



Desenvolvimento de Sistemas

Estrutura de dados: sintaxe, bibliotecas de linguagem e aplicabilidade

Introdução

Você aprendeu anteriormente no curso a usar estruturas de dados a partir de classes implementadas na biblioteca padrão de Java, entre elas pilhas (classe **Stack**), filas (classes **Queue** e **LinkedList**), listas de dados (interface **List** e classes **ArrayList**, **LinkedList**, **Vector** e outras) e coleções sem duplicatas (interface **Set** e classes **HashSet** e **TreeSet**, por exemplo). Neste conteúdo, você concentrará sua atenção na estrutura mais usual, que são as listas (**List<E>**), a forma mais prática e dinâmica de trabalhar com uma coleção de objetos integrando essa estrutura de dados a elementos de interface gráfica, usando tabela (**JTable**), que é um componente da Swing utilizado para exibir listas de dados na tela da sua aplicação.

Sintaxe

Comece seu estudo das listas em Java revisando um conceito trabalhado no início do curso. Lembre-se dos estudos relativos a vetores? Os vetores são estruturas de dados que armazenam uma quantidade fixa de um certo tipo de dados.

Imagine que precise armazenar dez números inteiros. Nessa situação, você tem uma quantidade fixa de dados para ser armazenada e um simples vetor resolve o problema. Confira a seguir:



```
public static void main(String[] args) {  
    // TODO code application logic here  
  
    //Criação do vetor de inteiros vetInt  
    int[] vetInt = new int[10];  
}
```

Mas, e se você não souber a quantidade de elementos que precisa armazenar em uma estrutura?

Imagine que você precisa trabalhar com uma lista com todos os nomes dos alunos de uma escola. Nesse caso em que não se sabe de antemão o número de elementos a armazenar, é possível usar uma estrutura chamada **List**.

Em Java, há a interface **List<E>**, que define operações comuns a listas em geral. Podem-se usar as classes concretas que implementam essa interface. As principais delas são:

- ◆ **ArrayList<E>**: uma espécie de vetor sem limitação de tamanho
- ◆ **Vector<E>**: uma versão “*thread-safe*” (ou seja, à prova de problemas em processamentos paralelos) de **ArrayList**
- ◆ **LinkedList<E>**: uma implementação de Java para listas ligadas, recomendada para situações em que importa mais a eficiência na inclusão e na exclusão de valores do que na busca

Note a presença da notação de *generics*, **<E>**, que define o tipo de elemento encontrado nessa lista (ou seja, os elementos da estrutura de dados serão do tipo informado no lugar de “E”). É possível utilizar os tipos primitivos e os tipos customizados, ou seja, os objetos que são as abstrações do mundo real.

Se você quiser uma lista de *strings*, utilize dessa maneira:



```
public static void main(String[] args) {  
    // TODO code application logic here  
  
    /*  
     * Assim definimos a List que vai conter String, pelo nome dado  
     * a lista (namesList), podemos concluir que cada elemento  
     * da lista será um nome  
     */  
    List<String> namesList;  
}
```

Geralmente, a sintaxe para instanciación de uma lista fica como o exemplo a seguir:

```
public static void main(String[] args) {  
    // TODO code application logic here  
  
    //instanciação de tipo ArrayList que conterá elementos float.  
    ArrayList<Float> listaFloats = new ArrayList<>();  
  
    //criando lista de inteiros. Preferível usar o tipo mais genérico List  
    //que o ArrayList como tipo de referência do objeto.  
    List<Integer> listaInteiros = new ArrayList<>();  
  
    //instanciando um tipo Vector a uma referência genérica List  
    List<String> listaTextos = new Vector<String>();  
}
```

Confira um exemplo de criação de um objeto **ArrayList**:



```
public static void main(String[] args) {  
    /*  
    * Uma vez definida a Lista, precisamos criar uma instância  
    * dessa lista para podermos utilizá-la. Para isso  
    * instanciamos um novo ArrayList;  
    * Perceba que não precisamos informar o tamanho.  
    */  
    List<String> namesList = new ArrayList<>();  
}
```

Perceba que, ao contrário dos vetores, não é preciso informar o tamanho, isso porque as listas são dinâmicas e pode-se, a qualquer instante, adicionar ou remover elementos da lista.

Atenção: ao informar o *import* para *list*, opte sempre por **java.util.List** e não **java.awt.List**, frequentemente sugerido pelo NetBeans.

A seguir, veja os principais métodos de **List<E>** (lembrando que “E” é o tipo informado na instanciação do objeto).



| Tipo de retorno | Método | Descrição |
|-----------------|----------------------------------|--|
| boolean | add(E e) | Insere o elemento especificado ao final da lista. |
| void | clear() | Remove todos os elementos da lista. |
| boolean | contains(Object o) | Retorna <i>true</i> se o elemento for encontrado na lista. |
| boolean | equals(Object o) | Retorna <i>true</i> se o elemento especificado for igual ao objeto encontrado na lista. |
| E | get(int index) | Retorna o elemento da posição especificada na lista. |
| Int | indexOf(Object o) | Retorna o índice da primeira ocorrência do elemento especificado. |
| boolean | isEmpty() | Retorna <i>true</i> se a lista não contém elementos. |
| E | remove(int index) | Remove o elemento da posição especificada da lista. |
| boolean | remove(Object o) | Remove a primeira ocorrência do objeto especificado da lista. |
| E | set(int index, E element) | Substitui o elemento na posição especificada pelo elemento especificado. |
| Int | size() | Retorna o número de elementos dessa lista, ou seja, a quantidade de elementos contidos na lista. |

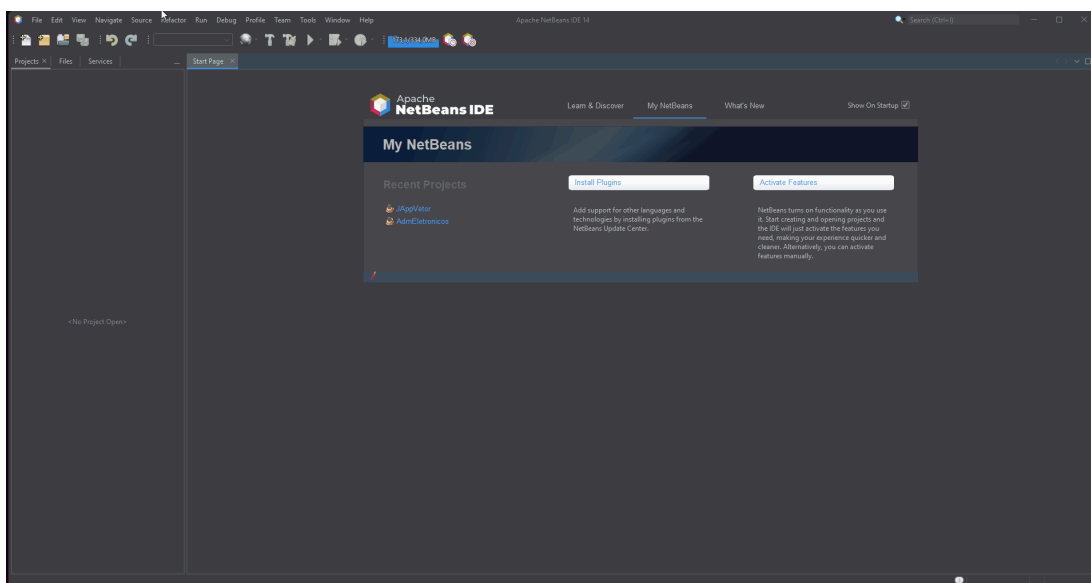
Uma vez que são conhecidos os principais métodos da *list*, você os codificará. A partir de agora, tenha muita atenção aos códigos. Eles estarão comentados para que você entenda toda a estrutura de declaração e utilização.

Como exemplo, será criado um programa que permita inclusão, remoção, localização e listagem de nomes pessoais. Para isso, será desenvolvido um menu de opções com as seguintes opções:

1. Inserir nome
2. Verificar se o nome está lista
3. Deletar nome
4. Listar todos os nomes
5. Sair

Efetue os seguintes passos:

- ◆ Crie um novo projeto chamado **JAppExampleList**:



- ◆ Implemente o código a seguir:



```
        /*
        * 1o. passo: definir a lista que vamos armazenar os nomes
        */
        List<String> namesList = new ArrayList<>();

        /*
        * 2o. passo: criar a variável que controla a parada da nossa aplicação.
        * A opção que vai ser digitada pelo usuário
        */
        int option = 0;

        /*
        * Criação do objeto "keyboard" que vai capturar tudo o que for digitado
        */
        Scanner keyboard = new Scanner(System.in);

        /*
        * 4o. passo: criar a estrutura de repetição que vai manter nossa aplicação funcionando
        */
        do {

            /*
            * 5o. passo: Exibir as opções para o usuário poder escolher
            */

            System.out.println("\nTrabalhando com List\n");
            System.out.println("1. Inserir nome");
            System.out.println("2. Verificar nome");
            System.out.println("3. Remover nome");
            System.out.println("4. Listar todos os nomes");
            System.out.println("5. Sair\n");

            /*
            * 6o. passo: ler a opção digitada pelo usuário
            */
            System.out.print("Digite sua opção [1 a 5]: ");
            option = keyboard.nextInt();

            /*
            * 7o. passo: criar o seletor da opção digitada pelo usuário
            */
            switch (option) {
                case 1:
                    //Aqui vamos inserir o novo nome
                    System.out.println("\nINSERIR NOME\n");
                    /*
```

```

        * Para inserir o nome na lista vamos recorrer ao método "add(String name)" passando
        * como parâmetro a string digitada pelo usuário; Como esse método
        * retorna true "se inserido" e false "se não inserido". Vamos aproveitar
        * para emitir uma mensagem de inserido com sucesso ou não.
        * Para termos um certo padrão vamos deixar os nomes com todas as letras
        * maiúsculas usando o método toUpperCase().
        */
        System.out.print("Digite o nome: ");
        boolean itsInserted = namesList.add(keyboard.next().toUpperCase());

        if(itsInserted){
            //se itsInserted = true
            System.out.println("Nome inserido");
        }else{
            //se itsInserted = false
            System.out.println("FALHA! Nome não inserido.");
        }

        break;

    case 2:
        //Aqui vamos buscar um nome na lista
        System.out.println("\nVERIFICAR NOME\n");

        /*
        * Aqui vamos buscar por um nome na lista, mas é conveniente antes de fazer isso,
        * verificar se a lista está vazia para isso utilizamos o método isEmpty() que retorna
        * true se a lista está vazia e false se contem 1 ou mais elementos
        */
        if(namesList.isEmpty()){
            //se true exibe mensagem de lista vazia.
            System.out.println("ATENÇÃO! Não há nomes na lista.");
        }else{
            /*
            * se false, precisamos pedir para o usuário o nome a ser procurado na lista.
            * Podemos utilizar dois métodos: o contains(Object o) o equals(Object o).
            * Como nossa lista é uma lista de String nosso objeto é um string.
            * Como armazenamos os nomes com todas as letras em maiúsculo
            */

