Fundamentos da Programação Orientada a Objetos

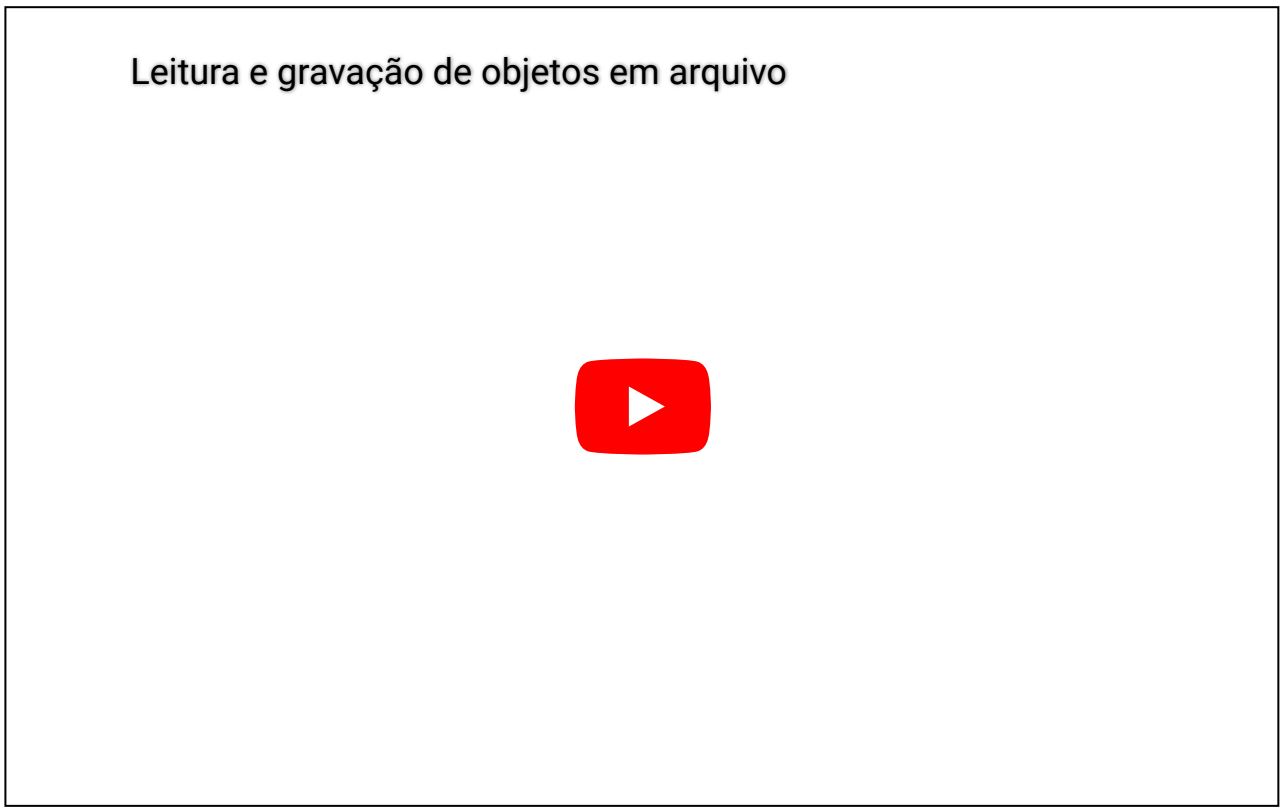
UNIDADE 08

Componentes de Entrada e Saída

*Nesta última Unidade da disciplina, serão apresentadas algumas alternativas para **troca de dados** entre nossos programas e outros elementos, tais como:*

- **Arquivos para escrita (ou saída) e leitura (ou entrada) de dados**
  - As informações podem ser **armazenadas** (escritas) e **recuperadas** (lidas) de outros elementos, como arquivos, graças a um sistema de comunicação (troca de dados), conhecido como fluxo (***stream***); essa troca pode ser feita com **dados primitivos**, como bytes, ou com **objetos**.
- **Interface gráfica, para interação por eventos**

- **Eventos** são **ocorrências** que disparam alguma funcionalidade, como quando acionamos um botão em uma interface gráfica e aparece uma janela com alguma mensagem.



Vamos estudar, com exemplos, essas alternativas para troca de dados nas seções a seguir.

Leitura e gravação de dados em arquivo

Nos nossos programas, manipulamos dados por meio de **variáveis**, que podem ser tipos primitivos, vetores, objetos etc. Contudo, esse armazenamento de dados é **temporário**, pois esses **dados se perdem** quando a **variável deixa de existir**, uma vez que a rotina em que a variável foi declarada finaliza ou porque o programa todo termina sua execução.

Para **manter nossos dados a longo prazo**, precisamos **persisti-los**, o que significa que eles serão gravados em **meio não volátil** (como arquivos), para então serem recuperados em outro momento.

> **PERSISTÊNCIA DE DADOS** garante de que um **dado** foi **salvo** (**escrito**) e que poderá ser **recuperado** (**lido**) em um outro momento.

Na computação, **persistência** significa que o **dado** (**estado do programa**) sobrevive ao programa no qual ele foi criado; **dados persistidos** são chamados de **acessíveis a longo prazo**.

Representação binária: no computador, todo dado, seja um texto, arquivo de som, arquivo de imagem, e assim por diante, é representado como um **conjunto de *bits***, que são representados por números **zeros** (**0**) e **uns** (**1**). É o que chamamos de representação binária, por ter apenas **dois símbolos**.

Por essa razão, quando nossos programas leem um dado, eles recebem sua representação binária.

Exemplo: **letra maiúscula A**: **código decimal = 65** e **código binário = 01000001** (conjunto de ***bits***).

É o **código binário** da letra **A** que fica nas variáveis dos nossos programas, que é transportado pela rede, que é armazenado em arquivo, que é passado pelo teclado, e assim por diante. **No computador, tudo é binário**.

São os nossos programas que manipulam esses **códigos binários em *bits*** para apresentar as informações **adequadas** e **esperadas**, como texto, som, imagem etc.

O ***bit*** é a menor informação que o computador pode armazenar.

- Comumente, usado para medir a velocidade da Internet, como **100 Mbps** (megabits 100×10^6 bits por segundo); embora a internet forneça dados em bytes, eles são transportados pela rede de ***bit* em *bit***, um por vez, **em série**.

O ***byte*** é uma unidade de memória que contém **oito *bits***:

- São usados para especificar a quantidade de memória ou a capacidade de armazenamento de um certo dispositivo, como **8 GB** (giga bytes 8x 10^9 bytes) de RAM, **1TB** (tera bytes 1 x 10^{12} GB) de disco rígido etc.

Agora que identificamos ***bits*** e ***bytes***, veja como esses **dados binários** são **armazenados** e **recuperados** por meio de um sistema de comunicação conhecido como fluxo (*stream*). Detalhes na Figura 1 a seguir.

