|  | FTCE  Faculdade de Tecnologia e Ciências Exatas |
| --- | --- |

USJT – 2021/1 – Programação de Soluções Computacionais

**Aula: 15 – Programação Orientada a Objetos – Interface Gráfica (GUI) e JDBC**

**1 Introdução a Interfaces gráficas**

Neste material ainda daremos continuidade ao desenvolvimento da aplicação da última aula. Vamos falar sobre o relacionamento entre alunos e cursos, desde o modelo conceitual dos dados, passando pelo mapeamento para o modelo relacional chegando à sua implementação no banco de dados. A aplicação receberá uma nova tela, que será chamada a partir da tela de cursos, que deverá exibir todos os alunos do curso, utilizando a JTable do pacote javax.swing.

**2 O modelo de dados**

**2.1 (Modelagem Conceitual)** Sabemos que um dos modelos conceituais de bancos de dados é o Modelo Entidade Relacionamento, que contém as entidades sobre as quais devemos armazenar informações e os relacionamentos entre elas. Essas informações são representadas como seus atributos, que podem ser tanto de entidades quanto de relacionamentos. A figura 2.1.1 mostra o modelo que vamos considerar para esta aula.

Figura 2.1.1

Tela de celular com texto preto sobre fundo branco

Descrição gerada automaticamente

**2.2 (Modelo Lógico)** A fim de implementarmos esse modelo, devemos fazer o seu mapeamento do modelo conceitual para o modelo lógico relacional, pois o SGBD que utilizamos implementa esse modelo. A Listagem 2.2.1 mostra como fica esse mapeamento.

Listagem 2.2.1

| aluno (ra, nome, ano\_nascimento)  curso (id, nome, tipo)  aluno\_curso (**ra**, **id**, semestre, ano)  **ra referencia aluno**  **id referencia curso** |
| --- |

Conforme ilustra a Listagem 2.2.1 os atributos identificadores das entidades do modelo ER foram mapeados para as chaves primárias das relações (tabelas) que representam essas entidades e o relacionamento (n para n) foi mapeado para uma tabela, cuja chave primária é composta pelas chaves estrangeiras que referenciam cada tabela que representa suas respectivas entidades.

**2.3 (Modelo Físico)** Finalmente, vamos implementar o modelo físico dos novos componentes, no nosso banco **db\_sistema\_academico**. A listagem 2.3.1 contém a criação das tabelas *aluno* e *aluno\_curso*.

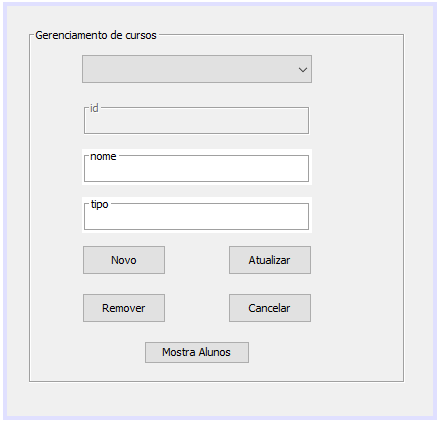
Listagem 2.3.1

| CREATE TABLE `db\_sistema\_academico`.`tb\_aluno` (  `RA` INT(11) NOT NULL,  `nome` VARCHAR(80) NOT NULL,  `ano\_nascimento` INT NULL,  PRIMARY KEY (`RA`)  );  CREATE TABLE `db\_sistema\_academico`.`tb\_aluno\_curso` (  `id\_curso` INT(11) NOT NULL,  `RA\_aluno` INT(11) NOT NULL,  `ano` INT NULL,  `semestre` INT NULL,  PRIMARY KEY (`id\_curso`, `RA\_aluno`),  INDEX `fk\_aluno\_idx` (`RA\_aluno` ASC),  CONSTRAINT `fk\_aluno`  FOREIGN KEY (`RA\_aluno`)  REFERENCES `db\_sistema\_academico`.`tb\_aluno` (`RA`)  ON DELETE NO ACTION  ON UPDATE NO ACTION,  CONSTRAINT `fk\_curso`  FOREIGN KEY (`id\_curso`)  REFERENCES `db\_sistema\_academico`.`tb\_curso` (`id`)  ON DELETE NO ACTION  ON UPDATE NO ACTION  ); |
| --- |

As chaves estrangeiras permitem a verificação da integridade referencial, isto é, não é possível criar um registro na tabela aluno\_curso sem que o aluno exista na tabela aluno e o curso exista na tabela curso. Mais adiante, vamos inserir alguns dados no banco para preparamos os elementos para a nova tela.

