Módulo II
Interface com usuário – GUI Avançado

UniverCidade - Prof. Ismael H F Santos

Ementa

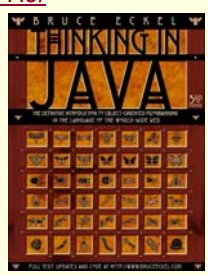
- Módulo II – Interface com o usuário (GUI)
  - MVC em Swing
    - Modelo MVC e MVC em Swing
    - JList e ListModel
    - JTree e TreeModel
    - JTable e TableModel
  - Threads e Swing

Bibliografia

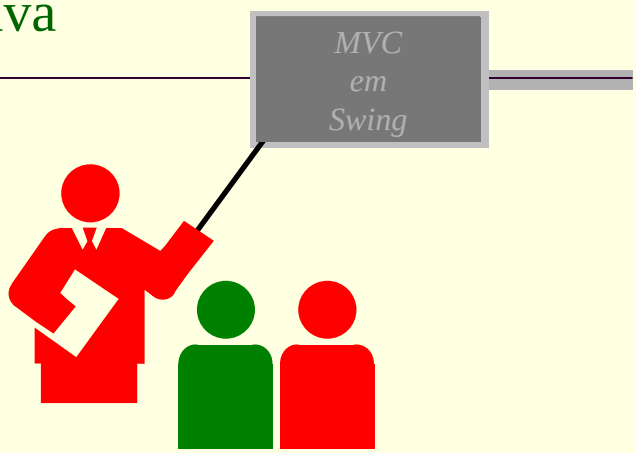
- *Linguagem de Programação JAVA*
  - *Ismael H. F. Santos, Apostila UniverCidade, 2002*
- *The Java Tutorial: A practical guide for programmers*
  - *Tutorial on-line: http://java.sun.com/docs/books/tutorial*
- *Java in a Nutshell*
  - *David Flanagan, O´Reilly & Associates*
- *Just Java 2*
  - *Mark C. Chan, Steven W. Griffith e Anthony F. Iasi, Makron Books.*
- *Java 1.2*
  - *Laura Lemay & Rogers Cadenhead, Editora Campos*

Livros

- **Core Java 2**, Cay S. Horstmann, Gary Cornell
  - Volume 1 (Fundamentos)
  - Volume 2 (Características Avançadas)
- **Java: Como Programar**, Deitel & Deitel
- **Thinking in Patterns with JAVA**, Bruce Eckel
  - **Gratuito**. http://www.mindview.net/Books/TIJ/



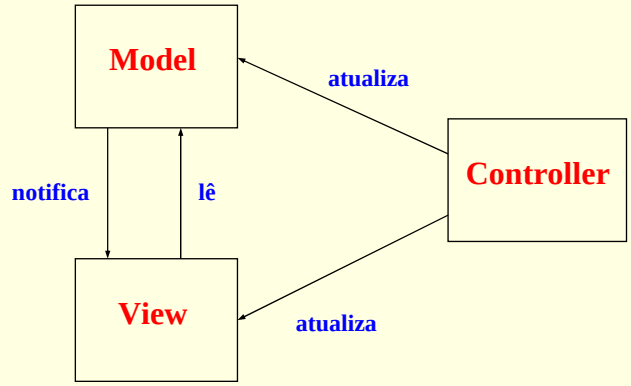
POO-Java



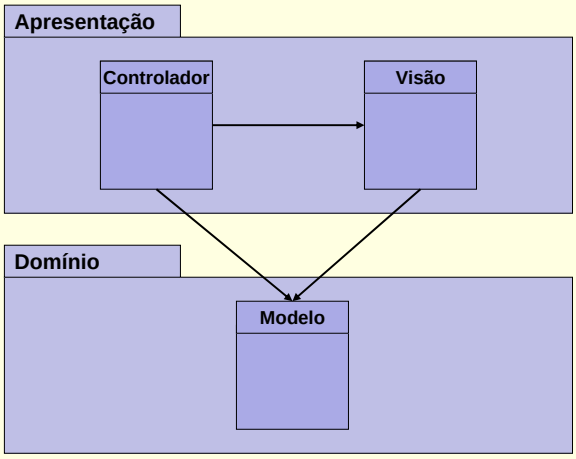
Arquitetura MVC

- O Swing adota uma arquitetura conhecida como *Model-View-Controller* (MVC)
  - Modelo = dados / conteúdo
    - estado de um botão, texto
  - Visão = aparência
    - cor, tamanho
  - Controle = comportamento
    - reação a eventos

