UNIVERSIDADE DO ESTADO DE SANTA CATARINA
DCC – Departamento de Ciência da Computação
Joinville – SC

CURSO: Bacharelado em Ciência da Computação
DISCIPLINA: POO0001 – Programação Orientada a Objetos
PROFESSOR: Diego Buchinger
AULA 09 – Introdução a Interfaces Gráficas em Java – Swing

1. INTERFACE GRÁFICA NO JAVA

A implementação de interface gráfica utilizando linguagens um pouco mais antigas como C e C++ é um processo que pode ser bastante trabalhoso, principalmente se não for utilizada uma API externa que simplifique o processo. De fato, programar interações com uma janela gráfica usando as ferramentas mais básicas como OpenGL e DirectX pode ser um desafio para programadores iniciantes e até mesmo para pessoas experientes. Neste sentido, a linguagem de programação Java conta com uma API básica que simplifica o processo de criação de janelas e dos principais elementos que são utilizados nela, como botões, *labels* e espaço para entrada de dados. Essa API nativa é chamada de GUI (*Graphical User Interface*) Swing.

A fim de criarmos janelas gráficas de interação com o usuário, criaremos classes que irão abstrair os elementos e eventos que ocorrem em uma determinada janela. Para simplificar a criação de um esboço inicial de janela podemos utilizar os modelos padrões básicos que a IDE Netbeans nos disponibiliza. Para tanto, podemos acessar a opção Arquivo (no menu superior), então em Novo Arquivo e, na nova janela que aparece, selecionamos a categoria "Forms GUI Swing", conforme apresentado na Figura 1. Ao selecionar tal categoria, serão listados alguns tipos de arquivos padrão disponíveis. No momento, vamos utilizar o formulário ou janela de mais alto nível que seria o "Form JFrame". Pressione o botão Próximo, renomeie a classe apropriadamente e finalize o processo de criação.

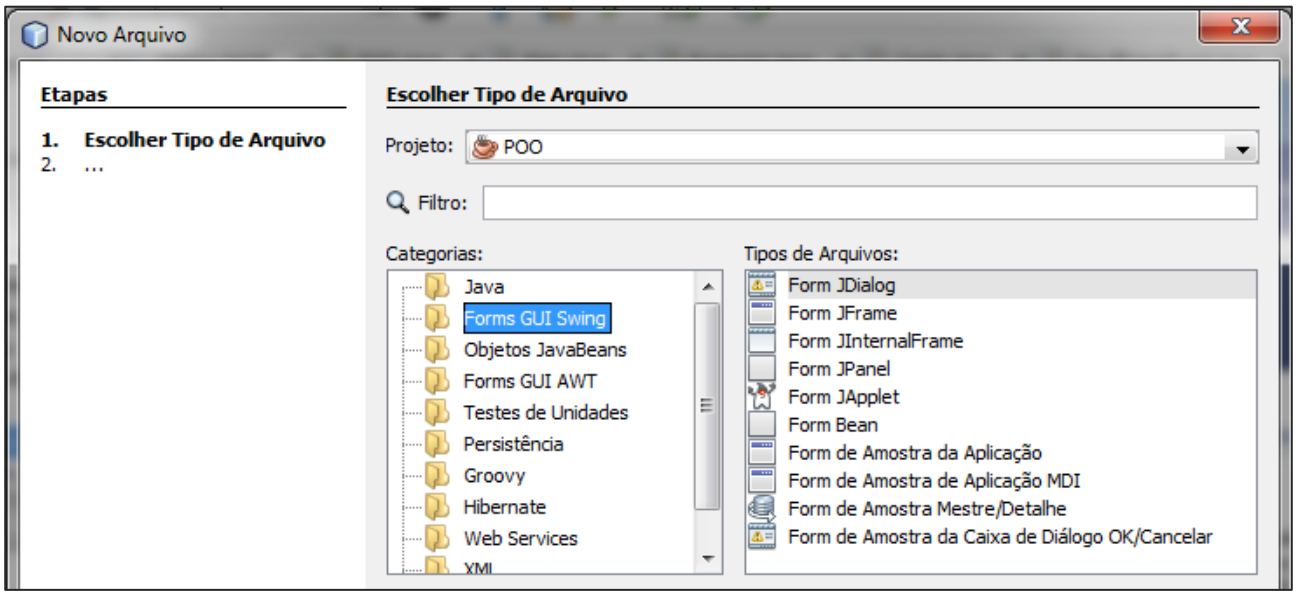
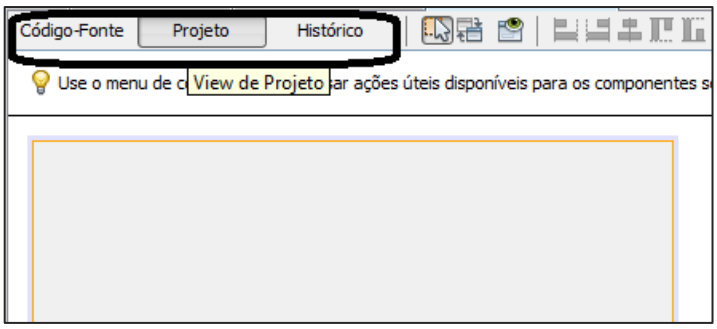


