instanciadas. Ao tornar uma classe abstrata, o programador impede que sejam criadas instâncias dela. Frequentemente é criada uma classe mãe para conter o código que é comum a todas as suas classes filhas (FANDOM, 2021).

No sistema de gerenciamento do zoológico em questão a função é aplicada nas Classes ancestrais: “Animal” que possui como descendentes “Anfibios”, “Aves”, “Mamiferos”, “Peixes” e “Repteis”; e “Funcionario” que possui como descendentes “Gerente”, “Veterinario” e “Zelador”, são possíveis de visualizar nas imagens abaixo.

```

1  package Zoologico;
2
3  import javax.swing.JOptionPane;
4
5  public abstract class Animal {

```

Figura 34 – Classe abstrata “Animal”.

```

1  package Zoologico;
2
3  public abstract class Funcionario {

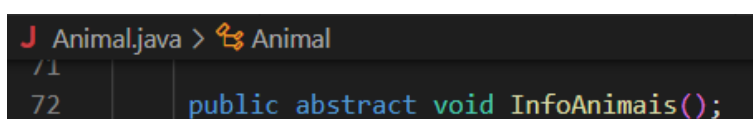
```

Figura 35 – Classe abstrata “Funcionario”.

3.7.2. MÉTODOS ABSTRATOS

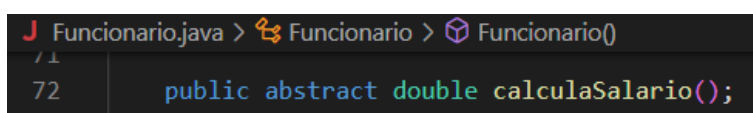
Classes abstratas podem conter métodos abstratos. Um método abstrato é declarado na classe, mas não tem seu código implementado, ele será implementado em alguma classe descendente. Uma classe abstrata pode ou não ter métodos abstratos, mas se uma classe contém métodos abstratos, então obrigatoriamente a classe deve ser declarada abstrata. Uma classe descendente de uma classe abstrata precisa implementar o código de todos os métodos abstratos para se tornar concreta e poder ser instanciada (FANDOM, 2021).

No sistema de gerenciamento do zoológico em questão os métodos abstratos estão presentes nas classes abstratas “Animal” com o método “InfoAnimais”, e “Funcionario” com o método “calculaSalario”, sendo implementadas nas classes filhas, os métodos abstratos estão exposto nas imagens abaixo.



```
J Animal.java > Animal
/1
72 public abstract void InfoAnimais();
```

Figura 36 – Método abstrato “InfoAnimais”.



```
J Funcionario.java > Funcionario > Funcionario()
/1
72 public abstract double calculaSalario();
```

Figura 37 – Método abstrato “calculaSalario”.

3.7.3. INTERFACE

Uma interface é uma estrutura análoga com uma classe, podendo ter métodos e atributos, mas que não implementa nada realmente, não tem nenhum código, então está muito longe de ser uma classe. Os métodos em uma interface são sempre abstratos e públicos. Podem ter atributos, mas serão sempre públicos, estáticos e finais, não permitindo a modificação do valor. Assim como as classes abstratas, a função das interfaces é criar uma linguagem padrão de comunicação com as classes, definindo métodos que serão implementados nas classes, porém, diferente das classes abstratas pode-se adicionar quantas interfaces forem necessárias (FANDOM, 2021).

A aplicação dessa estrutura no sistema de gerenciamento do zoológico está na interface “Gerenciador”, sendo utilizada para definir os métodos necessários nas classes “GerenciadorFuncionarios” e “GerenciadorAnimal”. A estrutura presente na interface pode ser visualizada na figura a seguir.

```

J Gerenciador.java > ...
1  package Zoologico;
2
3  public interface Gerenciador {
4
5      public void remover(int N);
6      public void imprimir(int M);
7      public void imprimirTodos();
8  }
9

```

Figura 38 – Interface “Gerenciador”.

3.8. POLIMORFISMO DE INCLUSÃO/ SOBREPOSIÇÃO E SOBRECARGA

3.8.1. POLIMORFISMO DE INCLUSÃO/SOBREPOSIÇÃO

O polimorfismo de inclusão / sobreposição ocorre quando uma classe filha implementa uma nova versão de um método que existe em alguma classe mãe, usando exatamente a mesma assinatura. Nesse caso, a nova versão se sobrepõe à versão antiga.