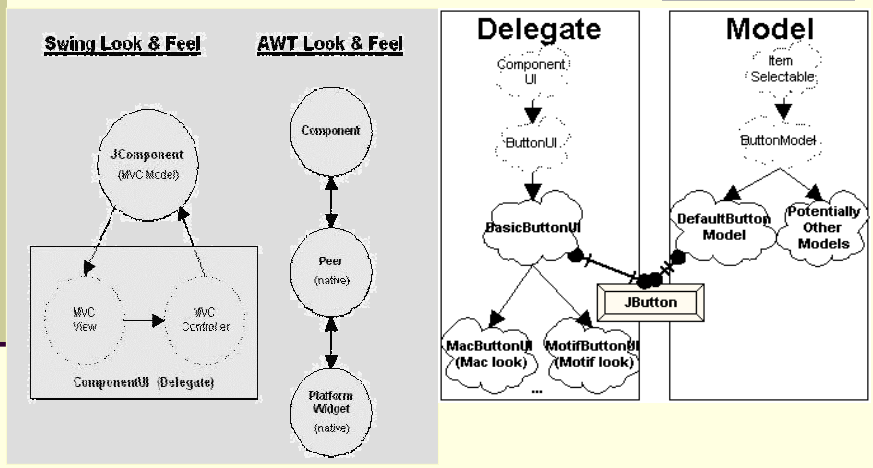
Interação entre os objetos MVC



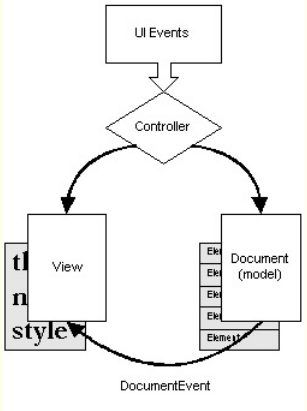
Arquitetura MVC no Swing



Modelo-Delegado



Documentos Swing



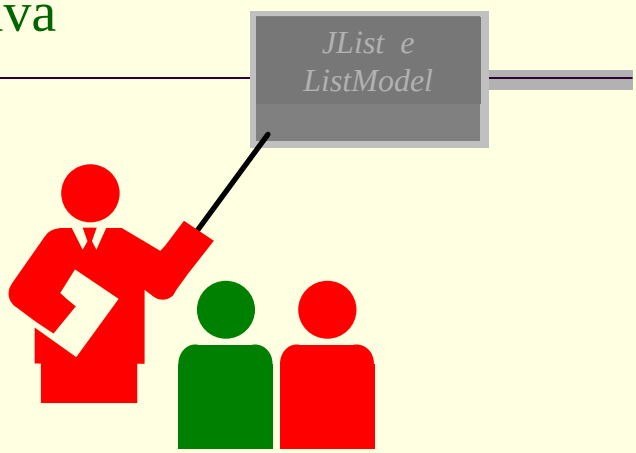
Explorando a Arquitetura MVC

- Como os dados (o modelo) não fazem parte integrante do elemento de interface que os exibe, podemos gerenciá-los em separado
- Por exemplo, é possível exibir um mesmo conjunto de dados em mais de um elemento de interface, simultaneamente
- Também é possível fazer com que o elemento de interface use os dados originais, sem copiá-los

Exemplo de Uso

- Suponha que você tem uma lista de nomes muito grande e deseja exibí-la em uma **JList**
- Usando a forma que vimos, esses nomes seriam copiados para dentro da lista
- Para evitar essa replicação, podemos utilizar um modelo próprio, que permitirá à **JList** acessar diretamente a lista de nomes

POO-Java



Interface **ListModel**

- Define o modelo usado pela classe **JList**
- Abrange dois aspectos:
  1. o acesso aos dados
  2. o controle da modificação dos dados

- Métodos de ListModel

```
int getSize()
Object getElementAt(int index)
void addListDataListener(ListDataListener l)
void removeListDataListener(ListDataListener l)
```

De Volta ao Exemplo

- Imagine que os nomes estão armazenados em um *array* de **String**
- Assumindo que a lista de nomes não é modificada, podemos ignorar o *listener*
- Basta, então, definir uma classe que implemente ListModel e utilize o *array* como fonte dos dados