**2.4 (Criando a nova tela)** A tela do nosso dashboard não será alterada, você pode fazer como exercício o gerenciamento dos alunos, com base no gerenciamento de cursos lá no item 2.6 você verá que serão acrescentados alguns detalhes importantes quanto à integridade dos dados. O que vamos implementar é uma funcionalidade na nossa tela de cursos que nos permite visualizar os alunos de um curso escolhido. Para isso, vamos alterar a nossa classe CursosTela, acrescentando um botão que abra a tela que deve conter os alunos do curso escolhido. A Figura 2.4.1 mostra como deve ficar essa tela. Não se esqueça de atualizar o nome da variável para o novo botão, seu texto e utilizar os 2 clicks sobre ele para que ele acrescente ao código o método private void mostraAlunosButtonActionPerformed(java.awt.event.ActionEvent evt).

Figura 2.4.1



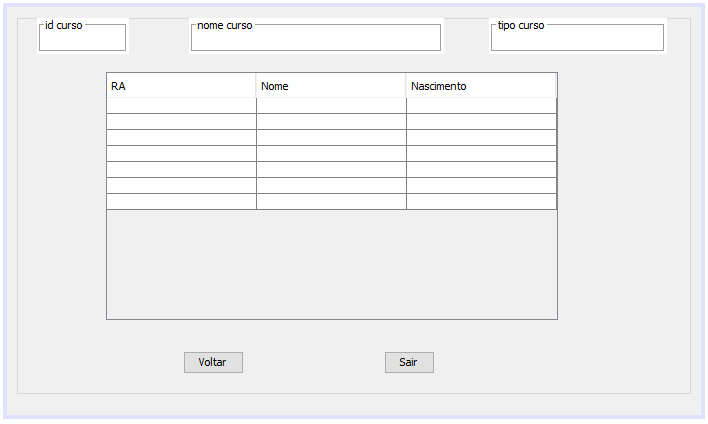
A ideia é que, ao clicarmos no botão mostraAlunosButton, seja exibida a tela mostraAlunosCursoTela ilustrada na Figura 2.4.2. Mais adiante, vamos acrescentar novas funcionalidades, mas por enquanto, vamos exibir os alunos de um determinado curso e voltar a tela anterior ou simplesmente sair da aplicação.

Antes da tela, vamos acrescentar a ação ao novo botão. A listagem 2.4.1 mostra o código que devemos acrescentar ao método gerado quando clicamos 2 vezes no botão, durante a criação da tela.

Listagem 2.4.1

| private void mostraAlunosButtonActionPerformed(java.awt.event.ActionEvent evt) {  mostraAlunosCursoTela ac = new mostraAlunosCursoTela();  ac.setVisible(true);  this.dispose();  } |
| --- |

Figura 2.4.2



As caixas de texto (jTextfields) já são conhecidas, vamos começar com elas. Não se esqueçam de nome apropriados para as variáveis. Os botões também já ficaram fácil, vamos alterar os textos exibidos, os nomes das variáveis e as ações que devem ser executadas quando clicados (2 clicks em cada um para gerar o esqueleto dos métodos ButtonActionPerformed). A Listagem 2.4.2 mostra o construtor com os ajustes necessários e os códigos associados a cada um dos botões.

Listagem 2.4.2

| public MostraAlunosCursoTela() {  super("Alunos por Curso");  initComponents();  setLocationRelativeTo(null);  }  private void voltarButtonActionPerformed(java.awt.event.ActionEvent evt) {  CursosTela ct = new CursosTela();  ct.setVisible(true);  this.dispose();  }  private void sairButtonActionPerformed(java.awt.event.ActionEvent evt) {  this.dispose();  } |
| --- |

Note que a tela para a qual desejamos voltar é a tela de cursos, pois é possível que o usuário queira escolher outro curso para exibir seus alunos ou de lá, ele pode voltar para o Dashboard. É importante termos esse mapa de navegação, ele está de acordo com o diagrama de sequência que descreve a sequência das ações no seu sistema.