```



```
        * devemos procurar com as letras todos em maiusculo. Aqui vamos criar uma
        * variável para armazenar o nome a ser encontrado
        */
        System.out.print("Digite o nome a ser localizado: ");
        String name = keyboard.next().toUpperCase();

        if(namesList.contains(name)){
            /*
            * se true foi encontrado
            * podemos exibir o nome e a posição que ele foi encontrado
            * para isso utilizamos o método indexOf(Object o)
            * a posição que o nome foi encontrado. Lembre-se o nome
            * na variável name;
            */

            int position = namesList.indexOf(name);
            System.out.println("Nome: "+name+" | Posição: "+position+" da lista.");
        }else{
            //se false não foi encontrado
            System.out.println("ATENÇÃO! Nome não encontrado na lista.");
        }

        break;

    case 3:

        //Aqui vamos remover um nome na lista
        System.out.println("\nREMOVER NOME\n");

        /*
        * Aqui vamos buscar por um nome na lista, mas é conveniente antes de fazer isso,
        * verificar se a lista está vazia para isso utilizamos o método isEmpty() que retorna
        * true se a lista está vazia e false se contem 1 ou mais elementos
        */
        if(namesList.isEmpty()){
            //se true exibe mensagem de lista vazia.
            System.out.println("ATENÇÃO! Não há nomes na lista.");
        }else{
```



```

        /*
        * se false, precisamos pedir para o usuário o nome a ser re
movido da lista.
        * O metodo responsável por isso é o remove(Object o). Esse
método retorna
        * true quando remove o objeto, e false quando não remove.
        * Como nossa lista é uma lista de String nosso objeto é uma
string.
        * Como armazenamos os nomes com todas as letras em maiúscul
o
        * devemos procurar com as letras todos em maiusculo. Aqui v
amos criar uma
        * variável para armazenar o nome a ser encontrado
        */
        System.out.print("Digite o nome a ser deletado: ");
        String name = keyboard.next().toUpperCase();

        if(namesList.remove(name)){
            /*
            * se true foi deletado.
            * Podemos exibir o nome e a mensagem de removido com su
cesso.
            */

            System.out.println("Nome: "+name+" - Removido com suces
so.");

        }else{
            //se false não foi encontrado
            System.out.println("ATENÇÃO! Nome não encontrado na lis
ta.");
        }

    }

    break;

    case 4:
        //Aqui vamos listar todos os nomes da lista
        System.out.println("\nLISTAR TODOS OS NOMES\n");

        /*
        * Aqui vamos buscar por todos os nome na lista, mas é convenie
nte antes de fazer isso,
        * verificar se a lista está vazia para isso utilizamos o método
isEmpty() que retorna
        * true se a lista está vazia e false se contem 1 ou mais elemen
tos
        */
        if(namesList.isEmpty()){

```

```
//se true exibe mensagem de lista vazia.
System.out.println("ATENÇÃO! Não há nomes na lista.");
}else{
    /*
    * se false, vamos ter passar por todas a posições da lista
    criada, pegar
    * o objeto, no nosso caso, uma String e exibir em tela aqui
    vamos revisitar
    * o laço de repetição for, que vai parar quando chegarmos a
    o fim da lista.
    * para saber quando parar vamos usar o método size() que re
    torna o número
    * de elementos tem na lista.
    * Para pegar o elemento da lista utilizamos o método get(in
    t index), esse
    * metodo retorna o objeto da posição informada como parâmet
    ro.
    */

    //variável que vai receber o nome da lista para ser exibi
    do em tela

    String name = "";

    for (int i = 0; i < namesList.size(); i++) {
        name = namesList.get(i);
        System.out.println("Posição "+i+": "+name);
    }

    }

    break;

    case 5:
        System.out.println("OBRIGADO POR UTILIZAR NOSSO APLICATIVO - S
        ENAC EAD");
        break;

        default:
            //Caso o usuário digite algum valor diferente aos do menu vam
            os exibir essa mensagem
            System.out.println("ATENÇÃO! Opção inválida!");
            break;
    }

    } while (option != 5);

}
```



Teste sua aplicação! Se tudo foi implementado corretamente, sua aplicação está funcionando corretamente.

Que tal um desafio?

Adicione mais duas funcionalidades na sua aplicação: “Listar a quantidade de elementos da lista” e “Limpar lista”.



Aplicabilidade

Agora que você relembrou os conceitos e exercitou o uso das listas em Java, já pode aplicar essa estrutura de dados com elementos de interface gráfica Swing de Java. A maneira mais direta de interação entre esses elementos é por meio do componente **JTable**.

O **JTable** é um componente de interface gráfica que permite manipular tabelas de dados e que também permite que o usuário digite o valor de cada célula a fim de preencher/atualizar a tabela. É um componente visual

utilizado para exibir dados em forma de *grid*, com cabeçalho, colunas e linhas.



Por ser um componente destinado à exibição de tabelas, ele não realiza a persistência dos dados, logo, quando a interface gráfica é fechada, os dados exibidos na tabela são perdidos.

Por pertencer ao pacote **javax.swing**, esse componente é totalmente customizável e oferece métodos para acessar, inserir e excluir registros. Podem-se adicionar eventos e outras funções como edição de propriedades das células da tabela.

Considerar a utilização desse componente implica pensar na exibição de dados oriundos de um banco de dados, geralmente organizados em uma lista de objetos trabalhada no tópico anterior. Isso possibilita a criação de *listeners* para chamar métodos específicos em conformidade com o tipo de interação realizada pelo usuário com a tabela, sendo esse o caminho para manter os dados da sua *database* e do componente sempre atualizados.

A seguir, confira os principais métodos de **JTable**.



| Tipo retorno | Método | Descrição |
|--------------|--|--|
| void | clearSelection() | Desmarca todas as linhas e colunas selecionadas. |
| boolean | editCellAt(int row, int column) | Inicia programaticamente a edição da célula na <i>row</i> (linha) e na <i>column</i> (coluna), se a célula em questão for válida e editável. |
| TableModel | getModel() | Retorna o TableModel que fornece os dados exibidos pela JTable . |
| int | getRowCount() | Retorna a quantidade de linhas que podem ser exibidas no JTable dada a limitação de espaço. |
| int | getSelectedColumn() | Retorna o índice da primeira coluna selecionada. -1 se nenhuma coluna for selecionada. |
| int[] | getSelectedColumns() | Retorna o índice das colunas selecionadas. |
| int | getSelectedRow() | Retorna o índice da primeira linha selecionada. -1 se nenhuma linha for selecionada. |
| int[] | getSelectedRows() | Retorna o índice das linhas selecionadas. |



| Tipo retorno | Método | Descrição |
|--------------|---|---|
| Object o | <code>getValueAt(int row, int column)</code> | Retorna o valor da célula em <i>row</i> (linha) e <i>column</i> (coluna). |
| void | <code>setColumnSelectionAllowed(boolean value)</code> | Define se as linhas poderão ser selecionadas. |
| void | <code>setModel(Tablemodel model)</code> | Configura o modelo de dados para a JTable . |
| void | <code>setRowSelectionAllowed(boolean value)</code> | Define se as linhas poderão ser selecionadas. |

Depois conhecer os principais métodos da **JTable**, é hora de codificar. Tenha muita atenção a partir de agora aos códigos. Eles estarão comentados para que você entenda toda a estrutura de declaração e utilização.

Concluindo o projeto “PortalAluno”

Como um primeiro exemplo, no qual você pode se concentrar no uso de **JTable**, conclua o projeto “PortalAluno”, iniciado no conteúdo **Interface desktop... Parte 2** desta unidade. Caso não tenha estudado esse conteúdo, recomenda-se firmemente que você retorne até ele e o leia, acompanhando o passo a passo da implementação do projeto.



Abra o projeto “PortalAluno” no seu NetBeans. Até agora, o seu cadastro de alunos está funcionando. Chegou a hora de concluir o sistema exibindo todos os dados salvos na lista de alunos.

Para isso, abra o arquivo **Listagem.java**, clique com o botão direito do *mouse* sobre o componente **JTable** e selecione a opção **Customize Code...** Se você tentar fazer isso diretamente no visualizador do GUI Builder, provavelmente encontrará dificuldades para focar no **JTable**, pois ele priorizará o componente **JScrollPane**. Então, tente fazer o processo na janela **Navigator** localizada no lado esquerdo do Apache NetBeans IDE, logo abaixo da estrutura de arquivos.