Criando um Modelo

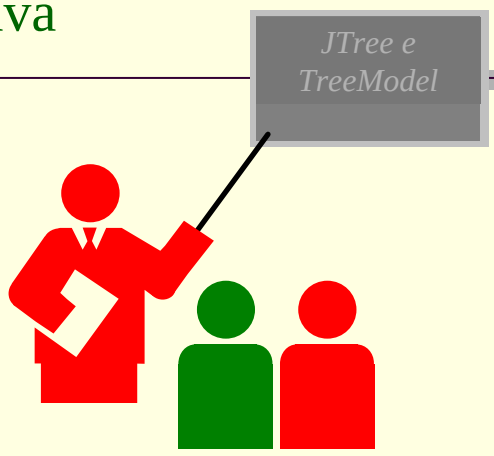
```
class ListaDeNomes implements ListModel {
  private String[] nomes;
  ListaDeNomes(String[] nomes) {
    this.nomes = nomes;
  }
  public int getSize() {
    return nomes.length;
  }
  public Object getElementAt(int index) {
    return nomes[index];
  }
  public void addListDataListener(ListDataListener l) {}
  public void removeListDataListener(ListDataListener l) {}
}
```

Usando o Modelo

```
JFrame f = new JFrame("Teste");
f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
String[] nomes = {"a", "b", "c", "d", "e", "f"};
JList l = new JList(new ListaDeNomes(nomes));
Container cp = f.getContentPane();
cp.add(new JScrollPane(l));
f.pack();
f.setVisible(true);
```

Exercícios – Questão 24 (again) - *Exemplo com DefaultListModel !*

POO-Java



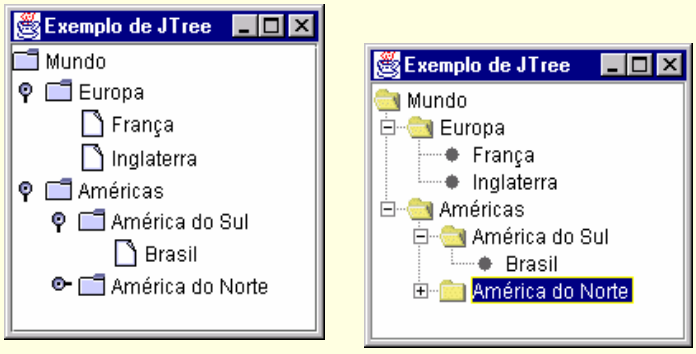
Classe **JTree**

- Componente que exibe uma estrutura de dados hierárquica (árvore)
- Segue o padrão MVC: os dados a serem exibidos são obtidos de um modelo (**TreeModel**)
  - o modelo a ser utilizado é fornecido no construtor do objeto **JTree**

Terminologia

- Uma árvore é composta de nós
  - um nó ou é uma folha ou possui nós filhos
  - todo nó, com exceção da raiz, tem exatamente um nó pai
  - toda árvore tem exatamente um nó raiz
- Tipicamente, o usuário pode ***expandir*** ou ***colapsar*** nós, tornando seus filhos, respectivamente, visíveis ou invisíveis

Exemplos



Interface **TreeModel**

- Define um modelo de dados adequado para um JTree
- Pertence ao pacote **javax.swing.tree**
- O Swing oferece uma implementação dessa interface: a classe **DefaultTreeModel**
  - modelo de árvore que utiliza TreeNodes

Métodos de **DefaultTreeModel**

```
DefaultTreeModel(TreeNode root)

Object getRoot()
int getChildCount(Object parent)
Object getChild(Object parent, int index)

void insertNodeInto(MutableTreeNode child,
                    MutableTreeNode parent,
                    int index)
void removeNodeFromParent(MutableTreeNode node)

void addTreeModelListener(TreeModelListener l)
```


Interface **MutableTreeNode**

- É uma subinterface de **TreeNode**
- Modela um nó que pode ser modificado
  - adição/remoção de filhos
  - modificação do conteúdo armazenado no nó (“user object”)
- O Swing oferece uma implementação dessa interface: a classe **DefaultMutableTreeNode**

Métodos de **DefaultMutableTreeNode**

```
DefaultMutableTreeNode(Object userObject)
DefaultMutableTreeNode(Object userObject,
                              boolean allowsChildren)
void add(MutableTreeNode child)
void remove(MutableTreeNode child)

Object getUserObject()
void setUserObject(Object userObject)
String toString()

boolean isLeaf()
Enumeration children()
```