**2.5 (Trazendo as informações da tela anterior)** Note que a tela de alunos por curso traz as informações do curso, que foi escolhido na classe CursosTela. Parece natural, então que a classe MostraAlunosCursoTela seja iniciada com essas informações, podemos fazer isso, passando a instância de curso como parâmetro em seu construtor, veja a Listagem 2.5.1.

Listagem 2.5.1

| public MostraAlunosCursoTela(Curso curso) {  super("Alunos por Curso");  initComponents();  idCursoLabel.setText(Integer.toString(curso.getId()));  nomeCursoLabel.setText(curso.getNome());  tipoCursoLabel.setText(curso.getTipo());  setLocationRelativeTo(null);  } |
| --- |

Como sugerido anteriormente, podemos deixar esses campos não editáveis, pois eles vêm da tela anterior. Podemos fazer essa alteração desmarcando a propriedade *editable* de cada um deles.

**2.6 (Voltando aos dados)** Nós deixamos o modelo do banco preparado para receber alunos, que serão matriculados em um curso. Lembre-se, para cadastrar um aluno num curso, é necessário que o aluno e o curso estejam cadastrados. Vamos fazer alguns testes. A Listagem 2.6.1 contém algumas inserções para realizarmos alguns testes. Vejam, é possível realizar múltiplas inserções.

Listagem 2.6.1

| INSERT INTO tb\_aluno (RA, nome, ano\_nascimento) VALUES  (123456, "João Henrique", 2000),  (345678, "Ana Maria", 2001),  (875321, "Paulo Rogerio", 2000),  (456789, "Maria Regina", 1999)  ; |
| --- |

A Listagem 2.6.2 mostra como registrar que um determinado aluno está associado a um determinado curso em um determinado período.

Listagem 2.6.2

| INSERT INTO tb\_aluno\_curso VALUES (1, 123456, 2020, 1); |
| --- |

**2.7 (JTable)** JTable é uma classe do pacote javax.swing usada para exibir e editar tabelas bidimensionais de itens denominados “células”. Sua documentação oficial pode ser encontrada na Link 2.7.1.

Link 2.7.1

https://docs.oracle.com/javase/8/docs/api/javax/swing/JTable.html

O Link 2.7.2 mostra um tutorial detalhado.

Link 2.7.2

https://docs.oracle.com/javase/tutorial/uiswing/components/table.html

Nesta seção criaremos pequenos exemplos a fim de ilustrar o uso do componente. Na próxima seção o aplicaremos no projeto.

**2.7.1 (O exemplo mais simples)** O uso mais simples de um JTable envolve a especificação dos nomes de suas colunas, o que pode ser feito utilizando-se um vetor simples. A seguir, definimos uma matriz, responsável por armazenar os dados.

- Comece criando uma nova classe chamada TesteJTable1 com um método main. Defina um vetor de nomes de colunas como na Listagem 2.7.1.1.

Listagem 2.7.1.1

| String [] colunas = {"Nome", "Idade", "Sexo", "Endereço"}; |
| --- |

- A seguir, defina uma matriz de Object, responsável por armazenar alguns dados. Veja a Listagem 2.7.1.2.

Listagem 2.7.1.2

| Object [][] dados = {  {"José", 15, 'M', "Rua B, 23"},  {"Maria", 22, 'F', "Rua J, 255"},  {"Pedro", 34, 'M', "Rua K, 10"}  }; |
| --- |

- A classe JTable possui um construtor que recebe uma matriz de Object (com os dados) e um vetor de Strings (com os nomes das colunas), nesta ordem. A Listagem 2.7.1.3 ilustra seu uso.

Listagem 2.7.1.3

| JTable table = new JTable (dados, colunas); |
| --- |

- O código da Listagem 2.7.1.4 constrói um JFrame, adiciona a tabela a ele e faz os ajustes para sua exibição.