FIGURA 1 – Tipos de Arquivos para criação de Formulários Gráficos usando a biblioteca Swing

Ao finalizar o processo de criação de um JFrame, você deve perceber que o Netbeans agora passa a apresentar três opções de edição para esta nova classe: (1) Código-Fonte, (2) Projeto e (3)



Histórico. A opção (2) Projeto, só aparece para classes de formulários que foram criados internamento no Netbeans. Na verdade, a IDE cria um arquivo auxiliar de mesmo nome do arquivo (.java), com uma extensão (.form). Dentro deste arquivo estão as definições da organização da janela / formulário, permitindo fácil posicionamento de elementos na janela com a abordagem arrastar-e-soltar.

Primeiramente vamos ver o que já foi gerado de código-fonte automaticamente para exibir uma janela sem conteúdo. Acesse a opção "Código-Fonte" citada anteriormente. Procure pela definição da classe da janela criada. Ela deve ser mais ou menos assim:

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

Perceba a utilização de herança. A nossa nova classe está herdando (ou estendendo/especializando) a classe JFrame que vem do pacote nativo *swing* dentro do pacote *javax*. Se você preferir é possível adicionar uma instrução import para esta classe, sendo possível especificar em código apenas a classe JFrame.

Um pouco mais abaixo da definição da classe está a definição de um método construtor para a nossa janela. É um construtor que não recebe parâmetros e invoca apenas um método chamado *initComponents()*. Este método é gerado automaticamente para cada modificação realizada na região de "Projeto". A sua definição aparece logo abaixo do método construtor, mas costuma ficar oculta na IDE Netbeans, sendo necessário expandir a região clicando sobre o + que aparece ao lado do número da linha. Ao expandir esta seção do código você poderá vislumbrar todas as operações básicas de criação de objetos e chamadas de métodos para inicializar os componentes da janela. Esta parte do código-fonte não é editável, pois está atrelada às definições da janela e qualquer modificação pode ter um impacto sério sobre a pré-visualização do formulário.

```
16      /**
17       * This method is called from withi
18       * WARNING: Do NOT modify this code
19       * regenerated by the Form Editor.
20       */
21      @SuppressWarnings("unchecked")
22      Generated Code
40
```

Agora procure mais para o final do código a definição do método *main()*, o qual permite a execução do programa, criando uma instância da nossa nova classe e mostrando o seu conteúdo. Nas versões mais novas do NetBeans, existe uma primeira parte do código que tenta primeiramente definir o estilo de layout da janela no novo padrão chamado de Nimbus. Este estilo diz respeito à forma ou modelo com que a janela em si e os elementos são desenhados. Caso o padrão não esteja acessível devido à versão da JVM, então se utiliza o padrão antigo (a Figura 2 mostra a diferença entre os dois modelos de estilos utilizados).

Na sequência do método *main()* temos uma forma diferente e mais atual de invocar um novo thread envolvida na parte gráfica, a qual faz uso da classe de fila de eventos (EventQueue) – uma classe gerenciadora de eventos – e de seu método *invokeLater()* para iniciar um novo thread. O thread em questão é bem simples e por isso é definida em linha (*in line*), especificando a criação de um objeto que implementa a interface Runnable e o seu método *run()* que é efetivamente o que será

feito pelo thread. Dentro deste método *run()* temos a criação de um objeto da nossa janela seguido pelo uso do método *setVisible(true)* que faz com que a janela se torne visível ao usuário.