Criando um JTree

```
DefaultMutableTreeNode mundo =
      new DefaultMutableTreeNode (“Mundo”);
DefaultMutableTreeNode europa =
      new DefaultMutableTreeNode (“Europa”);
DefaultMutableTreeNode americas =
      new DefaultMutableTreeNode (“Américas”);
mundo.add(europa);
mundo.add(americas);
…
JTree arvore = new JTree(new DefaultTreeModel(mundo));
```

Modos de Seleção

- O modo de seleção de um JTree é configurado (e gerenciado) por um "modelo de seleção" (**TreeSelectionModel**)
- Modos disponíveis:
  - SINGLE_TREE_SELECTION
  - CONTIGUOS_TREE_SELECTION
  - DISCONTIGUOUS_TREE_SELECTION

Configurando o modo de seleção

- Configurando modo de seleção

```
JTree arvore = new JTree(raiz);
int modo = TreeSelectionModel.SINGLE_TREE_SELECTION;
TreeSelectionModel tsm = arvore.getSelectionModel();
tsm.setSelectionMode(modo);
```

- Obtendo a seleção corrente

```
TreePath path = getSelectionPath()
if (path != null) {
 DefaultMutableTreeNode selNode =
(DefaultMutableTreeNode)path.getLastPathComponent();

String selValue = (String)selNode.getUserObject();
 …
}
```

Eventos de Seleção

- Eventos de seleção são gerados sempre que a seleção de uma árvore é alterada.
- Esses eventos podem ser tratados através da adição de um **TreeSelectionListener**.
- A interface **TreeSelectionListener** pertence ao pacote javax.swing.event e define apenas um método: `valueChanged`

Exercícios – Questão 27
Exercícios – Questão 28

POO-Java

JTablee e TableModel

JTable

- Utilizada para vizualizar dados em grid no Swing, classes adicionais definidas em javax.swing.table
- Modelo MVC
  - **Model**
    - implementado pela interface TableModel (AbstractTableModel e DefaultTableModel)
    - responsavel por fornecedor os dados para a tabela, através do método **Object getValueAt(row, col)**
  - **View**
    - implementado pela interface CellRenderer. Pode ser fornecido tanto um renderer para a tabela inteira como para uma coluna específica.
    - Para desenhar uma celula a view requisita o objeto que irá apresentar os dados pelo método JComponent getCellRendererComponent**(row, col).** O DefaultCellRenderer usa JLabel para apresentar os dados, que é a forma de apresentação mais comum para um valor.

JTable

- Modelo MVC (cont.)
  - **Controller**
    - É a parte que controla a apresentação dos dados na view.
- Exemplo:

```
String[] colunas = new String []{"Estado",
                                 "Cidade"};
String[][] dados = new String [][] {
        {"SP","Sao Paulo"},
        {"RJ","Rio de Janeiro"},
        {"RN","Rio Grande do Norte"},
        {"PR","Parana"}
    };
JTable jtable = new JTable(dados, colunas);
```

JTable (outro exemplo)

```
String[] columnNames = {
   "First Name", "Last Name", "Sport", "# of Years", "Vegetarian“ };
Object[][] data = {
   {"Mary", "Campione", "Snowboarding", new Integer(5), new Boolean(false)},
   {"Alison", "Huml", "Rowing", new Integer(3), new Boolean(true)},
   {"Kathy", "Walrath", "Knitting", new Integer(2), new Boolean(false)},
   {"Sharon", "Zakhour", "Speed reading", new Integer(20), new Boolean(true)},
   {"Philip", "Milne", "Pool", new Integer(10), new Boolean(false)} };
JTable table = new JTable(data, columnNames);
JScrollPane scrollPane = new JScrollPane(table);
table.setPreferredScrollableViewportSize(new Dimension(500, 70));
```

TableModel

- O constutor de JTable usado anteriormente constrói um TableModel assim:

```
new AbstractTableModel() {
  public String getColumnName(int col) {  return columnNames[col].toString(); }
  public int getRowCount() { return rowData.length; }
  public int getColumnCount() { return columnNames.length; }
  public Object getValueAt(int row, int col) {  return rowData[row][col]; }
  public boolean isCellEditable(int row, int col){ return true; }
  public void setValueAt(Object v, int row, int col) { rowData[row][col]=v;
     fireTableCellUpdated(row, col); }
}
```