Listagem 2.7.1.4

| JFrame frame = new JFrame ();  frame.getContentPane().add(table);  frame.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);  frame.setLocationRelativeTo(null);  frame.pack();  frame.setVisible(true); |
| --- |

- Também pode ser de interesse exibir barras de rolagem caso a tabela possua muitos dados. Para isso, basta utilizar um JscrollPane, como na Listagem 2.7.1.5.

Listagem 2.7.1.5

| JTable table = new JTable (dados, colunas);  JScrollPane scrollPane = new JScrollPane(table);  JFrame frame = new JFrame ();  frame.getContentPane().add(scrollPane);  frame.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);  frame.setLocationRelativeTo(null);  frame.pack();  frame.setVisible(true); |
| --- |

- Faça os seguintes testes para conhecer algumas características que um JTable tem por padrão.

- Clique em uma célula: A linha inteira é destacada.

- Arraste uma coluna para a direita ou para a esquerda, clicando no seu nome.

- Clique duas vezes em uma célula: É possível editar seu conteúdo.

- Clique no separador entre duas colunas e arraste: é possível alterar a largura das colunas.

- Aumente o tamanho da tela e veja como as colunas se adaptam automaticamente, ocupando todo o espaço.

**2.7.2 (Criando um TableModel)** O primeiro exemplo é muito simples de se utilizar, porém tem muitas limitações. As características destacadas (todas as células editáveis, por exemplo) podem ser indesejáveis. Além disso, pode ser de interesse exibir os dados de uma coluna de uma forma diferente, de acordo com o seu tipo.

- Adicione uma nova coluna à coleção de dados do exemplo anterior. Ela indica se a pessoa é vegetariana com um valor Booleano. Adicione também o nome da coluna. Veja a Listagem 2.7.2.1.

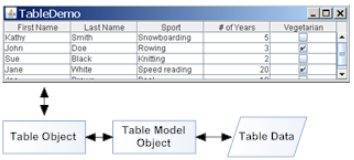
Listagem 2.7.2.1

| String [] colunas = {"Nome", "Idade", "Sexo", "Endereço", "Vegetariana(o)?"};    Object [][] dados = {  {"José", 15, 'M', "Rua B, 23", false},  {"Maria", 22, 'F', "Rua J, 255", false},  {"Pedro", 34, 'M', "Rua K, 10", true}  }; |
| --- |

- Execute o exemplo e veja que os valores booleanos são exibidos textualmente. Neste caso, poderia ser muito mais interessante exibir um checkbox. Para isso, precisamos criar um modelo de dados personalizado para a tabela.

- A forma como um JTable lida com um modelo de dados é ilustrada na Figura 2.7.2.1. Note que o modelo de dados é um intermediário entre o JTable (a parte gráfica) e os dados propriamente ditos (que até então estão armazenados em uma matriz de Object, mas que poderia ser qualquer estrutura de dados).

Figura 2.7.2.1



- Em geral, para criar um modelo de dados personalizado, escrevemos uma classe que herda de **AbstractTableModel** e fazemos a sobrescrita somente dos métodos de interesse. Como o nome sugere, ela é abstrata. Há três métodos abstratos que devem ser sobrescritos obrigatoriamente. Vamos manter o armazenamento de dados como no exemplo anterior neste momento. Veja a Listagem 2.7.2.2.

Listagem 2.7.2.2

| public class ExibicaoPersonalizadaTableModel extends AbstractTableModel {  public ExibicaoPersonalizadaTableModel (Object [][] dados, String [] colunas) {  this.dados = dados;  this.colunas = colunas;  }  private Object [][] dados;  String [] colunas;    @Override  public int getRowCount() {  return dados.length;  }  @Override  public int getColumnCount() {  return colunas.length;  }  @Override  public Object getValueAt(int rowIndex, int columnIndex) {  return dados[rowIndex][columnIndex];  }  } |
| --- |

- Para alterar a forma como O JTable exibe um elemento, basta fazer a sobrescrita do método **getColumnClass**. Ele devolve a classe do elemento. Por exemplo, para valores booleanos, esse método devolve a constante de classe Boolean. Assim o JTable decidirá sobre a melhor forma de exibi-lo. Veja a Listagem 2.7.2.3.