FLUXO (***STREAM***) - representa uma **origem** (*DATA SOURCE*) ou um **destino de dados** (*DATA DESTINATION*), que pode ser:

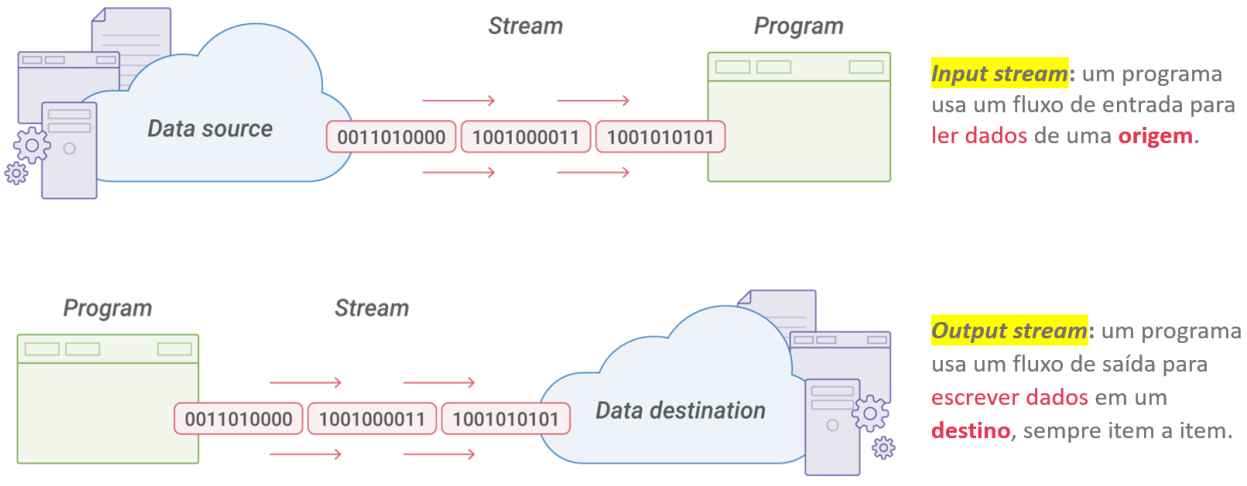
- Arquivos em disco.
- Dispositivos físicos (monitor, teclado etc.).
- Outros programas.
- Um socket de rede*, etc.

O ***stream*** também suporta **diferentes tipos de dados**:

- *Bytes*.
- Vetores.
- Tipos de dados heterogêneos.
- Até **objetos**.

* Ponta final de um fluxo de comunicação entre processos por meio de uma rede de computadores

Figura 1 – Fluxo (*stream*)



Fonte: Autores (2021).

EXERCÍCIO

Fluxo de *bytes*

Neste exercício, apresentamos como utilizar as **classes** *`FileOutputStream`* e *`FileInputStream`* para **mandar** e **receber** dados de um arquivo, ***byte a byte***, ou de **oito em oito *bits***. Observe que essas classes **ocultam** como a manipulação binária é feita – apenas precisamos saber como utilizá-las.

Utilizados para **entrada**/**saída** de *bytes* (8 *bits*).

Existem várias classes de fluxos de byte, todas derivadas de *`OutputStream/InputStream`* :

- Objeto *`System.out`* é um *`OutputStream`* para escrever na **tela**;
- Objeto *`System.in`* é um *`InputStream`* para ler dados do **teclado**.

*Os **bytes** são **inteiros** que variam entre **0** e **255**.*

Dados de um fluxo podem também vir de arquivos:

Formato geral para **leitura** de dados - *`FileInputStream`* (ler)

```
FileInputStream arqIn   = new FileInputStream ("dadosIn.dat");
...
arqIn.close(); //Após leitura, um arquivo deve ser fechado com o método close
```

Formato geral para **escrita** de dados - *`FileOutputStream`* (escrever)

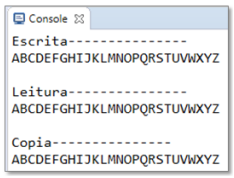
```
FileOutputStream arqOut = new FileOutputStream("dadosOut.dat");
...
arqOut.close(); //Após escrita, um arquivo deve ser fechado com o método close
```

Analise o código do exemplo a seguir, que escreve e lê *bytes* em arquivo, e que também copia, *byte* a *byte*, dados de um arquivo para outro.

Veja se você consegue executar esse exemplo, obtendo a tela de console apresentada.

Figura 2 – Programa simples com fluxo de *bytes*

</>




```
import
java.io.FileInputStream;
import
java.io.FileNotFoundExceptio
import
java.io.FileOutputStream;
import
java.io.IOException;

public class Byte {
    public static void
EscreverDados()  //
escreve bytes para um
arquivo
    {

FileOutputStream arq;
// referência para
arquivo de saída
        try {
            arq = new
FileOutputStream("arq1.txt");
// arquivo na pasta
local
            //Do
caractere A (código 65)
ao Z (código 90)
            for(int
cont = 65; cont < 91;
cont++) {

arq.write(cont); //
Escreve Byte (8bits) em
arquivo

System.out.print((char)cont);
// imprime o código
como char
            }

arq.close();         //
fecha arquivo de
escrita
        } catch
```

```
45  (FileNotFoundException
46  | IOException e) {
47
48  e.printStackTrace();
49          } catch
50  (IOException e) {
51
52  e.printStackTrace();
53          }
54      }
55      public static void
56  LerDados(){  // lê
57  bytes de um arquivo
58          FileInputStream
59  arq; // referência para
60  arquivo de entrada
61          int letra = 0;
62          try {
63              arq = new
64  FileInputStream
65  ("arq1.txt"); //
66  arquivo na pasta local
67              while(letra
68  != -1){     // -1 é o
69  código para fim de
70  arquivo
71                  letra =
72  arq.read(); // Lê Byte
73  (8bits) de arquivo
74                  if
75  (letra != -1)    // Se
76  não fim de arquivo,
77
78  System.out.print((char)letra)
79  // imprime como char
80              }
81
82  arq.close();
83  // fecha arquivo de
84  leitura
85          } catch
86  (FileNotFoundException
87  e) {
88
89  e.printStackTrace();
```

```
90          }catch
91  (IOException e) {
92
93  e.printStackTrace();
94          }
95      }
96      public static void
97  CopiarDados() { //copia
98  um arq para outro, byte
99  a byte
100         FileInputStream
101 in  = null;
102
103 FileOutputStream out =
104 null;
105
106         try {
107             in  = new
108 FileInputStream("arq1.txt");
109             out = new
110 FileOutputStream("arq2.txt");
111             int c;
112             while ((c =
113 in.read()) != -1) { //
114 lê byte de arq1.txt
115
116 out.write(c);
117 // escreve byte para
118 arq2.txt
119
120 System.out.print((char)c);
121 // imprime como char
122             }
123             in.close();
124 // fecha arquivo de
125 leitura
126
127 out.close();   // fecha
128 arquivo de escrita
129
130 }catch(FileNotFoundException
131 e) {
132
133 System.out.println("Arquivo
134 nao encontrado.");
```

```
135
136     }catch(IOException e) {
137
138     System.out.println("Arquivo
139     nao encontrado.");
140             }
141         }
142         public static void
143     main(String[] args)
144     {
145
146     System.out.println("\n\nEscr
147     --------------");
148         EscreverDados();
149
150     System.out.println("\n\nLeitu
151     --------------");
152         LerDados();
153
154     System.out.println("\n\nCopia
155     --------------");
156         CopiarDados();
157         }
158     }
159
160
```

Fonte: Autores (2021).