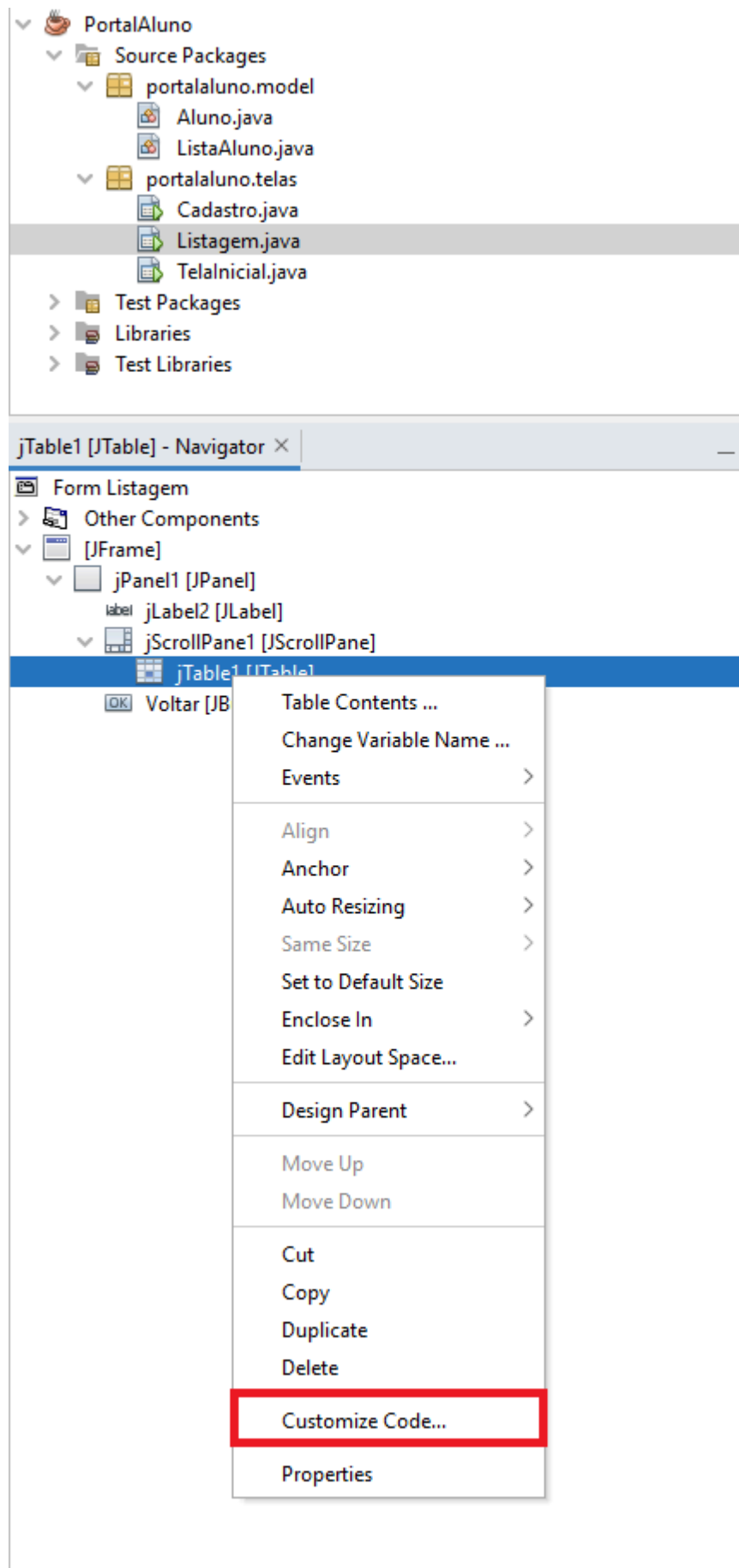
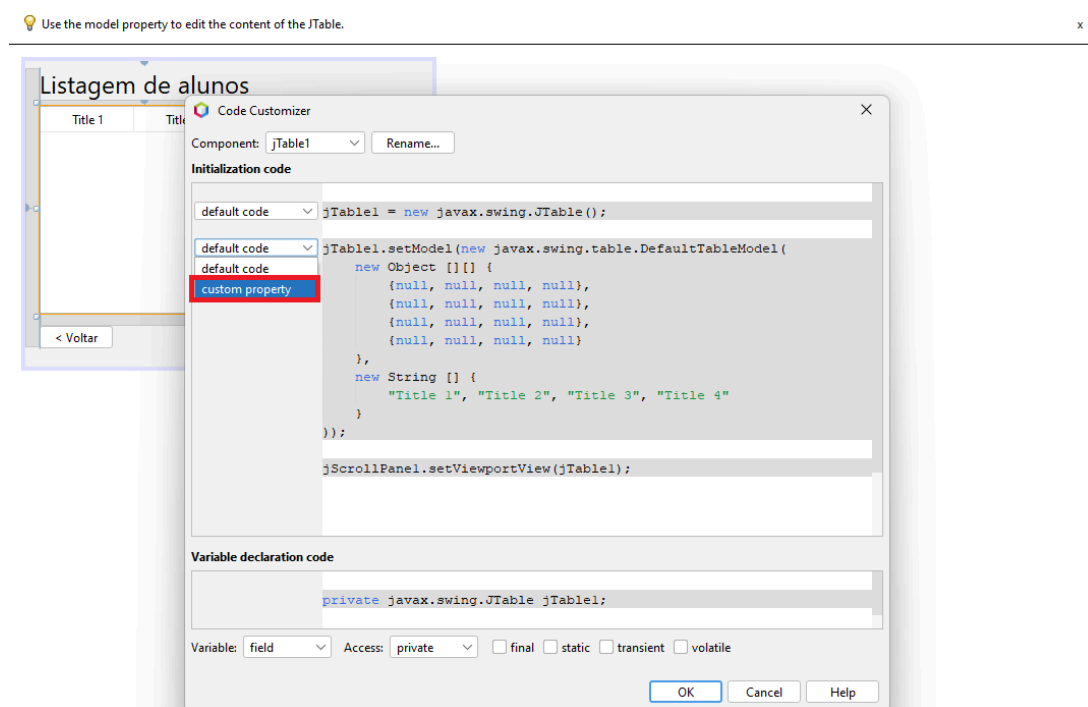


Figura 1 – Acessando o **Customize Code** do componente **JTable**

Fonte: Apache NetBeans IDE (2022)

Ao selecionar a opção, uma nova janela aparecerá. Repare que, ao lado do código, haverá duas caixas de seleção com o valor **default code** selecionado. Esse é o código de inicialização do componente **JTable** e essas caixas de seleção estão definindo que o código executado será o padrão. Para que o **JTable** exiba os dados da lista criada em **ListaAluno.java**, você precisa atualizar esse código de inicialização. Para isso, altere o valor da segunda caixa de seleção para **custom property**.

Figura 2 – **Code Customizer** do componente **JTable**

Fonte: Apache NetBeans IDE (2022)

Depois disso, escreva o seguinte código:



```
jTable1 = new javax.swing.JTable();

// Definimos o nome das colunas que queremos exibir
String[] colunas = { "Nome", "E-mail", "Curso" };
DefaultTableModel tabeloModelo = new DefaultTableModel(colunas, 0);

// Pegamos os dados cadastrados na lista de alunos
List<Aluno> listaCompleta = ListaAluno.Listar();

// Para cada item que houver na lista de alunos, iremos adicionar uma n
ova linha na nossa tabela
for(int i = 0; i < listaCompleta.size(); i++) {
    // Extraímos os dados
    Aluno alunoAtual = listaCompleta.get(i);

    // Montamos a linha
    String[] linha = {
        alunoAtual.getNome(),
        alunoAtual.getEmail(),
        alunoAtual.getCurso()
    };

    // Adicionamos a linha na tabela
    tabeloModelo.addRow(linha);
}
// Por fim, passamos para o jTable as novas configurações da tabela
jTable1.setModel(tabeloModelo);

jScrollPane1.setViewportViewView(jTable1);
```

Perceba que, ao escrever o código, cada nova linha gerará uma nova caixa de seleção ao lado com o valor **pre-init**. Mantenha esse valor. No final, a sua janela de customização de código deve ficar assim:

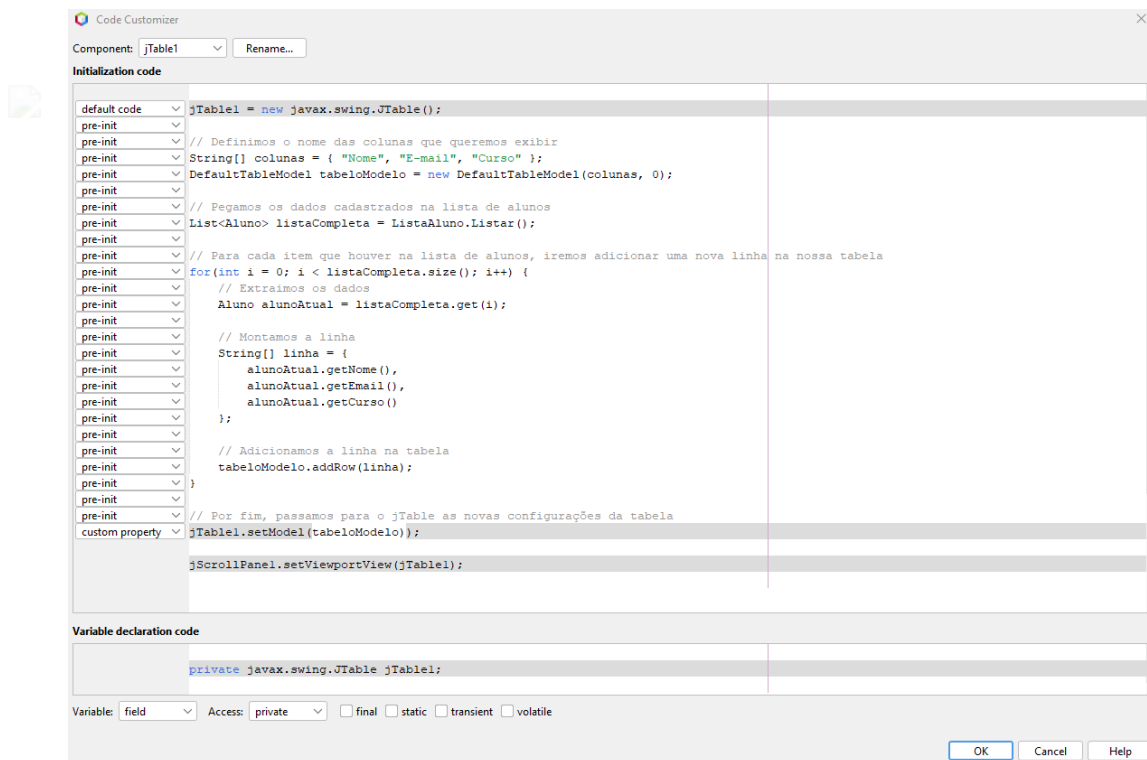


Figura 3 – Code Customize do componente JTable

Fonte: Apache NetBeans IDE (2022)