Listagem 2.7.2.3

| @Override  public Class getColumnClass(int columnIndex) {  return getValueAt(0, columnIndex).getClass();  } |
| --- |

- Para utilizar o modelo, basta instanciá-lo entregando-lhe os dados e colunas por meio de seu construtor. A seguir, o JTable pode ser instanciado recebendo o modelo como parâmetro. **Crie** **uma** **nova** **classe** para esse teste, chamada **TesteJTable2**. Veja a Listagem 2.7.2.4.

Listagem 2.7.2.4

| public class TesteJTable2 {  public static void main(String[] args) {  String [] colunas = {"Nome", "Idade", "Sexo", "Endereço", "Vegetariana(o)?"};    Object [][] dados = {  {"José", 15, 'M', "Rua B, 23", false},  {"Maria", 22, 'F', "Rua J, 255", false},  {"Pedro", 34, 'M', "Rua K, 10", true}  };      JTable table = new JTable  (new ExibicaoPersonalizadaTableModel(dados, colunas));  JScrollPane scrollPane = new JScrollPane(table);  JFrame frame = new JFrame ();  frame.getContentPane().add(scrollPane);  frame.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);  frame.setLocationRelativeTo(null);  frame.pack();  frame.setVisible(true);  }  } |
| --- |

- Note que os nomes das colunas ainda não foram utilizados. Outro detalhe é o fato de as células não serem editáveis. Podemos alterar essas características sobrescrevendo os métodos da Listagem 2.7.2.5.

Listagem 2.7.2.5

| @Override  public String getColumnName(int column) {  return this.colunas [column];  }    @Override  public boolean isCellEditable(int rowIndex, int columnIndex) {  return true;  }  public void setValueAt(Object value, int row, int col) {  dados[row][col] = value;  fireTableCellUpdated(row, col);  } |
| --- |

**2.7.3 (Exibindo um ComboBox em uma tabela)** Uma célula pode exibir uma coleção de opções em um JComboBox. Para isso, basta alterar o seu editor padrão. Implemente a classe da Listagem 2.7.3.1 para esse novo teste. Note que há uma **nova** **coluna** para que seja selecionado o esporte preferido de cada pessoa.

Listagem 2.7.3.1

| public class TesteJTable3 {  public static void main(String[] args) {  String [] colunas = {"Nome", "Idade", "Sexo", "Endereço", "Vegetariana(o)?", "Esporte"};    Object [][] dados = {  {"José", 15, 'M', "Rua B, 23", false, "Volei"},  {"Maria", 22, 'F', "Rua J, 255", false, "Futebol"},  {"Pedro", 34, 'M', "Rua K, 10", true, "Volei"}  };      TableModel model = new ExibicaoPersonalizadaTableModel(dados, colunas);  JTable table = new JTable (model);  setComboBox(table, model);  JScrollPane scrollPane = new JScrollPane(table);  JFrame frame = new JFrame ();  frame.getContentPane().add(scrollPane);  frame.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);  frame.setLocationRelativeTo(null);  frame.pack();  frame.setVisible(true);  }    public static void setComboBox (JTable table, TableModel model) {  TableColumn column = table.getColumnModel().getColumn(model.getColumnCount() - 1);  JComboBox <String> comboBox = new JComboBox <>();  comboBox.addItem ("Futebol");  comboBox.addItem("Basquete");  comboBox.addItem("Volei");  column.setCellEditor(new DefaultCellEditor(comboBox));  }  } |
| --- |

**2.8 (Configurando a JTable da tela MostraAlunosCursoTela)** Já adicionamos um JTable à tela que exibe alunos por curso. Agora é necessário realizar uma busca no banco para dizer quais dados ela deve exibir.

- Antes de mais nada, altere o nome da variável para **alunosTable**.

- A seguir, clique com o direito sobre a tabela e escolha **Table Contents >> User Specified**. Com esse ajuste estamos removendo os dados que o Netbeans coloca por padrão na tabela e estamos nos responsabilizando por eles.