Utilizamos esse tipo de polimorfismo principalmente nos métodos abstratos mencionados anteriormente, nas imagens a seguir podemos visualizar o método “InfoAnimais” implementadas de maneiras distintas, expondo informações específicas de cada classe dos animais.

```

J Anfibios.java > Anfibios > Anfibios(double, int, String, Local, int)
17  @Override
18  public void InfoAnimais() {
19      // TODO Auto-generated method stub
20      JOptionPane.showMessageDialog(parentComponent: null, message: "PRINCIPAIS CARACTERÍSTICAS = A MAIORIA TEM QUATRO MENBROS. NA FASE LA
21  }

```

Figura 39 – Polimorfismo de inclusão/sobreposição no método “InfoAnimais” – Anfibios.

```

J Mamifero.java > {} Zoologico
44  @Override
45  public void InfoAnimais() {
46      // TODO Auto-generated method stub
47      JOptionPane.showMessageDialog(parentComponent: null, message: "PRINCIPAIS CARACTERÍSTICAS = PELOS E GLÂNDULAS MAMÁRIAS \n REPRODUÇÃO
48  }

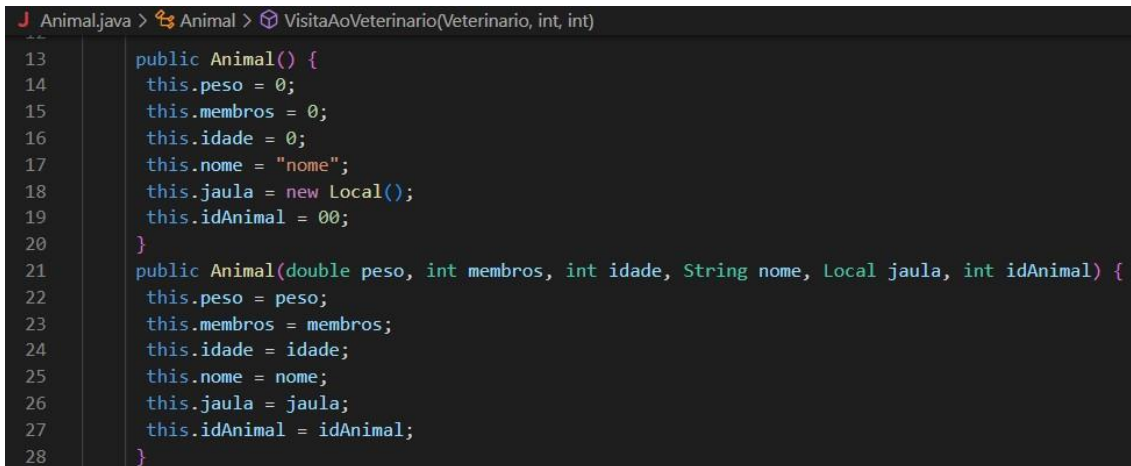
```

Figura 40 – Polimorfismo de inclusão/sobreposição no método “InfoAnimais” – Mamíferos.

3.8.2. POLIMORFISMO DE SOBRECARGA

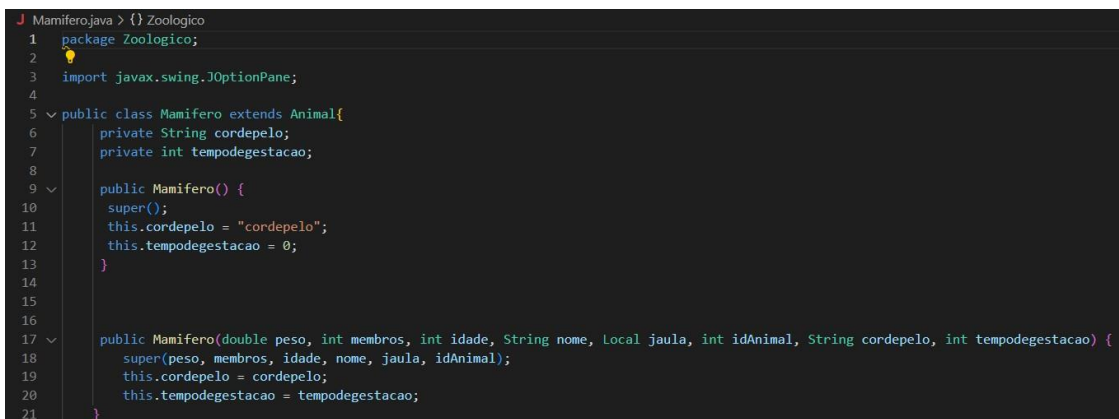
Polimorfismo de Sobrecarga é um recurso que permite uma classe implementar várias versões do mesmo método, usando o mesmo identificador, desde que estas versões tenham assinaturas diferentes. A sobrecarga pode ocorrer na mesma classe ou em classes descendentes. Esse importante recurso permite que a mesma tarefa seja executada de várias maneiras diferentes dependendo das informações que estão disponíveis no momento (FANDOM, 2021).