- Criando a customTableModel

```
public TableDemo() { ... JTable table = new JTable(new MyTableModel()); ... }
class MyTableModel extends AbstractTableModel {
  private String[] columnNames = ...//same as before...
  private Object[][] data = ...//same as before...
```

Demo JavaWebStart

DefaultTableModel

- Tendo JTable criada, podemos trabalhar em cima do modelo (TableModel) que ela criou através do método de JTable TableModel getModel()
- Métodos da interfaceTableModel (DefaultTableModel)
  - getValueAt(row, col): obtem o valor de uma determinada linha e coluna na JTable
  - setValueAt(newValue, row, col): seta o valor em uma determinada linha e coluna na JTable.
  - addRow(Object[] row): adiciona uma nova linha na JTable, recebe um array simples.
  - addColumn(Object[] col): adiciona uma nova coluna no modelo.
  - removeRow(int row): remove linha “row” da tabela

DefaultTableModel

- Exemplo:

```
String[] colunas = new String []{"Estado",
                                 "Cidade"};
String[][] dados = new String [][] {
        {"SP","Sao Paulo"},
        {"RJ","Rio de Janeiro"},
        {"RN","Rio Grande do Norte"},
        {"PR","Parana"}
    };
// Colocamos os dados em um modelo
DefaultTableModel modelo = new DefaultTableModel(dados, colunas);
// e passamos o modelo para criar a jtable
JTable jtable = new JTable(modelo);
```

Adicionando e removendo linhas

- Exemplo (cont)

```
public void adicionaLinha() {
  DefaultTableModel m = (DefaultTableModel)getTabela().getModel();
  m.addRow( new String [] {"", ""} ); // Adiciona linha em branco no modelo
}
public void removeLinha(int row) {
  DefaultTableModel m = (DefaultTableModel)getTabela().getModel();
  m.removeRow(row); // // Remove a linha do modelo
}
public void removeLinha() {
  int linhaSelecionada = getTabela().getSelectedRow();
  if( linhaSelecionada < 0 ){
    return;
  } else {
    removeLinha(linhaSelecionada);
  }
}
```


Adicionando e removendo linhas

- ListSelection
  - existem métodos para obter nao só a linha/coluna, mas o numero de linhas/colunas selecionadas e quais foram as linhas/colunas selecionadas (retorna num array de int[ ]).
  - Use o método jtable.setSelectionMode(ListSelectionModel.SINGLE_SELECTION) para permitir somente a seleção de uma única linha e não multiplas.
- getValueAt e setValueAt
  - Usados para obter/substituir o conteúdo de uma celula, ou de várias linhas em uma determinada coluna;
- Iterator
  - utilizado para trabalhar em cima dos dados da tabela. O iterator() varre o vetor de linhas dos dados.

```
Iterator i = modelo.getDataVector().iterator();
String [] row = i.next();
```

Adicionando e removendo linhas

- Exemplo (cont)

```
/** Método para subtituir um valor por outro em uma determinada coluna.
* @param oldValue   @param newValue  @param column
* @return Numero de substituições */
  public int substituirValor (String oldValue, String newValue, int column){
     int total = 0;  // Flag para saber se algum valor foi
     DefaultTableModel m = (DefaultTableModel)getTabela().getModel();
     // Looping em cima das linhas do modelo
     for( int i=0; i< m.getRowCount(); i++){
        // Obtem o valor atual na coluna
        String valorAtual = (String)m.getValueAt(i, column);
        if( valorAtual.equals(oldValue) ){
            // Substitui pelo novo valor na linha e coluna
            m.setValueAt(newValue, i, column);
            // Adiciona mais um no numero de linhas atualizadas
             total++;
        }
     }
     return total;
  }
```


Custom TableModel

- Para implementar o próprio TableModel, devemos tratar o controle dos dados (colunas e linhas)
- Opção simples para começar, é a de estender a classe AbstractTableModel que fornece diversos métodos já sobrescritos exceto aqueles que retornam dados:
  - public Object getValueAt(int row, int col)
  - public int getRowCount()
  - public int getColumnCount()
- AbstractTableModel também já implementa 7 métodos de atualização da JTable (TableModelEvents). Estes métodos são utilizados para informar as alterações ocorridas dentro de um modelo. Por exemplo, a inclusão de uma nova linha dispara o método fireTableRowsInserted

Custom TableModel

- A classe abstrata AbstractTableModel implementa diversas funcionalidades da TableModel: incluindo os eventos que sinalizam para a JTable alterações ocorridas no modelo;
- O modelo de dados funciona como um repositório dos dados. Por ter um conteúdo dinâmico que pode ser alterado em tempo de execução: (exemplo remover ou incluir novas linhas), nunca podemos saber o tamanho exato maximo que o vetor de linhas dos dados terá. O melhor jeito é usar uma Collection para armazená-lo.
- No próximo exemplo vamos usar a classe java.util.ArrayList para armazenar os dados da linha e um array de String para colunas.