Observação

Perceba que as linhas realçadas em cinza não são editáveis. É o caso dos seguintes trechos:

```
jTable1 = new javax.swing.JTable();

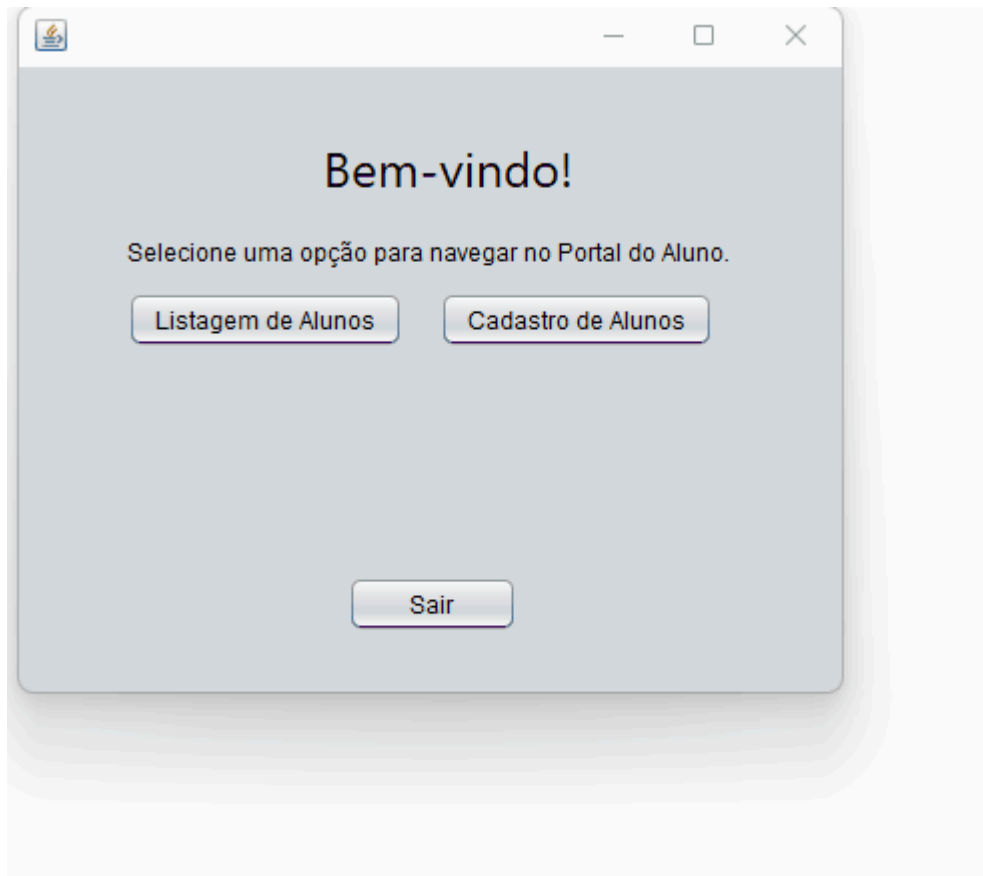
jTable1.setModel();

jScrollPane1.setViewportViewView(jTable1);
```

Para adicionar a variável **tabelaModelo** dentro do método **jTable1.setModel()**, escreva o código dentro dos parênteses.

Para concluir a edição, clique em **Ok**.

Se você tiver seguido esse passo a passo com bastante atenção, a sua aplicação já deve estar pronta e 100% funcional. Observe o resultado no GIF (*graphics interchange format*) a seguir.



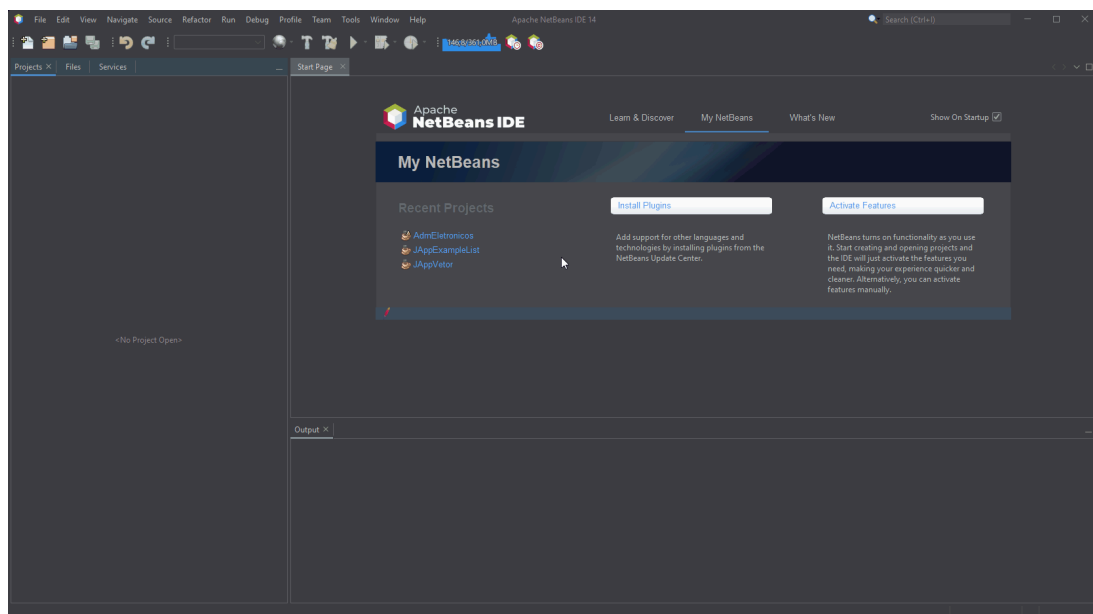
Construindo outro exemplo

Confira outro exemplo de construção de um novo projeto completo de cadastramento, utilizando **JTable** para controle de dados.

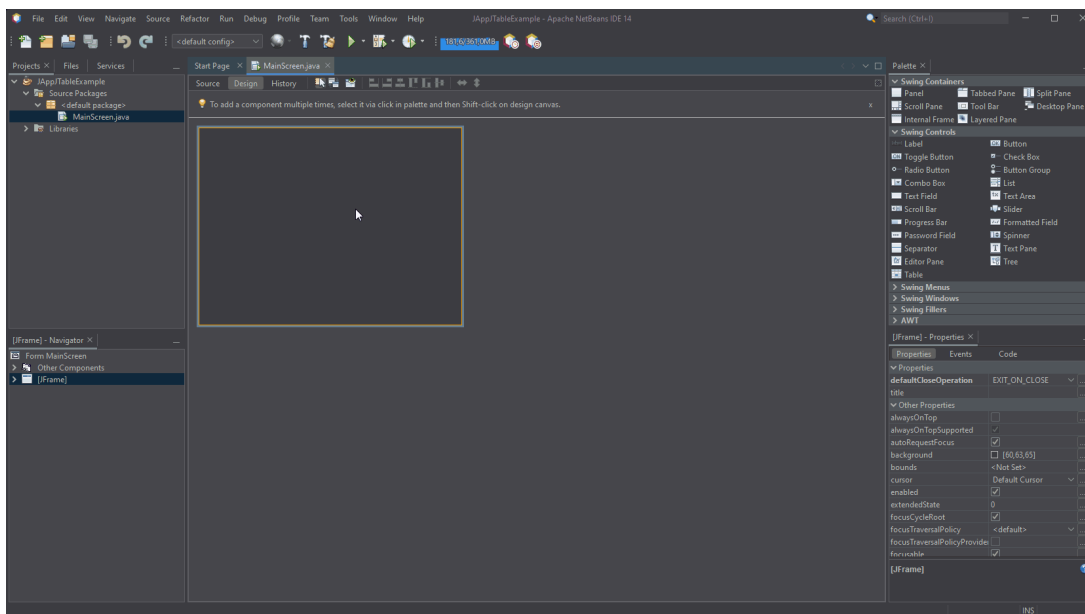
Será criada uma aplicação *desktop* utilizando a biblioteca **javax.swing** para a construção da interface gráfica. Cria-se um formulário de entrada de dados de alunos de uma instituição. Pelo formulário, inserem-se os alunos (nome, *e-mail* e idade). Toda vez que você clicar em **Salvar**, estará salvando

os dados novos em uma tabela ou atualizando-os, no caso de ser uma edição. Para editar ou remover alunos da tabela, será preciso selecionar o aluno correto. Caso tenha sido selecionada mais de uma linha, deve-se avisar o usuário e limpar a seleção.