Neste exercício, apresentamos como utilizar as **classes** `FileWriter` e `FileReader` para **mandar** e **receber** dados de um arquivo, **caractere a caractere**, ou de **16 em 16** ***bits***.

EXERCÍCIO

Fluxo de caracteres

Utilizados para **entrada**/**saída** de caracteres (16 *bits*).

- `FileReader`, para fluxo de caracteres de **entrada**.
- `FileWriter`, para fluxo de caracteres de **saída**.

Os fluxos de caracteres são um tipo **especializado** de **fluxo de *bytes*** para tratar caracteres **Unicode**, um **padrão** para representar e manipular texto dos sistemas de escrita existente. Com o padrão Unicode, é possível codificar cerca de 138 mil caracteres diferentes.

> Veja, no código, o método ***read*** (`c = in.read()`) retorna um **int** tanto para o **fluxo de *bytes*** como para o **fluxo de caracteres**. Contudo:
>
> - **Fluxo de *bytes***, o **inteiro** retornado = **8** *bits* de tamanho.
> - **Fluxo de caracteres**, o **inteiro** retornado = **16** *bits* de tamanho.

Lendo e escrevendo linhas

Para manipular arquivos de textos, com informações separadas por finalizador de linha (***\r, \n, \r\n***), são utilizadas as classes *`BufferedReader`* para melhor a eficiência das operações de **leitura/escrita** de dados, que geralmente são **lentas**, pois acessam o disco.

Para isso, um ***buffer*** (área temporária de memória) é usado para acumular leituras ou escritas.

Além disso, utilizamos também a classe **PrintWriter** para ter mais conforto na escrita do texto. Você já a conhece, pois é a mesma utilizada pelo **System.out**. Ela fornece vários métodos convenientes para a escrita de texto, como o println e o *printf*, usados nas Unidades passadas. Vários fluxos de dados permitem especificar um fluxo de destino em seu construtor. Isso fornece uma forma flexível de combiná-los.

Analise o código do exemplo a seguir, que escreve e lê caracteres em arquivo, e que também copia, **linha a linha**, dados de um **arquivo texto** para outro. Preste especial atenção sobre como

combinamos os fluxos entre si.

Veja se você consegue executar esse exemplo, obtendo a tela de console apresentada.

Figura 3 – Programa simples com fluxo de caracteres

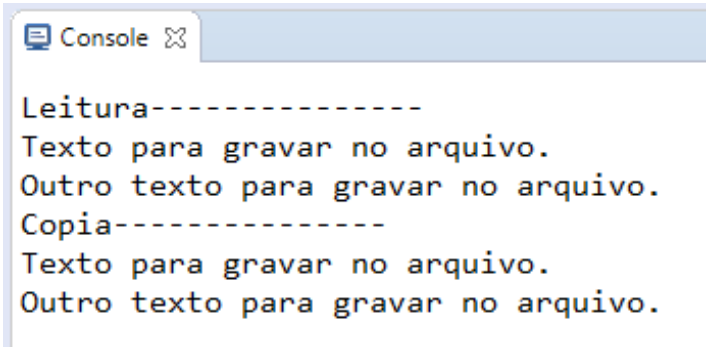
</>

```
import
java.io.BufferedReader;
import
java.io.FileNotFoundExceptio
import
java.io.FileReader;
import
java.io.FileWriter;
import
java.io.IOException;
import
java.io.PrintWriter;

public class
CopiarCaracter {
    public static void
EscreverCaracteres(){
        FileWriter out
= null;
        int contLetra =
0;
        String texto =
"Texto para gravar no
arq.\nOutro texto para
gravar no arq.";
        try {
            out = new
FileWriter("c:\\temp\\arqCha
//caminho absoluto arq
            while
(contLetra <
texto.length()){

out.write(texto.charAt(contL
escreve caractere a
caractere

contLetra++;
            }

out.close(); // fecha
arquivo de saída
        }catch
(FileNotFoundException
```

```
45  e) {
46
47  e.printStackTrace();
48          } catch
49  (IOException e) {
50
51  e.printStackTrace();
52          }
53      }
54      public static void
55  LerCaracteres() {
56          FileReader in =
57  null;
58          try {
59              in = new
60  FileReader("c:\\temp\\arqChar
61              int c;
62              while ((c =
63  in.read()) != -1) //
64  escreve caractere a
65  caractere; -1 = EOF
66
67  System.out.print((char)c);//
68  imprime como caractere
69              in.close();
70  // fecha arquivo de
71  entrada
72          } catch
73  (FileNotFoundException
74  e) {
75
76  e.printStackTrace();
77          } catch
78  (IOException e) {
79
80  e.printStackTrace();
81          }
82      }
83      public static void
84  CopiarLinha(){
85          BufferedReader
86  in  = null; //
87  bufferiza (acumula)
88  para leitura cjto de
89  chars
```

```
90          PrintWriter
91   out = null; //
92   bufferiza (acumula)
93   para escrita cjto de
94   chars
95          String linha;
96          try {//
97   BufferedReader /
98   PrintWriter  são usados
99   em conjunto com
100               //
101  FileReader e
102  FileWriter,
103  respectivamente
104  (acumulam caracteres
105  lidos)
106             in   = new
107  BufferedReader(new
108  FileReader("c:\\temp\\arqCha
109             out  = new
110  PrintWriter   (new
111  FileWriter("c:\\temp\\arqCha
112             while
113  ((linha =
114  in.readLine()) != null)
115  {
116
117  out.println(linha);
118
119  System.out.println(linha);
120             }
121             in.close();
122
123  out.close();
124         } catch
125  (FileNotFoundException
126  e){
127
128  e.printStackTrace();
129         } catch
130  (IOException e){
131
132  e.printStackTrace();
133         }
134      }
```

```
135         public static void
136     main(String[] args) {
137
138     System.out.println("\nEscrita
139     --------------");
140
141     EscreverCaracteres();
142
143     System.out.println("\nLeitura
144     --------------");
145
146     LerCaracteres();
147
148     System.out.println("\nCopia-
149     --------------");
150             CopiarLinha();
151         }
152     }
153
154
155
156
157
158
159
160
161
162
163
164
```



Fonte: Autores (2021).

Persistência de objeto com controle de versão

Assim como conseguimos gravar e recuperar **linhas inteiras de arquivos-texto**, também conseguimos gravar e recuperar **objetos**.

Para isso, precisamos primeiro transformar a estrutura do objeto em uma **sequência binária.** Fazemos isso **serializando** o **estado do objeto** (seus atributos) em uma ***stream*** (cadeia de *bytes*), que pode então ser **gravado em arquivo binário** ou **transportado por**

meio de uma rede.

E, para poder ser **serializada**, uma classe precisa **implementar a interface** `java.io.Serializable` ou ser **herdeira de uma classe que a implemente**.

> **Classes** que não implementam a interface `java.io.Serializable`, não podem **serializar**/**desserializar** seu **estado**. Ou seja, **seus objetos não podem ser persistidos**
>
> ou **transferidos pela rede**.

Controle de versão do objeto serializado

Durante a serialização, o Java Runtime (ambiente de execução Java) associa um **identificador**, que é um **número de versão**, a cada **classe serializável**.