Esse recurso está bastante presente nos construtores, como exposto nas imagens a seguir:



```
Animal.java > Animal > VisitaAoVeterinario(Veterinario, int, int)
13 public Animal() {
14     this.peso = 0;
15     this.membros = 0;
16     this.idade = 0;
17     this.nome = "nome";
18     this.jaula = new Local();
19     this.idAnimal = 00;
20 }
21 public Animal(double peso, int membros, int idade, String nome, Local jaula, int idAnimal) {
22     this.peso = peso;
23     this.membros = membros;
24     this.idade = idade;
25     this.nome = nome;
26     this.jaula = jaula;
27     this.idAnimal = idAnimal;
28 }
```

Figura 41 – Construtores - Animal.



```
Mamifero.java > {} Zoologico
1 package Zoologico;
2
3 import javax.swing.JOptionPane;
4
5 public class Mamifero extends Animal{
6     private String cordepelo;
7     private int tempodegestacao;
8
9     public Mamifero() {
10         super();
11         this.cordepelo = "cordepelo";
12         this.tempodegestacao = 0;
13     }
14
15
16
17     public Mamifero(double peso, int membros, int idade, String nome, Local jaula, int idAnimal, String cordepelo, int tempodegestacao) {
18         super(peso, membros, idade, nome, jaula, idAnimal);
19         this.cordepelo = cordepelo;
20         this.tempodegestacao = tempodegestacao;
21     }
22 }
```

Figura 42 – Construtores – Mamíferos.

3.9. DOWNCASTING E UPCASTING

3.9.1. DOWNCASTING

Downcasting é quando o objeto é transformado em um subtipo dele. Não há garantias que funcione e pode haver necessidade de conversões. O compilador só aceita se ele puder provar que o objeto se encaixa perfeitamente e seja de fato aquele objeto. Por isso deve ser explicitado pelo programador quando deseja essa ação.

No sistema de gerenciamento do zoológico esse recurso é utilizado para possibilitar o acesso à função específica de mamífero, como exposto na imagem a seguir.

```

J Principal.java > {} Zoologico
215 case 7:
216     int IDG = Integer.parseInt(JOptionPane.showInputDialog(parentComponent: null, message: "ID DO ANIMAL:", title: "ANÁLISE DA GESTAÇÃO",
217         JOptionPane.INFORMATION_MESSAGE));
218     Animal aux = A.buscarAnimal(IDG);
219     if (aux instanceof Mamifero) {
220         Mamifero MG = (Mamifero) A.buscarAnimal(IDG);
221         MG.Analisegestacao(Integer.parseInt(JOptionPane.showInputDialog(parentComponent: null, message: "TEMPO DE GESTAÇÃO: ",
222             title: "ANÁLISE DA GESTAÇÃO", JOptionPane.INFORMATION_MESSAGE)));
223     } else {
224         JOptionPane.showMessageDialog(parentComponent: null, message: "OPÇÃO INVÁLIDA! ANIMAL NÃO É UM MAMÍFERO.", title: "ERRO",
225             JOptionPane.ERROR_MESSAGE);
226     }
227     break;

```

Figura 43 – Método “Analisedegestacao” aplicado em um mamífero “MG”.

3.9.2. UPCASTING

Upcasting é quando um objeto é convertido em um supertipo dele. Ele sempre funcionará já que todo objeto é completamente compatível com um tipo do qual ele foi derivado.