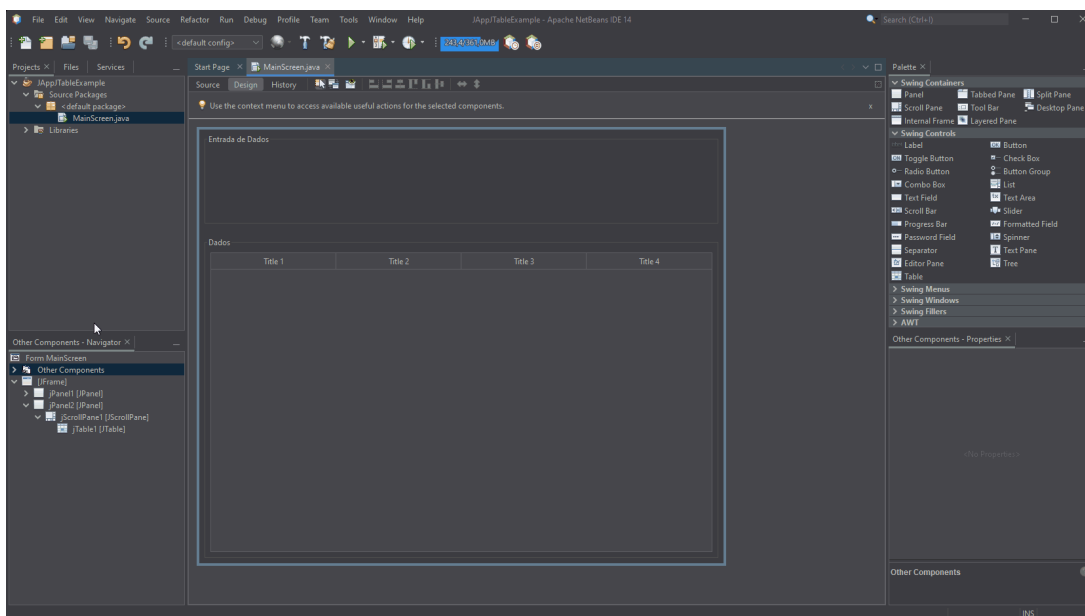
Para começar, crie então um novo projeto chamado “JAppJTableExample”, conforme o GIF a seguir.



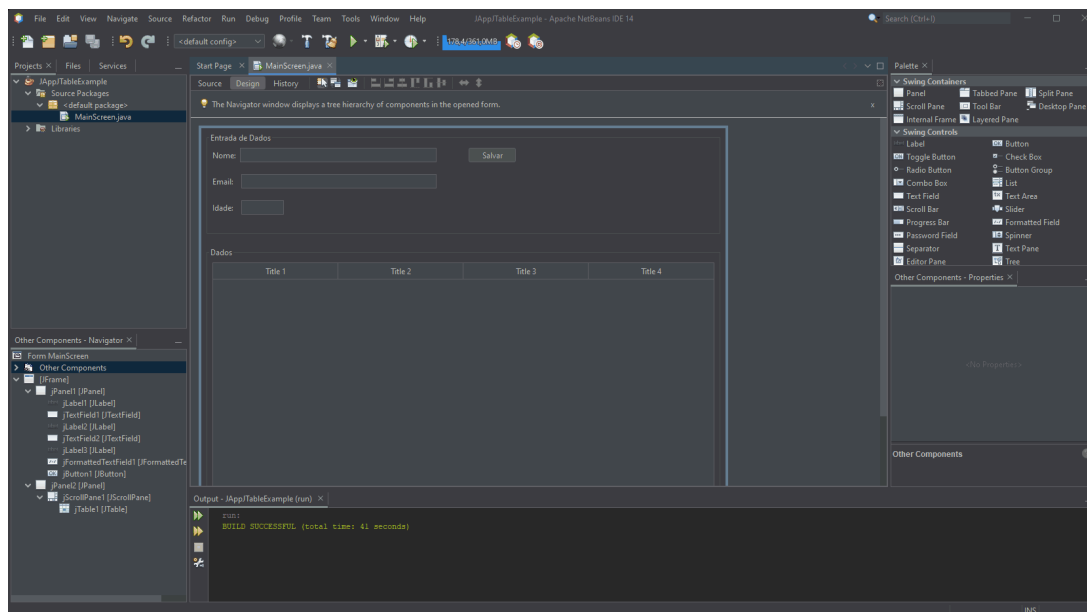
Agora que o projeto foi criado, ajuste o **JFrame**. Não se preocupe se você não conseguir deixar a sua exatamente igual, preocupe-se com a utilização dos componentes corretos. Nessa etapa, você incluirá dois painéis (**JPanel**) para organizar o que são entrada de dados e exibição dos dados. Em seguida, coloque a tabela na área de exibição de dados para poder observar como ela aparece no IDE (sigla em inglês para “ambiente de desenvolvimento integrado”), conforme este GIF.



Agora, insira os campos de textos e *labels* para nome, *e-mail* e idade. Em seguida, coloque o primeiro botão responsável por incluir o nome na tabela. Confira no GIF:

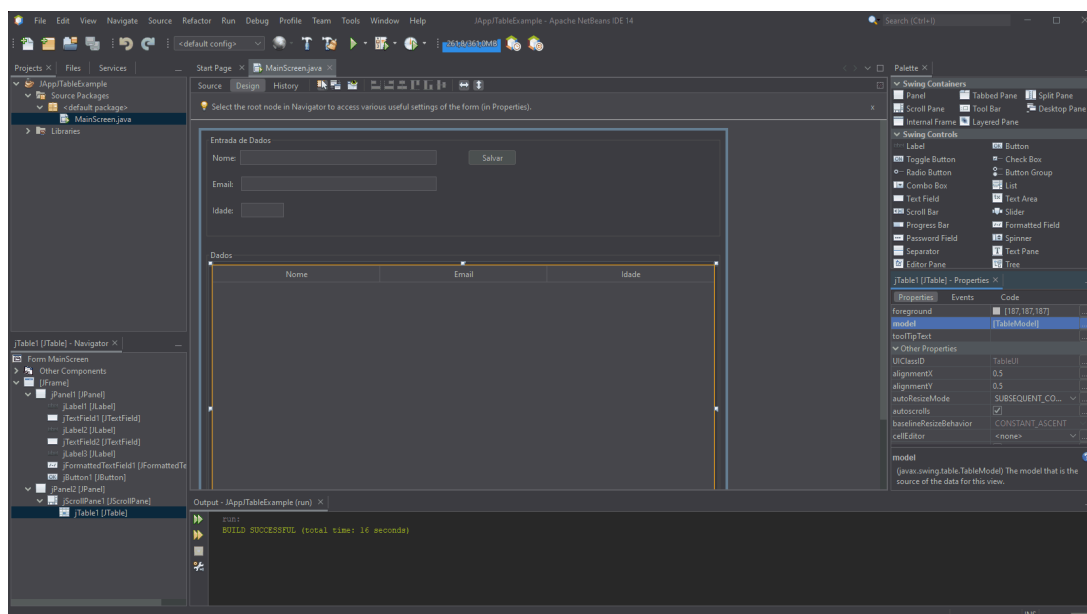


Perceba que a **JTable** está com quatro colunas e, para exibir os dados do aluno, são necessárias três colunas (nome, *e-mail* e idade). Realize os ajustes na tabela utilizando as propriedades dela para que as colunas fiquem corretas e com o cabeçalho correto. Ajuste também os tipos de dados de cada coluna e apague as linhas vazias que nela estão exibidas por padrão.



Antes de começar a codificar de fato, nomeie corretamente os componentes, pois, para utilizá-los, é preciso saber seus nomes. Geralmente, iniciam-se os nomes das *labels* como **lbl**, dos **TextFields** como **txt**, das **JTables** como **tbl**, dos *buttons* como **btn** e dos *panels* como **pnl**.

Observe o GIF a seguir:



Agora você criará a classe que representa o aluno com todas as suas características e os métodos assessores (*getters*) e modificadores (*setters*). Para isso, clique com o botão direito do *mouse* sobre o *default package* e

crie uma nova classe **Java**.





```
public class Aluno {

    /*
     * Definição e encapsulamento dos atributos da classe.
     * Representam as características de um aluno. Geralmente
     * são encapsuladas como private.
     */

    private String nome;
    private String email;
    private int idade;

    /*
     * Abaixo estão os métodos acessores e modificadores.
     * Os setters servem para atribuir valores aos atributos privados
     * Os getters servem para pegar os valores dos atributos privados
     */

    /**
     * @return the nome
     */
    public String getNome() {
        return nome;
    }

    /**
     * @param nome the nome to set
     */
    public void setNome(String nome) {
        this.nome = nome;
    }

    /**
     * @return the email
     */
    public String getEmail() {
        return email;
    }

    /**
     * @param email the email to set
     */
    public void setEmail(String email) {
        this.email = email;
    }

    /**
     * @return the idade
     */
}
```



```
    */  
    public int getIdade() {  
        return idade;  
    }  
  
    /**  
     * @param idade the idade to set  
     */  
    public void setIdade(int idade) {  
        this.idade = idade;  
    }  
}
```

Antes prosseguir, é importante saber que, para os dados “aparecerem” na tabela, os passos a seguir devem ser realizados exatamente na ordem apresentada.

A

Defina o *table model* com as mesmas características do componente editado via propriedades no modo *design* do IDE (quando são definidas as colunas “nome”, “email” e “idade”).