Esse número, denominado `serialVersionUID`, é usado durante a **desserialização**, para verificar se o **remetente** e o **receptor** de um **objeto serializado** carregaram **classes compatíveis desse objeto**.

Para melhor observar esses conceitos, vamos praticar a persistência de objetos, no próximo exercício.

EXERCÍCIO

Gravando e recuperando objetos

Lendo e escrevendo objetos

Usamos a classe `ObjectOutputStream` para **escrever objetos** e a classe `ObjectInputStream` para **ler objetos**.

Analise o código do exemplo a seguir, que **escreve e lê objetos em arquivo**. Observe que a classe dos objetos que são salvos e recuperados **implementa** a interface `java.io.Serializable`.

Veja se você consegue executar esse exemplo, obtendo a tela de console apresentada.

Responda:

1. Altere a linha 2 da classe **pessoa**, retirando a implementação da interface `Serializable`. O que acontece?
2. Qual a classe do objeto gravado?
3. Quantas e quais instâncias do objeto foram gravadas?
4. O que aconteceria se tentássemos ler objetos de um arquivo inexistente?
5. O que acontece se tentarmos acessar arquivos (ler, escrever, fechar) sem utilizar a estrutura *try-catch*?
6. Altere o código para gravar mais dois objetos (escolha os valores para o estado de cada novo objeto).

Figura 4 – Programa para gravar e recuperar objetos

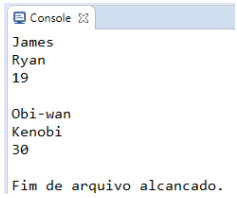
```
</>
```

```
import java.io.Serializable;
public class Pessoa implements
Serializable { // Classe comum, mas
serializável!

    // Adiciona um ID de versão serial
padrão a classe.
    private static final long
serialVersionUID = 1L;

    private String nome;
    private String sobrenome;
    private int idade;

    public String getNome() {
        return nome;
    }
    public void setNome(String firstName)
{
        this.nome = firstName;
    }
    public String getSobrenome() {
        return sobrenome;
    }
    public void setSobrenome(String
lastName) {
        this.sobrenome = lastName;
    }
    public int getIdade() {
        return idade;
    }
    public void setIdade(int age) {
        this.idade = age;
    }
    public String toString() {
           // Agrupa o valor dos atributos
em uma String
        StringBuilder builder = new
StringBuilder();
        builder.append(nome);
        builder.append("\n");
        builder.append(sobrenome);
        builder.append("\n");
        builder.append(idade);

```

```
            builder.append("\n");
            return builder.toString();
        }
    }

```

</>

```
1   import
2   java.io.EOFException;
3   import
4   java.io.FileInputStream;
5   import
6   java.io.FileNotFoundExceptio
7   import
8   java.io.FileOutputStream;
9   import
10  java.io.IOException;
11  import
12  java.io.ObjectInputStream;
13  import
14  java.io.ObjectOutputStream;
15
16  public class Objetos {
17
18      public static void
19  escrevePessoas(String
20  filename) {
21
22  ObjectOutputStream
23  outputStream = null;
24          try {
25
26  outputStream = new
27  ObjectOutputStream (new
28  FileOutputStream(filename));
29
30              Pessoa
31  person = new Pessoa();
32
33  person.setNome("James");
34
35  person.setSobrenome("Ryan");
36
37  person.setIdade(19);
38
39  outputStream.writeObject(per
40  //só escreve pq Pessoa
41  é seralizável
42
43              person =
44  new Pessoa();
```



```
45
46  person.setNome("Obi-
47  wan");
48
49  person.setSobrenome("Kenobi"
50
51  person.setIdade(30);
52
53  outputStream.writeObject(pers
54  //só escreve pq Pessoa
55  é seralizável
56
57
58  outputStream.flush();
59  // força dados em
60  buffer a serem gravados
61
62  outputStream.close();
63  // fecha arquivo de
64  escrita
65
66          } catch
67  (FileNotFoundException
68  ex) {
69
70  ex.printStackTrace();
71          } catch
72  (IOException ex) {
73
74  ex.printStackTrace();
75          }
76      }
77
78      public static void
79  lePessoas(String
80  filename) {
81
82  ObjectInputStream
83  inputStream = null;
84       try {
85          inputStream =
86  new ObjectInputStream
87  (new
88  FileInputStream(filename));
89          Object obj =
```

```
90  null;
91          while ((obj =
92  inputStream.readObject())
93  != null) {
94              if (obj
95  instanceof Pessoa) //
96  le um objeto genérico
97
98  System.out.println(((Pessoa)(
99  // cast para Pessoa
100         }
101
102 inputStream.close();
103      } catch
104 (EOFException ex) { //
105 quando EOF (End Of
106 File) é alçancado
107
108 System.out.println("Fim
109 de arquivo
110 alcançado.");
111      } catch
112 (ClassNotFoundException
113 ex) {
114
115 ex.printStackTrace();
116      } catch
117 (FileNotFoundException
118 ex) {
119
120 ex.printStackTrace();
121      } catch
122 (IOException ex) {
123
124 ex.printStackTrace();
125      }
126  }
127     public static void
128 main(String[] args) {
129
130 escrevePessoas("c:\\temp\\arq
131         lePessoas
132 ("c:\\temp\\arqObjs.txt");
133
134
```

```
135         }
136     }
137
138
```

Resolução do exercício

1. Ao retirar a implementação da interface serializable:

Na compilação do código, é gerado o erro: `java.io.NotSerializableException: Objetos.Pessoa`, pois não é possível gravar um objeto que não pode ser serializado.

2. Qual a classe do objeto gravado?

Resp.: classe **Person**.

3. Quantas e quais instâncias do objeto foram gravadas?

Resp.: duas instâncias, a saber: ("James Ryan“, age=19) e ("Obi-wan Kenobi“, age=30).

4. O que aconteceria se tentarmos ler objetos de um arquivo inexistente?

Resp.: gera exceção FileNotFoundException

5. O que acontece se tentarmos acessar arquivos (ler, escrever, fechar) sem utilizar a estrutura ***try-catch***?

Resp.: gera erro de compilação, pois manipulação de arquivo exige tratamento de ***try-catch***.

6. Altere o código para gravar mais dois objetos (escolha os valores para o estado de cada novo objeto).

Resp.: acrescentar as linhas Java:

```
Pessoa novaPerson = new Pessoa();

        novaPerson.setNome("Harry");
```

```
novaPerson.setSobrenome("Potter");

novaPerson.setIdade(11);

novaPerson.writeObject(novaPerson);


novaPerson.setNome("Ron");

novaPerson.setSobrenome("Weasley");

novaPerson.setIdade(11);

novaPerson.writeObject(novaPerson);
```

Fonte: Autores (2021).

Interface gráfica: interação por eventos

Com as interfaces gráficas, a interação do usuário com os nossos programas acontece por meio de **disparos de eventos**, como já comentado. Por exemplo, o acionamento de um botão é um **evento** que provoca a execução do código, como o que exibe algumas informações na tela.