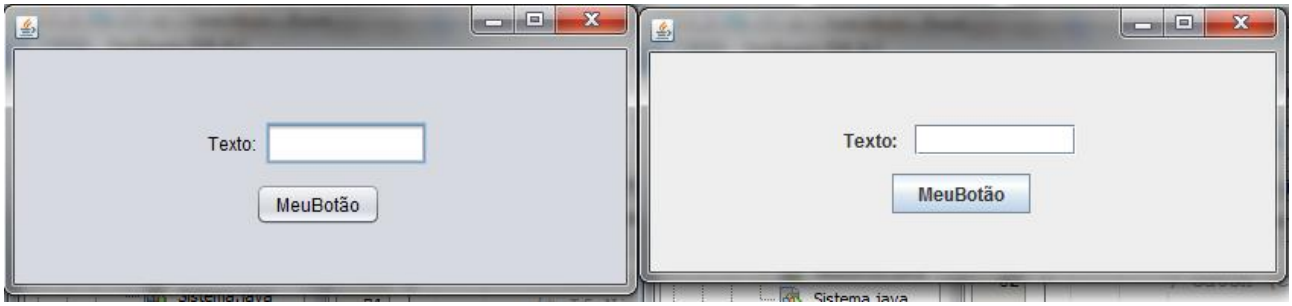


FIGURA 2 – Comparação entre estilos de interface gráfica no Java: Nimbus (esquerda) e tradicional (direita).

2. CRIANDO UM FORMULÁRIO

Vamos começar a colocar a mão na massa? Com o intuito de fazer uma dinâmica mais prática, considere que nosso objetivo será construir uma primeira janela gráfica que faz o papel de formulário de entrada de dados para cadastramento de dados de pessoas em uma agenda eletrônica. Antes de começarmos a trabalhar em cima da Janela propriamente dita, adicione duas classes que modelam os dados necessários para representar o registro de uma pessoa na aplicação proposta. As classes *Person* e *Address* são apresentadas nos quadros a seguir.

Com ambas as classes de estrutura de dados adicionadas no projeto, vamos iniciar a geração do layout visual do formulário. O primeiro passo será selecionar a opção de visualização do "Projeto", tendo a classe da janela aberta no editor. Caso seja de sua preferência, renomeie a classe da janela de acordo com o que vamos usar neste material: *FormAddPerson*. Para ajudar na localização dos elementos que serão utilizados, observe a Figura 3 que destaca três áreas importantes neste processo de criação do layout das janelas. Primeiramente utilize a janela Paleta (destacada pelo retângulo azul) encontrando os elementos de tela label (classe JLabel) e campo de texto (classe JTextField). Posicione os elementos de forma similar à Figura 4.

Arquivo: Person.java

```
 1.  package interfaceGrafica;
 2.
 3.  class Person {
 4.      private String name, phone;
 5.      private Address address;
 6.
 7.      public Person(String name, Address address, String phone){
 8.          this.name = name;
 9.          this.address = address;
10.          this.phone = phone;
11.      }
12.
13.      public void setName(String name){ this.name = name; }
14.      public void setPhone(String phone){ this.phone = phone; }
15.      public void setAddress(Address addr){ address = addr; }
16.      public String getName(){ return name; }
17.      public String getPhone(){ return phone; }
18.      public Address getAddress(){ return address; }
19.  }
```

Arquivo: Address.java

```
 1.  package interfaceGrafica;
 2.
 3.  public class Address{
 4.      private String street, district;
 5.      private int number;
 6.
 7.      public Address(String street, int number, String district){
 8.          this.street = street;
 9.          this.number = number;
10.          this.district = district;
11.      }
12.
13.      public void setStreet(String st){ this.street = st; }
14.      public void setDistrict(String dist){ district = dist; }
15.      public void setNumber(int num){ number = num; }
16.      public String getAdress(){
17.          return street + ", " + number + ", " + district;
18.      }
19.  }
```