1. Crie no código um vetor de *strings* com o título de cada coluna, mesmos nomes criados no modo *design*.

2. Agora defina o **DefaultTableModel** informando ao seu construtor dois parâmetros: o primeiro será o vetor com a lista de títulos das colunas e o segundo, o índice da linha em que serão exibidos. Como se quer na primeira linha, informe o zero.

B

Será necessária uma **List<Aluno>**, ou seja, uma lista de objeto “Aluno”, que deve ser instanciada uma única vez para que essa seja a única lista de alunos da sua aplicação.

Confira a seguir os dois passos comentados.

```
public class MainScreen extends javax.swing.JFrame {

    /**
     * Passo A. 1
     * Definição do vetor com os títulos de cada coluna da tabela
     */
    private final String[] tableColumns = {"Nome", "Email", "Idade"};

    /**
     * Passo A. 2
     * Definição do table model que vai conter a nossa lista de alunos
     * o primeiro parâmetro é nosso vetor com os títulos das colunas
     * e o segundo parâmetro é o índice da linha que se quer exibir os
     * títulos. Nesse caso na primeira linha.
     */
    DefaultTableModel tableModel = new DefaultTableModel(tableColumns,
0);

    /**
     * Passo C.
     * Definição do vetor com os títulos de cada coluna da tabela
     */
    private List<Aluno> alunosList = new ArrayList<>();
```

Agora, comece a criar os métodos necessários para que sua aplicação funcione corretamente.



```
// Variables declaration - do not modify//GEN-BEGIN:variables
private javax.swing.JButton jButton1;
private javax.swing.JScrollPane jScrollPane1;
private javax.swing.JLabel lblEmail;
private javax.swing.JLabel lblIdade;
private javax.swing.JLabel lblNome;
private javax.swing.JPanel pnlDataInput;
private javax.swing.JPanel pnlDataOutput;
private javax.swing.JTable tblAlunos;
private javax.swing.JTextField txtEmail;
private javax.swing.JFormattedTextField txtIdade;
private javax.swing.JTextField txtNome;
// End of variables declaration//GEN-END:variables

/*
 * A partir desse local vamos inserir nossos métodos.
 */
```

Todos os métodos criados serão inseridos ao final do arquivo **MainScreen.java**.


Agora será realizada a inclusão de novos registros de alunos na lista. O primeiro método será o **emptyFields()**, que validará se os campos de nome, *e-mail* e idade da tela estão preenchidos. Esse método retornará “verdadeiro”, caso existam quaisquer campos vazios, e “falso” somente quando todos estiverem preenchidos. Também emitirá uma caixa de mensagem ao detectar campo vazio.



```
/*
 * Metodo emptyFields(), retorna "true" quando qualquer um dos campos
 * estiver vazio e "false" quando todos estiverem preenchidos.
 */
private boolean emptyFields(){
    /*
     * Variável empty assume "true" por padrão e só vai mudar o seu esta
do
     * após verificarmos se todos os campos estão preenchidos. É ela que
     * retornamos ao final do método.
     */
    boolean empty = true;

    /*
     * Agora vamos testar campo a campo e exibir avisos caso encontre
     * algum campo vazio.
     */

    if(txtNome.getText().trim().isEmpty()){
        /*
         * se o campo txtNome estiver vazio exibimos a mensagem
         * de campo vazio. Caso contrário testamos o próximo
         */
        JOptionPane.showMessageDialog(rootPane, "ATENÇÃO! Nome não pode
ser vazio.");
    } else if(txtEmail.getText().trim().isEmpty()){
        /*
         * se o campo txtEmail estiver vazio exibimos a mensagem de campo
         * vazio. Caso contrário testamos o próximo campo.
         */
        JOptionPane.showMessageDialog(rootPane, "ATENÇÃO! Email não pod
e ser vazio.");
    } else if(txtIdade.getText().trim().isEmpty()){
        /*
         * se o campo txtIdade estiver vazio exibimos a mensagem de camp
o
         * vazio. Caso contrário testamos o próximo campo.
         */
        JOptionPane.showMessageDialog(rootPane, "ATENÇÃO! Idade não pod
e ser vazio.");
    } else{
        /*
         * se cairmos nessa condição é porque não existem campos vazios,
         * logo, a variável "empty" deve mudar seu valor para "false".
         */
        empty = false;
    }
}
```



```
/*
 * Aqui retornamos a variável "empty"
 */
return empty;
}
```

Agora, será criado o método **getAluno()**, que retorna um objeto Aluno com todos os seus atributos preenchidos a partir das informações dos campos da tela. Isso deve ser feito logo abaixo do método criado anteriormente:

```
/*
 * Metodo getAluno(), retorna um aluno
 * com valor em todos os atributos
 */
private Aluno getAluno(){

    /*
     * Criação de objeto Auno, ou seja, uma instância de Aluno
     */
    Aluno aluno = new Aluno();

    /*
     * Agora que temos um objeto aluno, vamos atribuir os valores
     * que foram digitados nos campos de texto. Nome, email e idade.
     * Para "nome", vamos deixar todas as letras em maiúsculo. Para
     * "email", vamos deixar todas as letras em minúsculo. Para idade, é
     * preciso, converter para inteiro;
     */
    aluno.setNome(txtNome.getText().toUpperCase());
    aluno.setEmail(txtEmail.getText().toLowerCase());
    aluno.setIdade(Integer.parseInt(txtIdade.getText()));

    /*
     * Aqui retornamos o objeto "aluno"
     */
    return aluno;
}
```

Uma vez verificado se existem campos vazios e “obtido o aluno” digitado pelo usuário, é o momento de criar o método sem retorno **inserirAluno(Aluno aluno)**, que inserirá esse aluno recebido por parâmetro na **List<Aluno>** toda vez que se clicar em **Salvar**.

```
/*
 * Metodo inserirAluno(Aluno aluno), insere o aluno
 * na lista de alunos
 */
private void inserirAluno(Aluno aluno){

    /*
     * Inserir o aluno na lista
     */
    alunosList.add(aluno);

}
```

No entanto, somente isso não fará com que a tabela exiba os dados desse aluno. Para isso, construa o método sem retorno **atualizarTabela()**, que percorrerá toda a lista, considerando aluno por aluno, insira seus atributos na nova **tableModel** e, ao final, “sete” a **tableModel** à tabela.



```
/*
 * Metodo atualizarTabela(), vai atualizar a tabela toda vez
 * que inserido, removido ou alterado dados de um aluno;
 */
private void atualizarTabela(){

    /*
     * antes de atualizar a tabela é uma boa prática
     * verificar se a lista de alunos NÃO está vazia, pois,
     * se estiver vazia não precisamos atualizar.
     */
    if(!alunosList.isEmpty()){

        /*
         * Como não está vazia, vamos criar um Aluno que será
         * será instanciado em cada ocorrência de aluno na lista,
         * isso dentro de um laço de repetição "for".
         */
        Aluno aluno;

        /*
         * Também é necessário recriar o tableModel, para limpar
         * os dados anteriores
         */
        tableModel = new DefaultTableModel(tableColumns, 0);


        /*
         * Agora vamos criar o for que vai repetir
         * até o tamanho da lista de alunos
         */
        for (int i = 0; i < alunosList.size(); i++) {

            /*
             * Instância de Aluno com os dados do aluno
             * de cada posição da lista
             */
            aluno = alunosList.get(i);

            /*
             * Agora vamos criar um vetor de String com cada um
             * dos valores dos atributos do aluno encontrado.
             * Vamos chamar de "rowData".
             */
            String[] rowData = {aluno.getNome(), aluno.getEmail(), String.valueOf(aluno.getIdade())};

            /*
             * Agora adicionamos o vetor de dados na tableModel

```



```
        */
        tableModel.addRow(rowData);
    }

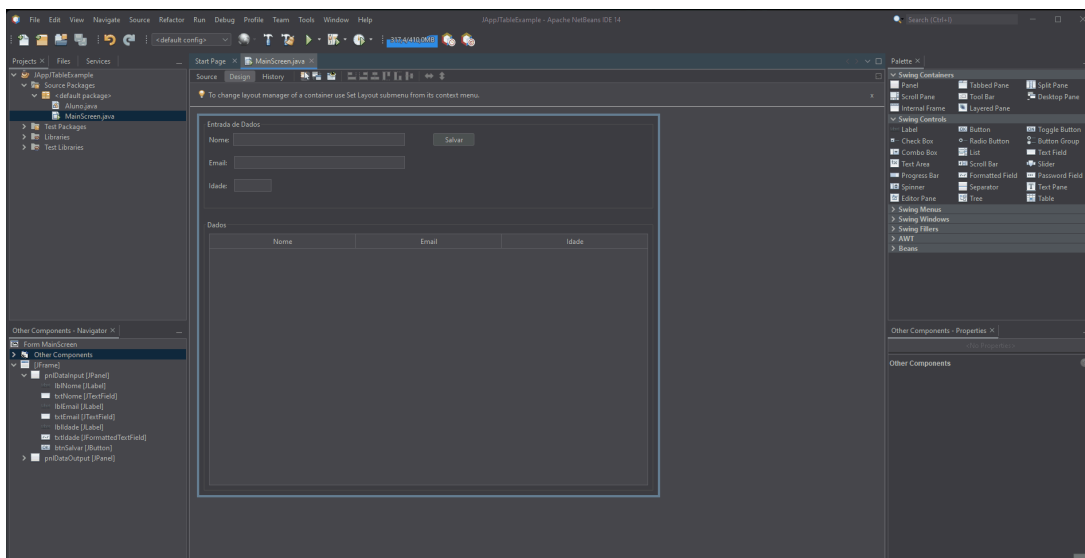
    /*
    * Após concluído o for e com isso colocar todos os
    * dados antigos e os novos no tableModel, é hora de
    * setar o modelo na tabela.
    */
    tblAlunos.setModel(tableModel);
}else{
    /*
    * No caso de excluirmos o último aluno da lista
    * teremos de criar uma tableModel nova para "limpar
    * a tabela" e setar na tblAlunos.
    */
    tableModel = new DefaultTableModel(tableColumns, 0);
    tblAlunos.setModel(tableModel);
}
}
```

Esse método deve ser chamado dentro do método **inserirAluno(Aluno aluno)**, logo após o aluno ser inserido na lista.

```
private void inserirAluno(Aluno aluno){
    /*
    * Inserir o aluno na lista
    */
    alunosList.add(aluno);

    /*
    * Inclusão do método atualizarTabela()
    */
    atualizarTabela();
}
```

Agora, cria-se o evento do botão **Salvar**. Os eventos de clique são os mais simples de se atribuir. Para isso, basta dar um duplo clique sobre o componente, neste caso, o botão **Salvar**. Confira no GIF a seguir:



Agora que já foi atribuído o evento, chame os métodos na ordem correta para poder incluir o aluno na tabela. Estabeleça a seguinte sequência:


A

Verifique os campos, ou seja, chame o método **emptyFields()**.