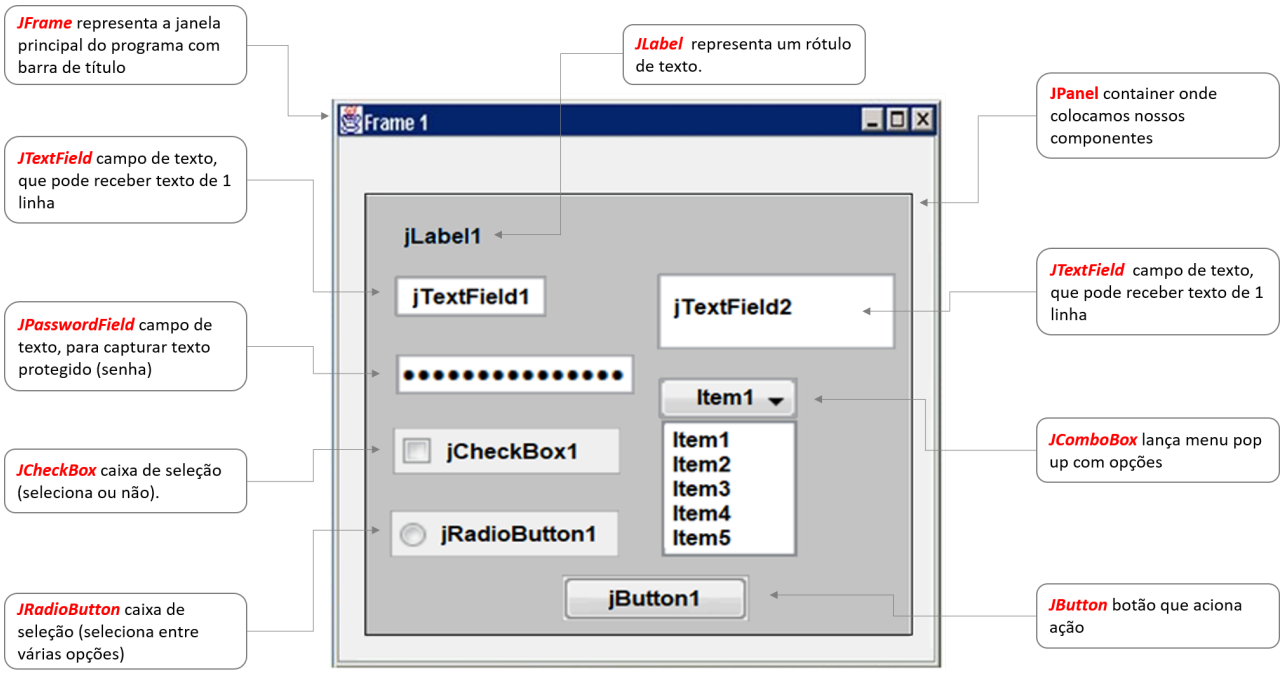
Na prática, existem processo implícitos nos nossos programas gráficos que se encarregam de capturar determinados eventos e executar os códigos de programamos, que são as respostas para estes eventos.

> A **OCORRÊNCIA DE UM EVENTO** pode provocar uma **reação** que pode ser uma **ação** (ou um conjunto delas) a ser tomada – de acordo com o que nós programadores definimos.

Para criar **nossa interface gráfica**, usaremos algumas funcionalidades do **Java Swing** que, como tudo no Java, é independente de plataforma. Isso significa que, com o Swing, nossos programas têm uma aparência muito parecida, independentemente do

sistema operacional, ou plataforma, em que está sendo utilizado. Sua biblioteca está no pacote `javax.swing`, que provê classes para a API Java Swing API, como `JButton`, `JTextField`, `JTextArea`, `JRadioButton`, `JCheckbox`, `JMenu`, `JColorChooser` etc. Veja como eles são utilizados na Figura 5 a seguir.

Figura 5 – Alguns componentes Java Swing



Fonte: ORACLE, Documentation (2020).

O **Swing** é parte do Java Foundation Classes (**JFC**) (https://www.oracle.com/java/technologies/java-foundation-classes.html), um conjunto abrangente de componentes e serviços de *Graphical User Interface* (**GUI**), ou interface gráfica do usuário, que simplificam bastante o desenvolvimento de aplicações gráficas para *desktop*, principalmente com apoio de um IDE. O IDE **NetBeans** (https://netbeans.org/) fornece vários recursos para trabalhar com aplicações gráficas com Swing. Mais detalhes podem ser encontrados em: Projetando uma GUI Swing no NetBeans IDE.

Iremos explorar algumas funcionalidades do Swing que vão nos permitir criar uma aplicação simples, mas que já permite interagir com o usuário via pequenas janelas, caixas de texto e botões.

Porém, antes, precisamos conhecer a classe *JOptionPane*. Vamos fazer isso na próxima prática, com o exercício 4, a seguir.

EXERCÍCIO

Interface gráfica

A classe `javax.swing.JOptionPane` é usada para fornecer **caixas de diálogo padrão** (pequenas janelas de pop-up), como as obtidas com os métodos estáticos:

1. `JOptionPane.showMessageDialog(componentePai,"mensagem")` – cria uma caixa de diálogo para exibir uma mensagem.
2. `JOptionPane. showInputDialog(componentePai,"mensagem")` – cria uma caixa de diálogo que também solicita um valor de **entrada**, digitada pelo usuário.

Em uma aplicação gráfica mais completa, é usual criar um objeto de `javax.swing.JFame` para ser usado como **`componentePai`** para as caixas de diálogo da `JOptionPane`.

No exemplo da nossa aplicação simples, usamos **`null`** como **`componentePai`**. Isso significa que as **nossas caixas de diálogo** são "**órfãs**", ou não estão vinculadas a nenhuma outra janela.

Verifique se você consegue obter a mesma sequência de caixas de diálogo do exercício.