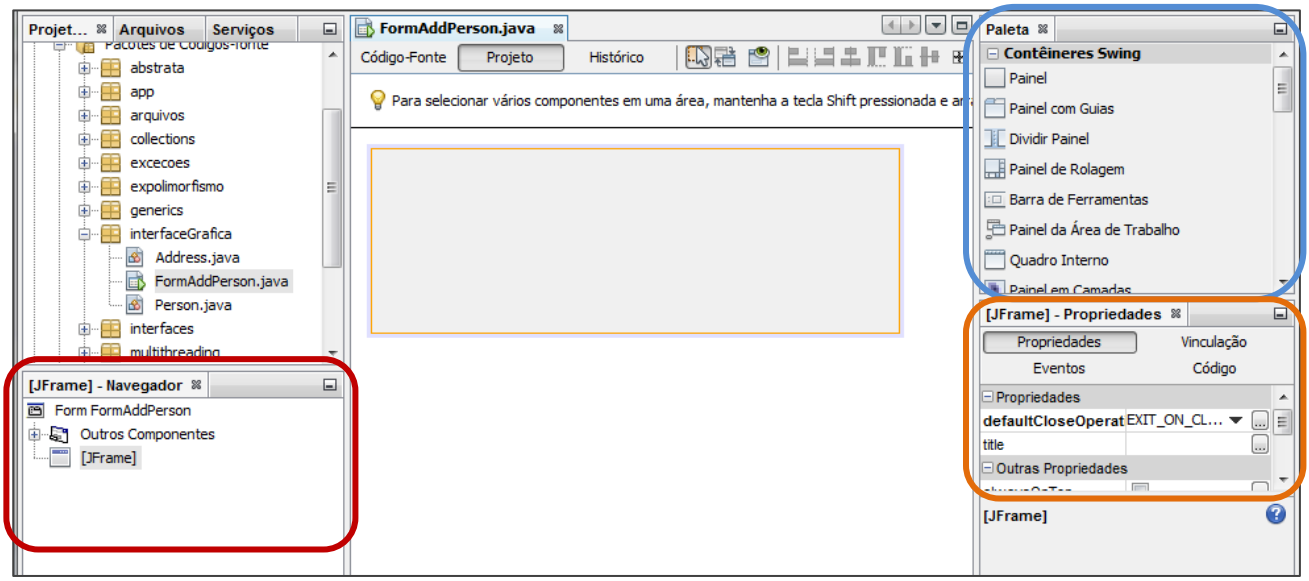


FIGURA 3 – Edição de um Formulário em Java: o retângulo vermelho demarca a região que exibe os objetos vinculados a janela que foram criados, o retângulo azul demarca os principais contêineres disponíveis na GUI Swing, e o retângulo laranja demarca região de propriedades do elemento selecionado.

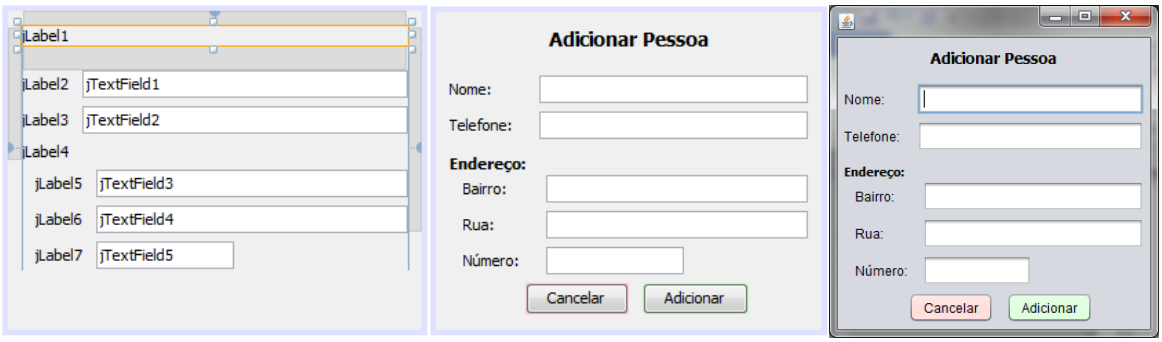


FIGURA 4 – Etapas de edição do formulário de inserção de um registro de pessoa. Posicionamento de alguns elementos (esquerda), alteração dos valores e edição de propriedades (centro) e janela renderizada durante a execução do programa.

Seguem algumas dicas de edição de formulário utilizando o Netbeans:

- Ao adicionar elementos gráficos, procure utilizar as linhas guias para realizar o posicionamento dos elementos. Elas são úteis para ajustar o tamanho dos elementos quando a janela é redimensionada. Se você tiver dificuldades no início, não se preocupe, pois o uso e ajuste destas linhas guias podem realmente aborrecer o programador;
- Para alterar o texto de um objeto visual, selecione-o e pressione a tecla F2;
- Os objetos visuais podem ter seu tamanho ajustado após serem selecionados. Isso pode ser feito utilizando os quadrados das bordas do elemento ou a janela de propriedades (retângulo alaranjado apresentado na Figura 3);
- Selecione os objetos visuais e altere suas propriedades pela janela de propriedades (retângulo alaranjado da Figura 3). Diversas propriedades estão disponíveis neste menu, sendo que elas mudam dependendo do tipo de elemento visual selecionado;
- É usual deixar os campos de caixa de texto vazios, contudo, antes de apagar o conteúdo, redimensione o tamanho de alguma forma, pois ao não fazer isso o campo pode encolher e ficar com tamanho mínimo, dificultando seu ajuste posterior;
- Ao clicar duas vezes (duplo clique) sobre um elemento visual do projeto de layout do formulário, você está utilizando um atalho que adiciona um evento de interação com aquele objeto (evento actionPerformed). Esta ação criará um método automaticamente na região do código da sua classe da janela. Para remover um evento criado por engano, deve-se voltar ao modo de visualização do formulário, clicar uma única vez no objeto que recebeu o evento de forma indevida. Então, na janela de propriedades, deve-se clicar em "Eventos" e pressionar o botão "..." da primeira linha, referente ao evento actionPerformed. Na janela que se abre, selecione o evento criado e pressione o botão remover;
- Renomeie os nomes dos elementos gráficos mais importantes da sua janela através da janela "Navegador" (retângulo vermelho apresentado na Figura 3). Por padrão, o Netbeans utiliza valores padronizados e sequenciais para cada elemento.