B

Se os campos não estiverem vazios, “pegue” os dados dos campos de textos e atribua-os ao aluno a ser incluído na lista. Para isso, chame o método **getAluno()**.

C

De posse do objeto `Aluno` com os atributos valorados, chame o método  **`inserirAluno(Aluno aluno)`**, passando o seu objeto que retornou do método **`getAluno()`**.

D

Após a inclusão, é uma boa prática limpar todos os campos para evitar que o usuário fique clicando em **Salvar** e incluindo várias vezes os mesmos dados.



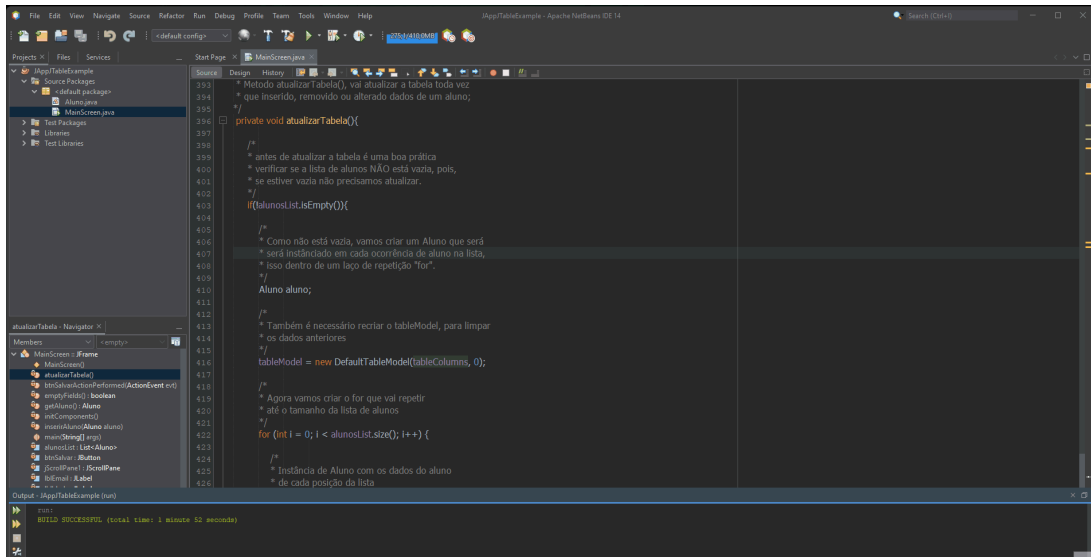
```
private void btnSalvarActionPerformed(java.awt.event.ActionEvent evt)
{
    //GEN-FIRST:event_btnSalvarActionPerformed
    // TODO add your handling code here:

    /*
     * Antes de fazermos qualquer coisa precisamos
     * saber se há algum campo vazio, como já criamos
     * um método que faz isso, é só chamá-lo. Lembre-se
     * que se algum campo estiver vazio o retorno é "true",
     * caso contrário é false. Usamos ele no if precedido pelo
     * "!" (ponto de exclamação). Dessa forma lê-se "se
     * campos não vazios". Dessa maneira se o retorno for
     * "false" podemos executar a inserção do aluno.
     */
    if(!emptyFields()){
        /*
         * Agora basta enviar o aluno que vai retornar
         * do método getAluno para o método inserir
         * aluno. O método getAluno(), vai ser executado
         * antes do inserirAluno.
         */
        inserirAluno(getAluno());

        /*
         * Depois de inserir o aluno é uma boa prática
         * limpar os campos do formulário e setar o
         * cursor no primeiro campo com o método
         * "requestFocus()";
         */
        txtNome.setText("");
        txtEmail.setText("");
        txtIdade.setText("");

        txtNome.requestFocus();
    }
}
```

Agora, você já pode compilar e testar a sua aplicação. Veja o resultado esperado neste GIF:



Sua aplicação já está inserindo dados, então agora siga para a exclusão de um aluno. Para isso, é preciso verificar se o aluno foi selecionado na tabela. Se foi selecionado, “pegue” sua posição na tabela (que é a mesma da lista), exclua-a da lista e atualize a tabela. Para tornar tudo mais fácil, crie um método chamado **getPosicaoAluno()**, que retornará um valor inteiro, o qual será a posição do aluno na lista ou -1, caso não esteja selecionado.



```
/*
 * Metodo getPosicaoAluno(), vai retornar um valor inteiro maior ou igual
 * a zero, caso algum aluno esteja selecionado ou -1 caso não haja seleção.
 */
private int getPosicaoAluno(){

    /*
     * Criação da variável posAluno do tipo inteiro que vai receber
     * o retorno do método getSelectedRow() da tblAlunos
     */
    int posAluno = tblAlunos.getSelectedRow();

    /*
     * Caso o valor seja -1 vamos informar ao usuário para ele
     * selecionar o aluno
     */
    if(posAluno == -1){
        JOptionPane.showMessageDialog(rootPane, "Selecione um aluno");
    }

    /*
     * Aqui retornamos a variável posAluno;
     */
    return posAluno;
}
```

Para excluir um aluno da lista, você pode criar um método, sem retorno, específico para isso, chamado **excluirAluno(int posAluno)**. Esse método será responsável por remover o aluno da lista de alunos na posição informada. Caso seja maior ou igual a zero, será uma posição válida para exclusão. Lembre-se de que esse assunto já foi tratado no método **getPosicaoAluno()**.



```
/*
 * Metodo sem retorno excluirAluno(int posAluno), vai pedir confirmação
o
 * para excluir o aluno da posição informada.
 */
private void excluirAluno(int posAluno){

    /*
     * Antes de realizar a exclusão precisamos ter certeza que a posição
     * é válida, ou seja, maior ou igual a zero;
     */
    if(posAluno >= 0){

        /*
         * Agora vamos pedir ao usuário a confirmação de exclusão.
         * Utilizaremos o método showOptionDialog() que permite
         * uma melhor customização se o usuário clicar "sim" o retorno
         * é 0. Se clicar em "Não" o retorno é 1. Posição do vetor "optio
ns";
         */

        String[] options = { "Sim", "Não" };

        int deletar = JOptionPane.showOptionDialog(
            rootPane,
            "Tem certeza que deseja excluir?",
            "Exclusão de aluno",
            JOptionPane.DEFAULT_OPTION,
            JOptionPane.QUESTION_MESSAGE,
            null,
            options,
            options[0]
        );

        /*
         * Se a variável deletar for 0 vamos deletar. Caso contrário
         * vamos limpar a seleção na tabela
         */
        if(deletar == 0){
            /*
             * Agora removemos a ocorrência desse aluno
             * de alunosList por meio do método, remove(int index)
             */
            alunosList.remove(posAluno);

            /*
             * Depois de removido da lista precisamos atualizar a

```

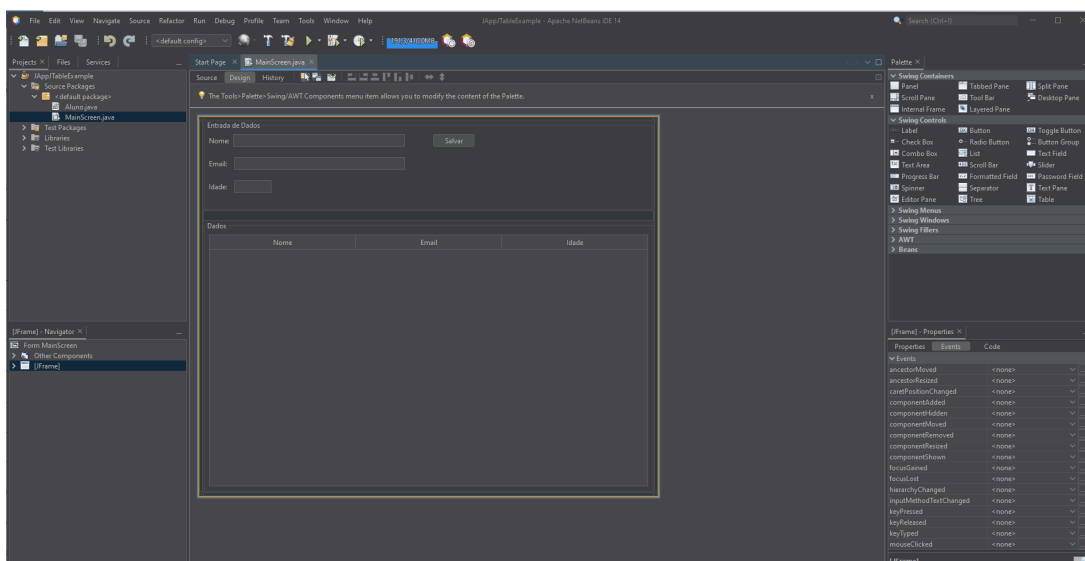



```

        * tabela. Basta chamar o método atualizarTabela() já
        * construído anteriormente
        */
        atualizarTabela();
    }else{
        /*
        * Limpar a seleção pois o usuário cancelou a exclusão.
        */
        tblAlunos.clearSelection();
    }
}
}

```

Atualize sua interface gráfica inserindo o botão **Excluir** e atribua o evento por meio do duplo clique sobre o componente, conforme o GIF a seguir:



MainScreen atualizada, é o momento de chamar os métodos na ordem correta para que você possa excluir o aluno da tabela. Caso o usuário não confirme a exclusão, limpe a seleção para evitar deleção por engano. Estabeleça a seguinte sequência:

A

Chame o método **excluirAluno(int posAluno)**.