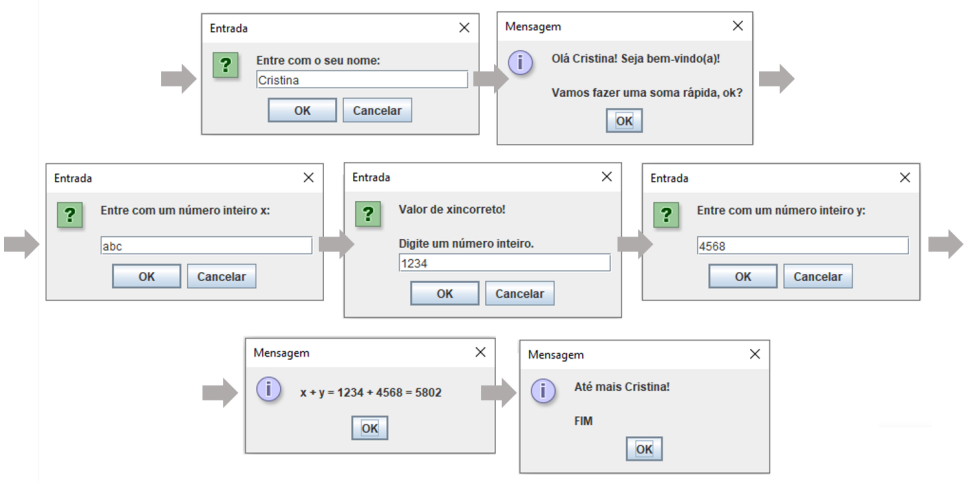
Figura 6 – Programa simples com interface gráfica

```
</>
```

```
20
21  import javax.swing.JOptionPane;  //
22  Biblioteca da classe JOptionPane
23
24  public class MinhaInterface {
25      private boolean
26  numeroInteiroValido(String s) {
27          boolean resultado;
28          try {
29              Integer.parseInt(s); // Tenta
30  transformar uma string em inteiro
31              resultado = true;
32          } catch (NumberFormatException e)
33  { // Gera erro se não consegue
34              resultado = false;
35          }
36          return resultado;
37      }
38      public int inteiroValido(String
39  nomeVar) { // retorna um valor inteiro
40          int numInt;
41          String entrada;
42
43          entrada =
44  JOptionPane.showInputDialog (null, "Entre
45  com um número inteiro " + nomeVar +
46  ":\n\n");
47
48          // Loop para garantir um inteiro
49  válido
50          while
51  (!this.numeroInteiroValido(entrada)) {
52                  entrada =
53  JOptionPane.showInputDialog(null, "Valor
54  de " + nomeVar + "incorreto!\n\nDigite um
55  número inteiro.");
56          }
57          return new Integer(entrada);
58      }
59      public static void main(String[] args)
60  {
61          MinhaInterface painel = new
62  MinhaInterface();
63          String nome, entrada;
64          int x, y;
```

```
65
66          nome = JOptionPane.showInputDialog
67  ("Entre com o seu nome: ");
68
69  JOptionPane.showMessageDialog(null,"Olá "
70  + nome + "! Bem-vindo(a)!");
71
72  JOptionPane.showMessageDialog(null,"Vamos
73  fazer uma soma rápida, ok?");
74          x = painel.inteiroValido("x");
75          y = painel.inteiroValido("y");
76
77  JOptionPane.showMessageDialog(null," x + y
78  = " + x+" + "+ y+ " = " + (x+y));
79
80  JOptionPane.showMessageDialog(null,"Até
81  mais " + nome + "! \n\nFIM");
82      }
83  }
84
85
```

Sequência de apresentação das caixas de diálogo do programa

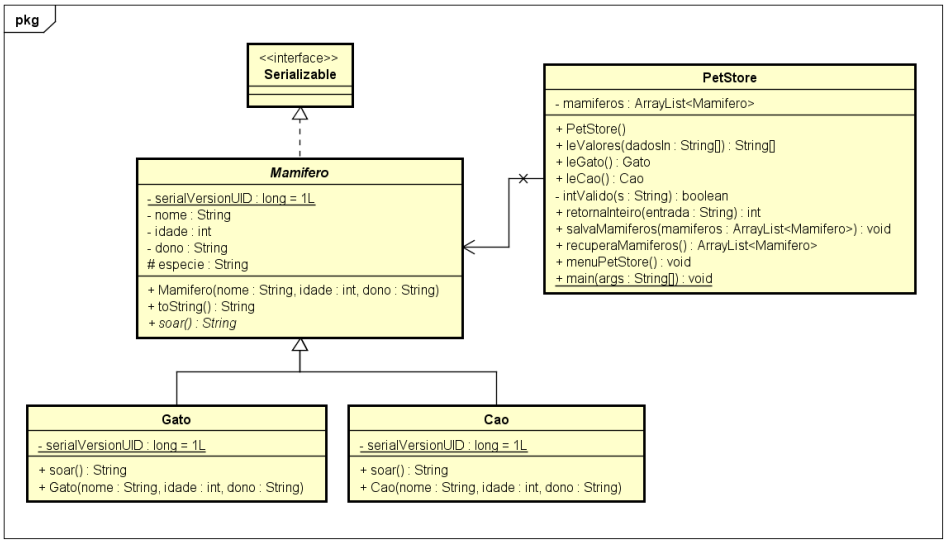


Fonte: Autores (2021).

EXERCÍCIO

Persistindo objetos com interface gráfica

Implemente as classes representadas no diagrama a seguir, conforme as definições expostas.



Para executar esse programa, você deve:

1. Observar as relações de **herança** (Java ***`extends`***) e **realização** (Java ***`implements`***) do diagrama UML para completar os códigos das classes **cao**, **gato** e **mamifero** – procurar e trocar no código os seguintes comentários:`//complete o código aqui`.
2. Após completar os códigos e conseguir executar, acrescente mais as classes **cavalo** e **leão**, que fazem **relinchos** e **rugidos**, respectivamente. Altere os códigos para que você consiga incluir todos esses **quatro tipos** de mamíferos.

Figura 7 – Exercício com interface gráfica, hierarquia, polimorfismo e persistência de objetos

```
</>
```

```
1
2
3
4
5
6
7
8
9
10
11
12
13
```

```
14
15  public class Cao /*complete o código aqui
16  */ {
17
18      private static final long
19  serialVersionUID = 1L;
20
21      public String soar() {
22          return “Faz latidos”;
23      }
24      //complete o código aqui
25      //complete o código aqui
26      }
27  }
28
```

</>

```
1   public class Gato /*complete o código aqui
2   */ {
3       //complete o código aqui
4       // ...
5       //complete o código aqui
6   }
7
```

</>

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
```

```
//complete o código aqui

public abstract class Mamifero /* complete o código aqui */ {

    private static final long serialVersionUID = 1L;
    //complete o código aqui
    //complete o código aqui
    //complete o código aqui
    //complete o código aqui

    public Mamifero(String nome, int idade, String dono) {
        this.nome = nome;
        this.idade = idade;
        this.dono = dono;
    }
    public String toString() {
        String retorno = “”;
        retorno += “Nome: ” + this.nome + “\n”;
        retorno += “Idade: ”    + this.idade    + “ anos\n”;
        retorno += “Dono: ”     + this.dono     + “\n”;
        retorno += “Espécie: ”  + this.especie  + “\n”;
        retorno += “Barulho: ”  + soar() + “\n”;
        return retorno;
    }
    //complete o código aqui
}
```

</>

```
```

```
7
8   import java.io.EOFException;
9   import java.io.FileInputStream;
10  import java.io.FileNotFoundException;
11  import java.io.FileOutputStream;
12  import java.io.IOException;
13  import java.io.ObjectInputStream;
14  import java.io.ObjectOutputStream;
15  import java.util.ArrayList;
16
17  import javax.swing.JoptionPane;
18
19  public class PetStore {
20      private ArrayList<Mamifero> mamiferos;
21
22      public PetStore() {
23          this.mamiferos = new
24  ArrayList<Mamifero>();
25      }
26      public String[] leValores (String []
27  dadosIn){
28          String [] dadosOut = new String
29  [dadosIn.length];
30
31          for (int i = 0; i <
32  dadosIn.length; i++)
33             dadosOut[i] =
34  JoptionPane.showInputDialog("Entre com "+
35  dadosIn[i] +": ");
36
37          return dadosOut;
38      }
39      public Gato leGato(){
40
41          String [] valores = new String
42  [3];
43          String [] nomeVal = {"Nome",
44  "Idade", "Dono"};
45          valores = leValores (nomeVal);
46
47          int idade =
48  this.retornaInteiro(valores[1]);
49
50          Gato = new Gato (valores[0],
51  idade, valores[2]);
```