Note que na Figura 4 já foram adicionados também dois botões (classe JButton) para a nossa janela. Esses botões tiveram a sua propriedade background-color definida como verde e laranja, por isso aparecem nestas cores na execução do programa. Para manter o mesmo padrão e ordem de nomes, renomeie os elementos de campo de texto e os botões da seguinte maneira: *inputName*, *inputPhone*, *inputDistrict*, *inputStreet*, *inputNumber*, *buttonAdd*, *buttonCancel* (note que os nomes são bastante sugestivos).

O próximo passo será criar um evento para o botão Adicionar e outro evento para o botão Cancelar. Para isto, podemos proceder de duas formas: (1) efetuar duplo clique sobre cada botão (um por vez) adicionando um evento actionPerformed ou, (2) selecionar um dos botões, selecionar Eventos na janela de propriedades, encontrar o evento "*mouseClicked*" e digitar o nome do novo evento no campo textual onde consta "<nenhum>" (exemplo: *cancelClicked*). O código do evento criado será similar ao exemplo mostrado abaixo:

```
private void cancelClicked(java.awt.event.MouseEvent evt) {
    // TODO add your handling code here:
}
private void addActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
}
```

Para alterar o nome de um evento criado com duplo clique, deve-se proceder de forma similar ao que foi feito para excluir um evento (ver dicas de edição apresentadas).

Agora precisamos escrever o código que será executado quando os botões forem pressionados. Vamos começar com o botão "Adicionar"! Quando este botão for pressionado queremos construir um objeto *Person* que representa o registro de pessoa que está sendo inserido e adicioná-lo a um repositório de *Person's* (container ou vetor), que será mantido por uma classe isolada chamada de Main.

A primeira pergunta que deve surgir é: "como eu faço para recuperar o que foi digitado no campo de texto da minha janela?". Como os elementos da janela nada mais são do que botões, vamos usar o nome destes objetos e invocar um método denominado de *getText*(), o qual retorna uma String com o conteúdo digitado. Note que para valores numéricos é necessário fazer uma conversão, pois o valor será retornado como uma String. De forma similar, podemos utilizar o método *setText(String)* que define/altera o valor do campo de texto. Vamos utilizar este segundo método para apagar o conteúdo que já foi adicionado (na verdade vamos alterar o valor para String vazia). Sem os devidos tratamentos de erros teríamos um código similar à:

Arquivo: FormAddPerson.java

```
package interfaceGrafica;

public class FormAddPerson extends JFrame {
   //...
   private void addClicked(java.awt.event.ActionEvent evt) {
      String district = inputDistrict.getText();
      String street = inputStreet.getText();
      int number = 0;
      try{
          number = Integer.parseInt( inputNumber.getText() );
      } catch(NumberFormatException ex){
          //... tratar o erro aqui!
      }
      Address address = new Address( street, number, district );
      String name = inputName.getText();
      String phone = inputPhone.getText();
      Person person = new Person( name, address, phone );
      boolean result = Main.addContact( person );
      if( result )
          clearInputFields();
      else
          //... tratar o erro aqui!
   }


   private void clearInputFields(){
      inputName.setText("");
      inputPhone.setText("");
      inputDistrict.setText("");
      inputStreet.setText("");
      inputNumber.setText("");
   }
   //...
}
```

Arquivo: Main.java

```
package interfaceGrafica;
import java.util.ArrayList;

public class Main {
    static ArrayList<Person> contactBook = new ArrayList< >();

    // This method adds a new Person in the contact book
    public static boolean addContact( Person p ){
        return contactBook.add(p);
    }
}
```

Bem, até o momento o nosso formulário já pode ser utilizado para adicionar os registros, mas não é feito tratamento de erro algum caso algum problema ocorra, como a digitação incorreta de um valor não numérico no campo número (endereço). Poderíamos adicionar um diálogo com o usuário avisando que houve um erro, ou então que o registro foi adicionado com sucesso. Para fazer isso, no entanto, não seria muito interessante utilizar a classe System e mostrar as mensagens no console. Seria melhor utilizar algum mecanismo de janela. Neste sentido, poderíamos: (1) criar uma nova janela personalizada, exatamente como fizemos para este nosso formulário, o que é um pouco mais trabalhoso, ou, (2) utilizar a classe de "pop-ups" do Java JOptionPane.