No sistema de gerenciamento do zoológico esse recurso é utilizado para instanciar um novo objeto, pois é impossível instanciar uma classe abstrata, porém posteriormente é possível converter da classe filha para a mãe “Animal”, como apresentado nas imagens a seguir.

```

J Principal.java > {} Zoologico
37 A.AddAnimal(new Anfibios(
38     Double.parseDouble(JOptionPane.showInputDialog(parentComponent: null, message: "INFORME O PESO DO ANFÍBIO: ", title: "ANFÍBIOS",
39         JOptionPane.INFORMATION_MESSAGE)),
40     Integer
41         .parseInt(JOptionPane.showInputDialog(parentComponent: null, message: "INFORME A QUANTIDADE DE MEMBROS DO ANFÍBIO: ",
42             title: "ANFÍBIOS", JOptionPane.INFORMATION_MESSAGE)),
43     Integer.parseInt(JOptionPane.showInputDialog(parentComponent: null, message: "INFORME A IDADE DO ANFÍBIO: ", title: "ANFÍBIOS",
44         JOptionPane.INFORMATION_MESSAGE)),
45     JOptionPane.showInputDialog(parentComponent: null, message: "NOME DO ANFÍBIO:", title: "ANFÍBIOS",
46         JOptionPane.INFORMATION_MESSAGE),
47     new Local(Integer.parseInt(JOptionPane.showInputDialog(parentComponent: null,
48         message: "INFORME O NÚMERO DA JAULA DO ANFÍBIO: ", title: "ANFÍBIOS", JOptionPane.INFORMATION_MESSAGE)), 2),
49     Integer.parseInt(JOptionPane.showInputDialog(parentComponent: null, message: "INFORME O ID DO ANFÍBIO: ", title: "ANFÍBIOS",
50         JOptionPane.INFORMATION_MESSAGE)));
51
52 break;
53
54 case 2:
55     A.AddAnimal(new Aves(
56         Double.parseDouble(JOptionPane.showInputDialog(parentComponent: null, message: "INFORME O PESO DA AVE: ", title: "AVES",
57             JOptionPane.INFORMATION_MESSAGE)),
58         Integer.parseInt(JOptionPane.showInputDialog(parentComponent: null, message: "INFORME A QUANTIDADE DE MEMBROS DA AVE: ",
59             title: "AVES", JOptionPane.INFORMATION_MESSAGE)),
60         Integer.parseInt(JOptionPane.showInputDialog(parentComponent: null, message: "INFORME A IDADE DA AVE: ", title: "AVES",
61             JOptionPane.INFORMATION_MESSAGE)),
62         JOptionPane.showInputDialog(parentComponent: null, message: "NOME DA AVE: ", title: "AVES", JOptionPane.INFORMATION_MESSAGE),
63         new Local(Integer.parseInt(JOptionPane.showInputDialog(parentComponent: null,
64             message: "INFORME O NÚMERO DA JAULA DA AVE: ", title: "AVES", JOptionPane.INFORMATION_MESSAGE)), 3),
65         Integer.parseInt(JOptionPane.showInputDialog(parentComponent: null, message: "INFORME O ID DA AVE: ", title: "AVES",
66             JOptionPane.INFORMATION_MESSAGE)),
67         JOptionPane.showInputDialog(parentComponent: null, message: "COR DAS PENAS DA AVE: ", title: "AVES",
68             JOptionPane.INFORMATION_MESSAGE)));
69
70 break;