```
52          return gato;
53      }
54
55      public Cao leCao(){
56          String [] valores = new String
57  [3];
58          String [] nomeVal = {“Nome”,
59  “Idade”, “Dono”};
60          valores = leValores (nomeVal);
61
62          int idade =
63  this.retornaInteiro(valores[1]);
64
65          Cao = new Cao (valores[0], idade,
66  valores[2]);
67          return cao;
68      }
69      private int intValido(String s) {
70          try {
71              Integer.parseInt(s); // Tenta
72  transformar uma string em inteiro
73              return true;
74          } catch (NumberFormatException e)
75  { // Se não consegue transformar, erro
76              return false;
77          }
78      }
79      public int retornaInteiro(String
80  entrada) { // retorna um valor inteiro
81          int numInt;
82
83          //Tenta converter o valor de
84  entrada para inteiro, senão permanece no
85  loop
86          while (!this.intValido(entrada)) {
87              entrada =
88  JoptionPane.showInputDialog(null, “Valor
89  incorreto!\n\nDigite um número inteiro.”);
90          }
91          return Integer.parseInt(entrada);
92      }
93
94      public void salvaMamiferos
95  (ArrayList<Mamifero>  amíferos){
96          ObjectOutputStream outputStream =
```

```
97  null;
98          try {
99              outputStream = new
100 ObjectOutputStream
101                     (new
102 FileOutputStream("c:\\temp\\petStore.dados"));
103             for (int i=0; i <
104 mamiferos.size(); i++)
105
106 outputStream.writeObject(mamiferos.get(i));
107         } catch (FileNotFoundException ex)
108 {
109
110 JoptionPane.showMessageDialog(null,
111 "Impossível criar arquivo!");
112             ex.printStackTrace();
113         } catch (IOException ex) {
114             ex.printStackTrace();
115         } finally {  // Fecha arquivo
116 ObjectOutputStream
117             try {
118                 if (outputStream != null)
119 {
120                     outputStream.flush();
121                     outputStream.close();
122                 }
123             } catch (IOException ex) {
124                 ex.printStackTrace();
125             }
126         }
127     }
128     @SuppressWarnings("finally")
129     public ArrayList<Mamifero>
130 recuperaMamiferos (){
131         ArrayList<Mamifero> mamiferosTemp
132 = new ArrayList<Mamifero>();
133
134         ObjectInputStream inputStream =
135 null;
136
137         try {
138             inputStream = new
139 ObjectInputStream
140                     (new
141 FileInputStream("c:\\temp\\petStore.dados"));
142
```

```
            Object obj = null;
            while ((obj =
inputStream.readObject()) != null) {
                if (obj instanceof
Mamifero) {

mamiferosTemp.add((Mamifero) obj);
                }
            }
        } catch (EOFException ex) { //
quando EOF é alcançado
            System.out.println("Fim de
arquivo.");
        } catch (ClassNotFoundException
ex) {
            ex.printStackTrace();
        } catch (FileNotFoundException ex)
{

JoptionPane.showMessageDialog(null,
"Arquivo com mamíferos NÃO existe!");
            ex.printStackTrace();
        } catch (IOException ex) {
            ex.printStackTrace();
        } finally {  // Fecha arquivo
ObjectInputStream
            try {
                if (inputStream != null) {
                    inputStream.close();
                }
            } catch (final IOException ex)
{
                ex.printStackTrace();
            }
            return mamiferosTemp;
        }
    }
    public void menuPetStore (){
        String menu = "";
        String entrada;
        int    opc1, opc2;
        do {
            menu = "Controle PetStore\n" +
                    "Opções:\n" +
                    "1. Entrar
```

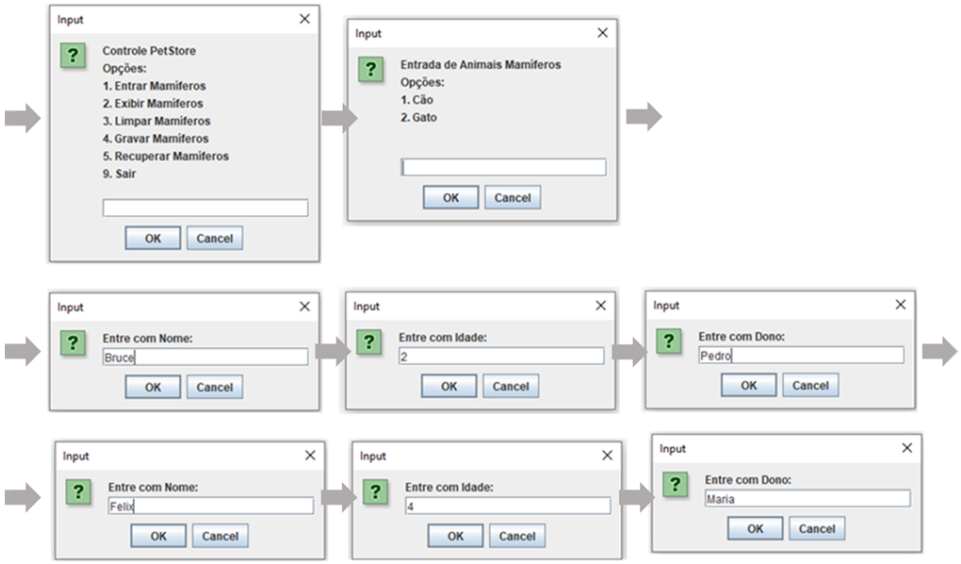
```
187
188     Mamíferos\n” +
189                         “2. Exibir
190     Mamíferos\n” +
191                         “3. Limpar
192     Mamíferos\n” +
193                         “4. Gravar
194     Mamíferos\n” +
195                         “5. Recuperar
        Mamíferos\n” +
                            “9. Sair”;
                entrada =
        JoptionPane.showInputDialog (menu +
        “\n\n”);
                opc1 =
        this.retornaInteiro(entrada);

                switch (opc1) {
                case 1:// Entrar dados
                    menu = “Entrada de Animais
        Mamíferos\n” +
                                “Opções:\n” +
                                “1. Cão\n” +
                                “2. Gato\n”;

                    entrada =
        JoptionPane.showInputDialog (menu +
        “\n\n”);
                    opc2 =
        this.retornaInteiro(entrada);

                    switch (opc2){
                    case 1:
        mamiferos.add((Mamifero)leCao());
                    break;
                    case 2:
        mamiferos.add((Mamifero) amífe());
                    break;
                    default:

        JoptionPane.showMessageDialog(null,
        “Entrada NÃO válida!”);
                    }

                    break;
                case 2: // Exibir dados
```

```
                if (mamiferos.size() == 0)
{

JoptionPane.showMessageDialog(null, “Entre
com mamíferos ...”);
                    break;
                }
                String dados = “”;
                for (int i=0; i <
mamiferos.size(); i++)    {
                    dados +=
mamiferos.get(i).toString() + “----------
----\n”;
                }

JoptionPane.showMessageDialog(null,dados);
                break;
            case 3: // Limpar Dados
                if ( mamiferos.size() ==
0) {

JoptionPane.showMessageDialog(null, “Entre
com mamíferos ...”);
                    break;
                }
                 mamiferos.clear();

JoptionPane.showMessageDialog(null, “Dados
LIMPOS com sucesso!”);
                break;
            case 4: // Grava Dados
                if (mamiferos.size() == 0)
{