3. JANELAS DE POP-UP NO JAVA SWING

A biblioteca Swing possui uma definição de classe específica para janelas pop-up que facilitam a criação de janelas de conversação rápida com o usuário. Esta classe possui diversos métodos que permitem a criação de diferentes tipos de janelas de dialogo com o usuário. Elas são:

```
JOptionPane.showMessageDialog( parent, msg, title, msgType, icon );

JOptionPane.showConfirmDialog( parent , message, title, optionType,
                               messageType, icon );

JOptionPane.showInputDialog( parent, message, title, messageType );

JOptionPane.showInputDialog( parent, message, title, messageType,
                             icon, object[], initialOpt );
```

Cada um dos métodos apresentados acima é sobrecarregado com várias versões nas quais nem todos os parâmetros são necessários. Considerando as versões apresentadas acima, os parâmetros especificados tem o seguinte significado:

- *parent*: é um objeto da classe *Component* e representa a qual componente o pop-up deve estar associado (centralizado). Se o valor nulo for passado, então a janela fica associada ao programa como um todo.
- *message*: é um objeto String que representa o texto que será exibido no pop-up.
- *title*: é um objeto String que representa o título que será exibido no pop-up.

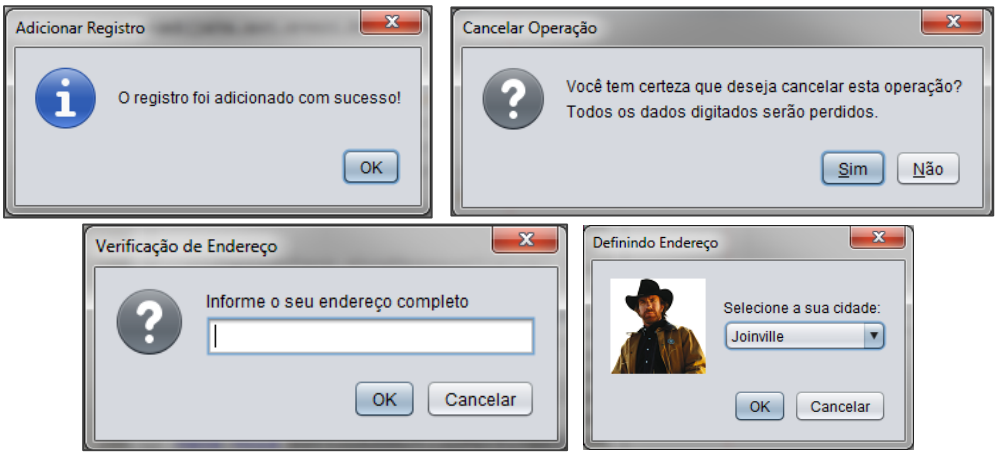


FIGURA 5 – Janelas de pop-up exibidas com auxílio da classe JOptionPane. Mensagem de diálogo ou aviso (superior esquerda), mensagem de confirmação ou pergunta (superior direita), mensagem com entrada de dados textual (inferior esquerda), e mensagem com entrada de dados por opção (inferior direita).

- *messageType*: é um valor numérico que representa o tipo de mensagem que o pop-up exibirá. A classe JOptionPane já possui constantes pré-estabelecidas que devem ser utilizadas: `JOptionPane.ERROR_MESSAGE;` `JOptionPane.INFORMATION_MESSAGE;` `JOptionPane.PLAIN_MESSAGE;` `JOptionPane.QUESTION_MESSAGE;`
- *icon*: é um objeto da classe Icon que especifica qual o ícone que será utilizado ao lado da mensagem, substituindo o ícone padrão escolhido pelo parâmetro *messageType*. Se um valor nulo for passado, o ícone padrão será utilizado. A instanciação de um ícone é bastante simples: `Icon icone = new Icon("imagem.png");` mas exige a importação das bibliotecas: import javax.swing.ImageIcon; e import javax.swing.Icon;
- *optionType*: é um valor numérico que representa o tipo de janela de confirmação que deve ser exibido ao usuário. A classe JOptionPane já possui constantes pré-estabelecidas que devem ser utilizadas: `JOptionPane.OK_CANCEL_OPTION;` `JOptionPane.YES_NO_OPTION;` `JOptionPane.YES_NO_CANCEL_OPTION;`
- *object[]:* um vetor de objetos que identificam quais as opções que estarão disponíveis ao usuário. É comum utilizar um vetor de Strings, ou algum objeto que tenha implementado o método toString(), a fim de exibir uma opção bem descrita;
- *initialOpt:* um objeto identificando qual é a opção que deve estar selecionada inicialmente.