- A busca devolverá uma lista de alunos associados ao curso escolhido. Por isso, precisamos escrever uma classe cuja finalidade é representar alunos. Veja a sua implementação na Listagem 2.8.1.

Listagem 2.8.1

| public class Aluno {  private int ra;  private String nome;  private int anoNascimento;  public Aluno(int ra, String nome, int anoNascimento) {  this.ra = ra;  this.nome = nome;  this.anoNascimento = anoNascimento;  }    //getters/setters  } |
| --- |

- A classe DAO é responsável por isolar o código de acesso à base. Por isso, cabe a ela implementar o método que busca os dados de alunos associados ao curso selecionado. Sua implementação é dada na Listagem 2.8.2.

Listagem 2.8.2

| public List <Aluno> buscaAlunosPorCurso (Curso curso) throws Exception{  String sql = "SELECT ra, nome, ano\_nascimento FROM tb\_aluno INNER JOIN tb\_aluno\_curso ON tb\_aluno.ra = tb\_aluno\_curso.ra\_aluno WHERE id\_curso = ?";  List <Aluno> alunos = new ArrayList <> ();  try (Connection conexao = ConexaoBD.obterConexao();  PreparedStatement ps = conexao.prepareStatement(sql)){  ps.setInt (1, curso.getId());  try (ResultSet rs = ps.executeQuery()){  while (rs.next()){  int ra = rs.getInt("ra");  String nome = rs.getString ("nome");  int anoNascimento = rs.getInt ("ano\_nascimento");  alunos.add(new Aluno (ra, nome, anoNascimento));  }  }  }  return alunos;  } |
| --- |

- Uma subclasse de AbstractTableModel pode ser utilizada para obter e representar adequadamente os dados de interesse, alimentando a JTable sempre que ela precisar. A classe da Listagem 2.8.3 representa os dados em uma coleção do tipo **ArrayList**, cujo funcionamento interno se baseia um vetor. Ocorre que seu método add se encarrega de verificar, antes de uma nova inserção, se o vetor já está cheio. Se já estiver, um novo é alocado para que o novo elemento caiba na coleção.

Listagem 2.8.3

| public class AlunosPorCursoTableModel extends AbstractTableModel {  private List <Aluno> alunos;  private String [] colunas = {"ra", "nome", "ano de nascimento"};    public AlunosPorCursoTableModel (Curso curso) throws Exception{  DAO dao = new DAO ();  this.alunos = dao.buscaAlunosPorCurso(curso);  }    @Override  public int getRowCount() {  return alunos.size();  }  @Override  public int getColumnCount() {  return 3;  }  @Override  public Object getValueAt(int rowIndex, int columnIndex) {  switch (columnIndex){  case 0:  return this.alunos.get(rowIndex).getRa();  case 1:  return this.alunos.get(rowIndex).getNome();  case 2:  return this.alunos.get(rowIndex).getAnoNascimento();  default:  return null;  }  }  @Override  public String getColumnName(int column) {  return this.colunas[column];  }  } |
| --- |

- O construtor da classe MostraAlunosCursoTela passa, portanto, a utilizar o TableModel. Veja a Listagem 2.8.4.

Listagem 2.8.4

| public MostraAlunosCursoTela(Curso curso) {  super("Alunos por Curso");  initComponents();  idCursoLabel.setText(Integer.toString(curso.getId()));  nomeCursoLabel.setText(curso.getNome());  tipoCursoLabel.setText(curso.getTipo());  setLocationRelativeTo(null);  try{  this.alunosTable.setModel(new AlunosPorCursoTableModel(curso));  }  catch (Exception e){  e.printStackTrace();  JOptionPane.showMessageDialog(null, "Falha técnica. Tente novamente mais tarde.");  }  } |
| --- |

***Referências***

DEITEL, P. e DEITEL, H. **Java Como Programar**. 8ª Edição. São Paulo, SP: Pearson, 2010.

ORACLE Java Tutorials. **Java Tutorials Learning Paths**. acesso em: <<https://docs.oracle.com/javase/tutorial/tutorialLearningPaths.html>>.