B

Dentro dos parênteses do método anterior, chame o método **getPosicaoAluno()**, dessa forma, você estará passando posição do aluno a ser excluído.

C

Se o usuário for deletado, basta atualizar a lista de alunos.

```
private void btnExcluirActionPerformed(java.awt.event.ActionEvent evt)
{
    //GEN-FIRST:event_btnExcluirActionPerformed
    // TODO add your handling code here:

    /*
     * Agora basta enviar a posição do aluno para
     * o método excluirAluno(int posAluno).
     * O método getPosicaoAluno(), vai ser executado
     * antes do excluirAluno.
     */
    excluirAluno(getPosicaoAluno());
}
}
```

Bibliotecas de linguagem: mapas e dicionários

Além das estruturas de lista, indexadas diretamente por números, Java conta com classes que implementam estruturas de dicionários, nas quais a indexação ocorre a partir de outro valor (um texto ou um objeto, por

exemplo). Isso ajuda a resolver problemas em que se necessite de acesso rápido a uma informação a partir de outra (desde que esta outra seja única).

Conceito

Mapas são estruturas de dados que armazenam pares de **chave/valor**, nos quais um valor-chave indexa a informação, permitindo operações rápidas de acesso, atualização e exclusão de dados usando a **chave para acesso direto ao valor**.

Já nas listas e nos vetores utiliza-se um número inteiro como índice para acessar uma informação armazenada. Nos mapas, o índice de acesso é algum valor calculado ou informado (podendo ser um objeto qualquer). Além disso, o valor da chave no mapa é único para cada entrada.

Confira na figura a seguir a esquematização de uma estrutura de mapa. Cada elemento é um par de chave+dados. À direita consta um exemplo de mapa cuja chave é o CPF de uma pessoa e cujo valor é o nome dessa pessoa.

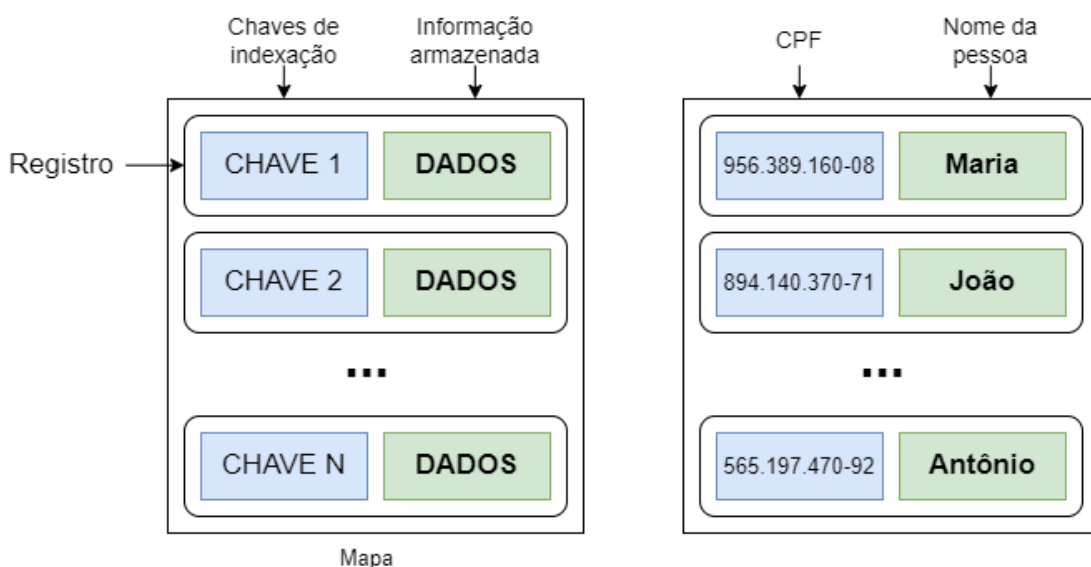


Figura 4 – Esquematização de uma estrutura de mapa



Fonte: Senac EAD (2022)

Os mapas também podem ser denominados **dicionários**. Formalmente, há uma pequena diferença nos conceitos, mas, na prática, funcionam da mesma maneira.

Java utilizava uma classe abstrata chamada **Dictionary** para mapas, mas há algumas versões essa classe se tornou obsoleta, recomendando o uso da interface **Map**.

Usabilidade

Mapas são úteis para situações em que é necessário realizar consultas frequentes a dados e esses dados possam ser referenciados por alguma informação específica. Um exemplo está na figura 4, na qual cada pessoa está indexada pelo seu CPF, que é uma informação única para cada cidadão.

Imagine outra situação em que é necessário processar os dados de alunos de cada sala de uma escola. Seria possível ter nesse caso um mapa cuja chave é o número da sala e o valor, uma lista de alunos. Assim, o usuário poderia acessar com velocidade e praticidade as informações necessárias a partir do número da sala.

Situações análogas a essas podem se beneficiar do uso de mapas/dicionários.

Aplicação

A interface **Map** é parte do Collections Framework de Java e define as operações básicas para mapas.

Os principais métodos para **Map<C,V>** são:

- ◆ **put(C chave, V valor)**: incluir no mapa “valor” (do tipo V especificado por *generics*) indexado por “chave” (do tipo genérico C).
- ◆ **remove(C chave)**: remover o dado indexado por “chave” no mapa.
- ◆ **keySet()**: retorna a lista (**Set<C>**) de chaves do mapa.
- ◆ **get(C chave)**: retorna o valor indexado por “chave”.
- ◆ **containsKey(Object o)**: retorna verdadeiro se há uma chave que seja igual ao objeto “o” informado por parâmetro.
- ◆ **isEmpty()**: retorna verdadeiro se o mapa está vazio.

A principal classe concreta que implementa **Map** é **HashMap**. O exemplo a seguir apresenta uma implementação baseada no mapa demonstrado na figura 4 com dados de pessoas indexadas pelo seu CPF.

Primeiro, cria-se uma classe muito simples para “Pessoa”.

```
public class Pessoa {  
    public String nome;  
    public int idade;  
  
    public Pessoa(String nome, int idade)  
    {  
        this.nome = nome;  
        this.idade = idade;  
    }  
}
```

No código principal, construa e utilize um **HashMap<C,V>** para armazenar objetos do tipo Pessoa indexados por CPF (*string*). Observe com atenção os comentários presentes no código.



```
import java.util.HashMap;
import java.util.Map;
import java.util.Scanner;
import java.util.Set;

public class MapTeste {

    public static void main(String[] args) {
        Scanner entrada = new Scanner(System.in);

        //declarando o HashMap, com tipo String para chave e tipo Pessoa
para valor
        Map<String, Pessoa> pessoas = new HashMap<String, Pessoa>();
        Pessoa p;

        //incluindo valores "Pessoa" no mapa
        p = new Pessoa("Maria Silva", 25);
        pessoas.put("854.093.660-79", p);
        p = new Pessoa("João Siqueira", 20);
        pessoas.put("750.005.080-18", p);

        //Pessoa, naturalmente, pode ser instanciado direto no parâmetro
        pessoas.put("196.761.060-68", new Pessoa("Antônio Sá", 45));

        //na inclusão abaixo fará substituição do valor anterior, já que
        usa a mesma chave
        pessoas.put("196.761.060-68", new Pessoa("Zeferino Querencia", 5
0));

        //mostrando na tela o mapa completo
        System.out.println("Mapa completo: ");
        Set<String> chaves = pessoas.keySet();
        for(String chave : chaves)
        {
            System.out.println(chave + " -> " + pessoas.get(chave).nome);
        }

        //realizando busca
        System.out.println("Digite um cpf: ");
        String cpfBusca = entrada.nextLine();
        if(pessoas.containsKey(cpfBusca))
        {
            Pessoa resultado = pessoas.get(cpfBusca);
            System.out.println("Pessoa: " + resultado.nome + "; Idade: "
+ resultado.idade);
        }
        else
        {
```



```
        System.out.println("Não há registro de pessoa com este CPF");  
    }  
}  
}
```

Uma classe semelhante a **HashMap** é **HashTable**, que funciona de maneira análoga, mas que pode ser usada em aplicações com *multithread*. **HashMap** admite *null* na chave e no valor; **HashTable** não aceita *null*.

Outra classe comparável a **TreeSet** é a **HashTree**, um mapa que mantém as chaves ordenadas.

Encerramento

Neste material, você estudou **List** e **JTable**, que são largamente utilizadas nas aplicações Java. As **Lists**, por meio dos seus principais métodos, são utilizadas para manipular uma coleção de objetos de modo dinâmico; a **JTable**, por intermédio dos seus principais métodos, é utilizada como uma componente de representação gráfica, em forma de tabela, das listas de objetos. Foram apresentados exemplos práticos que o aproximaram e muito da realidade da utilização desses conceitos em projetos de *software*. Este material deve ser revisitado sempre que necessário, então aproprie-se desse conteúdo, pois você o utilizará até o final do curso e em toda sua vida profissional.