```

Figura 44 – Adição de um Anfíbio e uma Ave no ArrayList de “Animal”.

```

J Principal.java > () Zoologico
244 case 1:
245     F.addFuncionario(new Gerente(
246         JOptionPane.showInputDialog(parentComponent: null, message: "NOME DO GERENTE: ", title: "GERENTE",
247             JOptionPane.INFORMATION_MESSAGE),
248         Integer.parseInt(JOptionPane.showInputDialog(parentComponent: null, message: "MATRÍCULA DO GERENTE: ", title: "GERENTE",
249             JOptionPane.INFORMATION_MESSAGE)),
250         Double.parseDouble(JOptionPane.showInputDialog(parentComponent: null, message: "SALÁRIO BASE DO GERENTE: ", title: "GERENTE",
251             JOptionPane.INFORMATION_MESSAGE)),
252         new Local(Integer.parseInt(JOptionPane.showInputDialog(parentComponent: null, message: "INFORME A SALA DO GERENTE: ",
253             title: "GERENTE", JOptionPane.INFORMATION_MESSAGE)), 1));
254     break;
255
256 case 2:
257     F.addFuncionario(new Veterinario(
258         JOptionPane.showInputDialog(parentComponent: null, message: "NOME DO VETERINÁRIO: ", title: "VETERINÁRIO",
259             JOptionPane.INFORMATION_MESSAGE),
260         Integer.parseInt(JOptionPane.showInputDialog(parentComponent: null, message: "MATRÍCULA DO VETERINÁRIO: ", title: "VETERINÁRIO",
261             JOptionPane.INFORMATION_MESSAGE)),
262         Double.parseDouble(JOptionPane.showInputDialog(parentComponent: null, message: "SALÁRIO BASE DO VETERINÁRIO: ",
263             title: "VETERINÁRIO", JOptionPane.INFORMATION_MESSAGE)),
264         new Local(Integer.parseInt(JOptionPane.showInputDialog(parentComponent: null, message: "INFORME A SALA DO VETERINÁRIO: ",
265             title: "VETERINÁRIO", JOptionPane.INFORMATION_MESSAGE)), 1),
266         comissaoAnimal: 250,
267         Integer.parseInt(JOptionPane.showInputDialog(parentComponent: null, message: "QUANTIDADE DE ANIMAIS ATENDIDOS: ",
268             title: "VETERINÁRIO", JOptionPane.INFORMATION_MESSAGE)));
269     break;
270 case 3:
271     F.addFuncionario(new Zelador(
272         JOptionPane.showInputDialog(parentComponent: null, message: "NOME DO ZELADOR: ", title: "ZELADOR",
273             JOptionPane.INFORMATION_MESSAGE),
274         Integer.parseInt(JOptionPane.showInputDialog(parentComponent: null, message: "MATRÍCULA DO ZELADOR: ", title: "ZELADOR",
275             JOptionPane.INFORMATION_MESSAGE)),
276         Double.parseDouble(JOptionPane.showInputDialog(parentComponent: null, message: "SALÁRIO BASE DO ZELADOR: ", title: "ZELADOR",
277             JOptionPane.INFORMATION_MESSAGE)),
278         new Local(Integer.parseInt(JOptionPane.showInputDialog(parentComponent: null, message: "INFORME A SALA DO ZELADOR: ",
279             title: "ZELADOR", JOptionPane.INFORMATION_MESSAGE)), 1));
280     break;

```

Figura 45– Adição de um gerente, um veterinário e um zelador no ArrayList de “Funcionario”.

3.10. CLASSE GENÉRICA

Uma classe genérica permite a entrada de vários tipos de dados como parâmetro, seja ele: caractere, int, double, int, integer, String ou qualquer outro definido pelo usuário, fazendo ser possível criar classe que trabalhem com diferentes tipos de dados e criar listas de diferentes tipos. Com isso, no nosso projeto utilizamos classes genéricas para criarmos uma lista de animais na classe “GerenciadorAnimais” e também criamos na classe “GerenciadorFuncionarios” uma lista de funcionários para que possamos manipular essas listas adicionando objetos específicos em cada tipo de lista. Exemplos no sistema:

```

package Zoologico;

import java.util.ArrayList;

import javax.swing.JOptionPane;

public class GerenciadorAnimal implements Gerenciador{
    private ArrayList<Animal> listaAnimal;

```

Figura 46. Exemplo de Classe genérica no sistema.


```
package Zoologico;

import java.util.ArrayList;

import javax.swing.JOptionPane;

public class GerenciadorFuncionarios implements Gerenciador {
    private ArrayList<Funcionario> listaFuncionarios;
}
```

Figura 47. Exemplo de Classe genérica no sistema.

3.11. EXCEÇÕES

As exceções quando não forem tratadas no programa, ela irá se encerrar apresentando uma mensagem de erro, então usamos o tratamento de exceções para que seja capturada as exceções existentes no programa e sejam tratadas sem precisar encerrar o programa e continuar a execução normalmente, mas na maioria das vezes esses erros são quando o usuário acessa recursos que não está disponível no programa.