Custom TableModel

- Exemplo 2

```
public class SimpleTableModel extends AbstractTableModel {
    private ArrayList linhas = null;
    private String [] colunas = null;
    public String[] getColunas() {return colunas;}
    public ArrayList getLinhas() {return linhas;}
    public void setColunas(String[] strings) {colunas = strings;}
    public void setLinhas (ArrayList list) {linhas = list;}
}
public int getColumnCount() {return getColunas().length;}
public int getRowCount() {return getLinhas().size();}
public Object getValueAt(int rowIndex, int columnIndex) {
  String [] linha = (String [])getLinhas().get(rowIndex);
  return linha[columnIndex];
}
public SimpleTableModel(ArrayList dados, String[] colunas){
    setLinhas(dados);
    setColunas(colunas);
}
```

Custom TableModel

```
public JTable createJTable() {
    ArrayList dados = new ArrayList();
    String[] colunas = new String[] { "Estado", "Cidade" };

    // Alimenta as linhas de dados - ArrayList
    dados.add(new String[] { "SP", "São Paulo" });
    dados.add(new String[] { "RJ", "Rio de Janeiro" });
    dados.add(new String[] { "RN", "Rio Grande do Norte" });
    dados.add(new String[] { "ES", "Espirito Santo" });

    SimpleTableModel m = new SimpleTableModel(dados, colunas);
    Table jtable = new JTable(m);
    jtable.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);

    return jtable;
}
```

- O código acima apresentar JTable com os dados, mas não irá permitir a sua alteração, diferentemente de DefaultTableModel. Isso porque não implementamos o método isCellEditable, que é herdando de AbstractTableModel, cuja implementação retorna false para todas as células.

Alterando conteudo

- Para permitir a edição da célula, e que seu valor venha atualizar o modelo devemos implementar os seguintes métodos:

```
public boolean isCellEditable (int row, int col) { return true; }
public void setValueAt (Object value, int row, int col) {
    // Obtem a linha, que é uma String []
    String [] linha = (String [])getLinhas().get(row);
    // Altera o conteudo no indice da coluna passado
    linha[col] = (String)value;
    // dispara o evento de celula alterada
    fireTableCellUpdated (row,col);
}
```

- O método setValueAt deve informar a JTable a alteração do modelo através do método fireTableCellUpdated.

Inserindo células

- Para incluir novas linhas adicionamos um novo objeto no array de linhas e avisar a JTable a inclusão de nova linha com o método fireTableRowsInserted(row-i, row-f)

```
public void addRow( String [] dadosLinha ) {
    getLinhas().add(dadosLinha);
    // Informa a jtable de que houve linhas incluidas no modelo
    // Como adicionamos no final, pegamos o tamanho total do modelo
    // menos 1 para obter a linha incluida.
    int linha = getLinhas().size()-1;
    fireTableRowsInserted(linha,linha);
    return;
}
```


Excluindo células

- A exclusão de linhas é feita diretamente pelo modelo nos dados e o método fireTableRowsDeleted(row-i, row-f) avisa a JTable
- Remove linha

```
public void removeRow( int row ) {
    getLinhas(). remove(0);
    fireTableRowsDeleted(row, row);
    return;
}
public boolean removeRow(String val, int col) {
      Iterator i = getLinhas().iterator(); int row = 0;
     while( i.hasNext() ) {
        String[] rowCorrente = (String[])i.next();  row++;
         if( rowCorrente[col].equals(val) ){ // procura valor na posicao (row,col)
               getLinhas().remove(row);
              fireTableRowsDeleted(row, row);
              return true;
         }
    }
    return false;
 }
```

Observações

- Dentro do modelo, não importa como você armazene seus dados. Não interessa para a JTable se buscou na hora direto do seu banco, ou se está tudo em um array.
- O importante é fornecer o valor através dos métodos, e informar das alterações ocorridas no seu modelo seguindo a estrutura do padrão MVC.