JoptionPane.showMessageDialog(null, “Entre
com mamíferos ... ”);
                    break;
                }
                salvaMamiferos(mamiferos);

JoptionPane.showMessageDialog(null, “Dados
SALVOS com sucesso!”);
                break;
            case 5: // Recupera Dados
                 mamiferos =
```

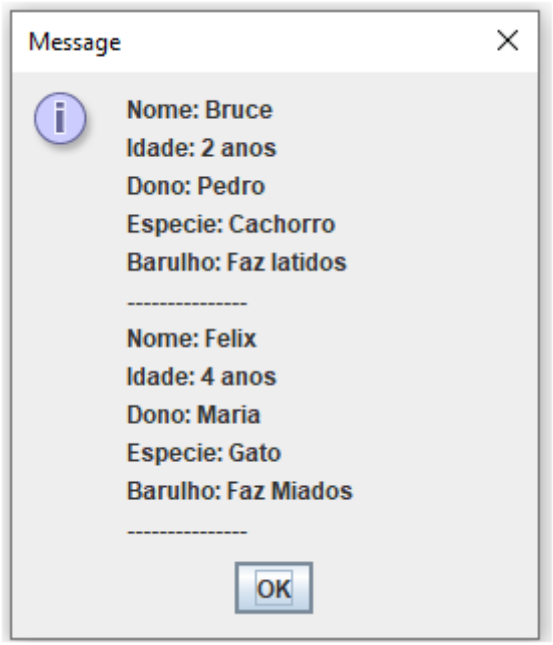
```
recuperaMamiferos();
                if ( mamiferos.size() ==
0) {

JoptionPane.showMessageDialog(null,“Sem
dados para apresentar.”);
                    break;
                }

JoptionPane.showMessageDialog(null,“Dados
RECUPERADOS com sucesso!”);
                break;
            case 9:

JoptionPane.showMessageDialog(null, “Fim
do aplicativo PETSTORE”);
                break;
            }
        } while (opc1 != 9);
    }
    public static void main (String []
args){
        PetStore pet = new PetStore ();
        pet.menuPetStore();
    }
}
```

Sequência de apresentação das caixas de diálogo do programa

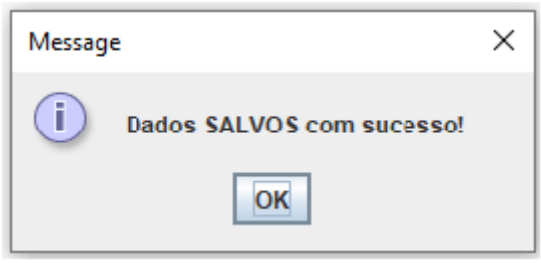
1. Entrar os dados de um cão e de um gato:



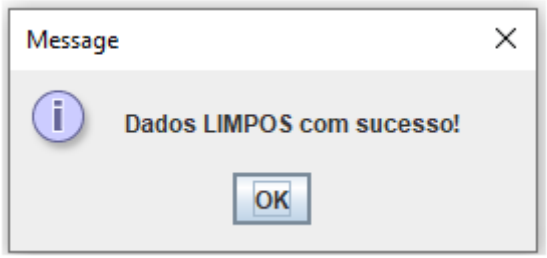
2. Exibir a lista de mamíferos:



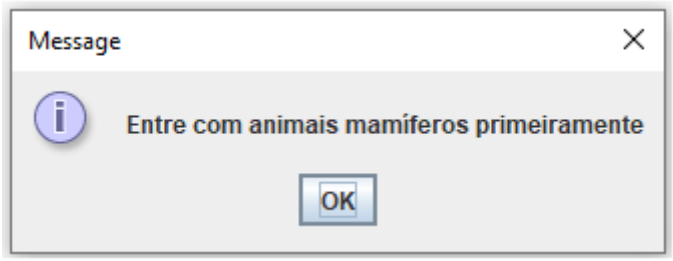
3. Gravar os objetos criados em arquivo:



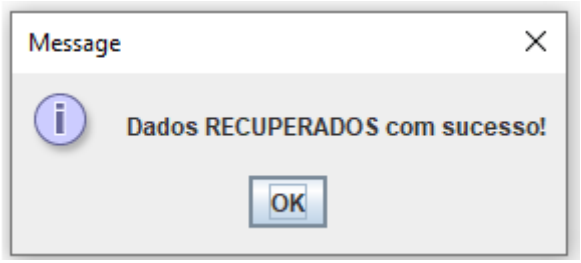
4. Limpar da memória (*ArrayList*) os objetos recém-criados:



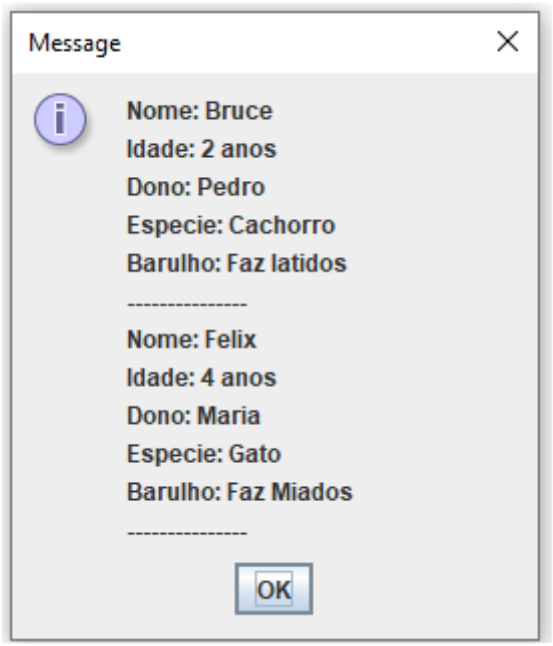
5. Exibir novamente a lista de mamíferos:



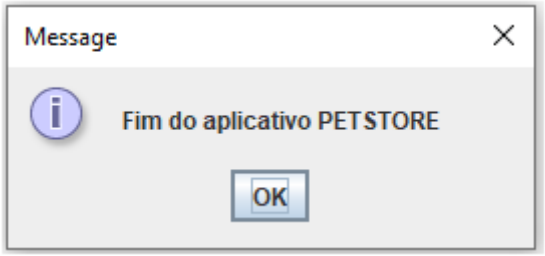
6. Recuperar os objetos persistidos:



7. Exibir novamente a lista de mamíferos:



8. Sair do programa:



Resolução do exercício

Resposta questão 1:

```
</>
1   import java.io.Serializable;
2
3   public abstract class Mamifero implements
4   Serializable {
5        private   String nome;
6       private   int idade;
7       private   String dono;
8       protected String especie;
9   ...
10  public abstract String soar();
11  }
12  ------------------------------
13  public class Cao extends Mamifero {
14
15      private static final long
16  serialVersionUID = 1L;
17
18      public String soar() {
19          return "Faz latidos";
20      }
21      public Cao(String nome, int idade,
22  String dono) {
23          super(nome, idade, dono);
24          this.especie = "Cachorro";
25          }
26  }
27
```

Fonte: Autores (2021).

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.

ORACLE. The Java™ Tutorials. **ORACLE**, Documentation, 2020. Disponível em: https://docs.oracle.com/javase/tutorial/uiswing/components/componentlist.html. Acesso em: 20 fev. 2021.