No nosso sistema utilizamos as exceções para evitar que o usuário digite alguma coisa que não seja o que o programa requer e dê um erro de lógica. Com isso, implementamos uma função que foi aplicada o try e o catch, onde nessa função pedimos para o usuário digitar apenas um número para saber se o animal está ou não doente, que será 1 para sim e 2 para não, caso ele digite uma string o programa não será interrompido e não finaliza a execução com uma mensagem de erro, mas sim irá continuar com a normal execução do sistema para o usuário. Exemplos no sistema:

```
public abstract void InfoAnimais();
public void VisitaAoVeterinario(Veterinario P, int V, int idAnimal) {
    // TODO Auto-generated method stub
    do {
        try {
            V = Integer.parseInt(JOptionPane.showInputDialog(parentComponent: null, "O ANIMAL" + idAnimal + " ESTÁ DOENTE?\n 1 - SIM \n 2 - NÃO", title: "VISITA AO VETERINÁRIO", JOptionPane.INFORMATION_MESSAGE));
        } catch (Exception E) {
            JOptionPane.showMessageDialog(parentComponent: null, P.getNome() + "DEVE REALIZAR CONSULTA PARA O ANIMAL: " + idAnimal, title: "VISITA AO VETERINÁRIO", JOptionPane.INFORMATION_MESSAGE);
        }
        if (V == 1) {
            JOptionPane.showMessageDialog(parentComponent: null, P.getNome() + "DEVE REALIZAR CONSULTA PARA O ANIMAL: " + idAnimal, title: "VISITA AO VETERINÁRIO", JOptionPane.INFORMATION_MESSAGE);
        } else {
            if (V == 2){
                JOptionPane.showMessageDialog(parentComponent: null, idAnimal + " COM SAÚDE!", title: "VISITA AO VETERINÁRIO", JOptionPane.INFORMATION_MESSAGE);
            }
        }
        if (V != 1 && V != 2){
            JOptionPane.showMessageDialog(parentComponent: null, message: "VALOR INVÁLIDO!", title: "VISITA AO VETERINÁRIO", JOptionPane.INFORMATION_MESSAGE);
        }
    } while (V != 1 && V != 2);
}
```

Figura 48. Exemplo de Exceções no sistema.

E também, como as exceções são muito comuns e vemos que o tratamento delas são muito importantes, aplicamos o tratamento de exceções com o try e catch na nossa classe Principal, pois com ele evitaria de ocorrer algum erro e interromper o programa quando o usuário estiver executando. Exemplos no sistema:

Início do programa na principal:


```

package Zoologico;

import java.util.Scanner;

import javax.swing.JOptionPane;

public class PrincipalTeste {
    Run | Debug
    public static void main(String[] args) {
        GerenciadorAnimal A = new GerenciadorAnimal();
        GerenciadorFuncionarios F = new GerenciadorFuncionarios();
        Scanner sc = new Scanner(System.in);
        int T1 = 0;
        int TA = 0;
        int TA2 = 0;
        int TAVV = 0;
        int TAINF = 0;
        int TF = 0;
        int TF2 = 0;
        int TFCS = 0;

        do{
            try {
                T1 = Integer.parseInt(JOptionPane.showInputDialog(parentComponent: null,
                message: "DIGITE:\n1-ANIMAIS\n2-FUNCINARIOS\n3-SAIR DO PROGRAMA", title: "MENU", JOptionPane.
                INFORMATION_MESSAGE));
                switch(T1) {
                    case 1:

```

Figura 49. Exemplo de Classe genérica no sistema.

Fim do programa:

```

243         }
244     } catch (Exception E) {
245         JOptionPane.showMessageDialog(parentComponent: null, message: "VALOR INVÁLIDO!",
246         title: "ERRO!", JOptionPane.INFORMATION_MESSAGE);
247     }
248     }while(T1 != 3);
249     sc.close();
250 }

```

Figura 50. Exemplo de Classe genérica no sistema.

4. CONCLUSÃO

A partir do resultado do sistema de gerenciamento de um zoológico, concluímos que tivemos algumas dificuldades no decorrer do nosso programa e que foram de suma importância para que pudéssemos buscar conhecimentos além do adquirido. Tivemos algumas dificuldades na função visita ao veterinário, em exceções e E/S com JOptionPane, pois foi preciso um trabalho em grupo, uma lógica mais complexa e alguns conhecimentos buscados. No meio desse tempo mudamos bastante do planejado no início, vimos que algumas coisas que colocamos não fazia mais sentido, adicionamos funções novas, novas classes, novos atributos adicionamos uma interface para uma melhor experiência para o usuário, fizemos tudo para ir se adaptando com a ideia que queríamos chegar no final de tudo.

Sem dúvidas, não foi um desafio fácil, porém conseguimos desenvolver o nosso sistema proposto apesar das dificuldades, deixando o sistema com as funcionalidades e a execução que planejamos no início. Com isso, conseguimos alcançar nosso objetivo de concluir nosso sistema, deixando pronto para execução e uso do usuário.