Com exceção do primeiro método, *showMessageDialog()*, que não possui um retorno, os demais métodos retornam qual foi a escolha do usuário perante o diálogo. No método *showConfirmDialog()*, pode-se receber como retorno um inteiro que identifica qual foi a opção escolhida. Para verificar isto, devem-se utilizar as constantes definidas na própria classe: `JOptionPane.YES_OPTION;` `JOptionPane.NO_OPTION;` `JOptionPane.CANCEL_OPTION`. Já o método *showInputDialog()* retorna uma String ou um Object, dependendo da versão utilizada. Lembre-se que o objeto selecionado é uma referência para o objeto que foi escolhido pelo usuário.

Enfim, com o conhecimento destas janelas de pop-up, podemos agora complementar o código-fonte da nossa janela tratando erros e mostrando mensagens ilustrativas e interativas. Por fim, é importante destacar que poderíamos também realizar verificações dos demais campos do formulário a fim de garantir valores válidos. Isso poderia ser feito diretamente na classe da janela ou, preferencialmente, na definição das classes que são instanciadas, usando de exceções.

Arquivo: FormAddPerson.java (adaptação)

```
public class FormAddPerson extends JFrame {
    //...
    private void addClicked(java.awt.event.ActionEvent evt) {
        //...
        try{
            number = Integer.parseInt( inputNumber.getText() );
        } catch(NumberFormatException ex){
            JOptionPane.showMessageDialog( this , "Verifique o valor"
                + "digitado no campo número no registro desta nova"
                + "pessoa.", "Adicionar Registro",
                JOptionPane.ERROR_MESSAGE);
            return;
        }
        //...
        boolean result = Main.addContact( person );
        if( !result ){
              JOptionPane.showMessageDialog( this , "Houve um erro "
                  + "no momento de adicionar o novo registro!\n"
                  + "A estrutura não suporta a adição do elemento.",
                  "Adicionar Registro",
                  JOptionPane.ERROR_MESSAGE);
              return;
          }
          clearInputFields();
          JOptionPane.showMessageDialog( this , "O registro foi "
              + "adicionado com sucesso!", "Adicionar Registro",
              JOptionPane.INFORMATION_MESSAGE);
    }
    //...
}
```

4. PROPRIEDADES DAS JANELAS

Para deixarmos o nosso projeto mais interessante é necessário entender as propriedades básicas das janelas JFrame. Primeiramente quanto às propriedades principais das janelas, elas podem ser definidas no modo de visualização de "Projeto" no Netbeans. Para tanto, devemos selecionar a janela com um clique simples sobre ela, ou então, selecionando o elemento JFrame na janela do Navegador (retângulo vermelho apresentado na Figura 3). Com a janela JFrame selecionada, podemos visualizar suas propriedades no painel de propriedades (retângulo laranja apresentado na Figura 3). Entre as propriedades estão:

- *defaultCloseOperation*: define o que deve ser feito quando a janela é fechada:
  - `HIDE`: a janela continua aberta, mas é ocultada;
  - `EXIT_ON_CLOSE`: o programa é finalizado;
  - `DO_NOTHING`: faz nada, a janela permanece visível como se nada tivesse ocorrido;
  - `DISPOSE`: fecha a janela liberando os recursos ocupados por ela.
- *title*: define o texto que é apresentado na barra de título da janela;
- *location*: localização na qual a janela será apresentada na tela para o usuário;
- *maximumSize* e *minimumSize*: tamanho máximo e mínimo que a janela pode assumir;
- *resizable*: determina se a janela pode ser redimensionada ou não.

Uma propriedade que merece atenção é a cor de fundo da janela. A princípio existe a propriedade *background* que permite a definição de uma cor de plano de fundo, mas esta pode não alterar efetivamente a cor de fundo da janela. Para fazer isto é necessário adicionar um elemento gráfico de painel (*JPanel*) e definir a propriedade *background* deste elemento para a cor desejada. Note que uma vez que os elementos da janela já foram posicionados, seria necessário reposicionar todos os elementos sobre o novo painel.

Por fim, se quisermos adicionar um evento próprio e específico ao fechar uma janela, podemos definir um método que é invocado antes da janela efetivamente ser fechada. Para tanto, primeiro precisamos alterar *defaultCloseOperation* para `DO_NOTHING`. Em seguida devemos selecionar o elemento JFrame, então selecionar a opção de "Eventos" na janela de propriedades, procurar pelo evento "*windowClosing*" e adicionar um nome de método. Dessa maneira um novo método vinculado ao evento de fechamento da janela será adicionado no código fonte. Um diálogo de fechamento de janela poderia ser:

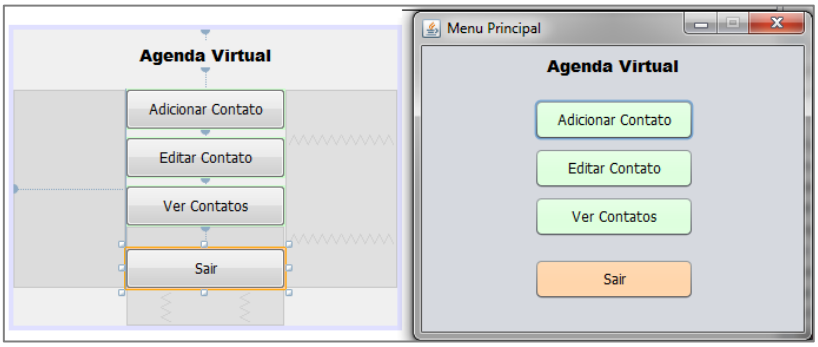
Arquivo: FormAddPerson.java (adaptação)

```
public class FormAddPerson extends JFrame {
    //...
    private void windowClosing(java.awt.event.WindowEvent evt) {
        int answer = JOptionPane.showConfirmDialog(null, "Deseja
           realmente fechar esta janela?\n Todos os dados
           digitados serão perdidos.", "Fechar Formulário",
           JOptionPane.YES_NO_OPTION, JOptionPane.QUESTION_MESSAGE);
        if( answer == JOptionPane.YES_OPTION )
            this.dispose();
    }
    //...
}
```

Por fim, para completar o código deste nosso primeiro formulário, poderíamos vincular um novo método para o evento de pressionar o botão "Cancelar" que poderia inclusive chamar o método *windowClosing* passando como parâmetro um valor nulo já que o evento não é utilizado pelo método em si.

4.1. Vinculando Janelas

Bem, o nosso formulário está pronto, mas ele não tem vínculo algum com outras janelas. O próximo passo será criar novas janelas gráficas que estão amarradas umas nas outras. Para tanto, teremos que gerenciar a criação e possível ocultamento de janelas. Para fazer um exemplo de vinculação de janelas vamos criar um menu principal para o nosso sistema. Tente construir um novo formulário (nome da classe: MainMenu) com a seguinte aparência/layout:



Em seguida, vamos adicionar dois eventos para este formulário, um para o botão “Sair” (*closeClicked*) e outro para o botão “Adicionar Contato” (*addPersonClicked*). Para o botão “Sair” iremos adicionar uma chamada ao método *closeDialog()* que realiza um diálogo com o usuário a fim de confirmar a intenção de encerrar o software. Este método adicional *closeDialog()* pode ser invocado também no evento de fechamento do formulário (conforme adição de evento de fechamento da janela visto anteriormente). Já para o botão “Adicionar Contato”, adicionaremos uma instanciação de um objeto do formulário FormAddPerson que foi criado anteriormente. Por padrão, a instanciação de um objeto de formulário não o exibe em tela. Para fazer isso, precisamos usar o método *setVisible()* do objeto formulário, passando o argumento *true*. Isso pode ser feito após a instanciação de um objeto deste tipo, ou então, poderíamos adicionar a linha: *this.setVisible(true);* no construtor do formulário, fazendo com que, sempre que um formulário é instanciado, ele é automaticamente exibido em tela.

Arquivo: MainMenu.java

```
public class MainMenu extends JFrame {
    //...
    private void closeClicked(java.awt.event.MouseEvent evt) {
        closeDialog();
    }

    private void closeDialog(){
        int answer = JOptionPane.showConfirmDialog(null,
          "Deseja realmente sair do programa?", "Fechar Programa",
          JOptionPane.YES_NO_OPTION, JOptionPane.QUESTION_MESSAGE);
        if( answer == JOptionPane.YES_OPTION )
            System.exit(0);
    }

    private void addPersonClicked(java.awt.event.MouseEvent evt) {
        FormAddPerson formAdd = new FormAddPerson();
        formAdd.setVisible( true );
    }
    //...
}
```

O código acima realiza o vínculo prometido, mas perceba que ele não oculta a janela do menu principal que já estava sendo exibida. Antes de qualquer coisa, perceba que a janela de menu principal provavelmente será aberta e fechada diversas vezes durante o uso do software. Considerando isto, qual das seguintes abordagens você acha que vale mais a pena:

(1) Instanciar um novo objeto de formulário para exibi-lo e finalizá-lo, gastando processamento, mas liberando a memória, a cada vez que o formulário for requisitado;

(2) Instanciar apenas uma vez a janela de formulário para exibi-la, e quando ela não estiver ativa apenas ocultá-la, a fim de poupar processamento, mas ocupando memória.

Para este caso de formulário, em específico, a segunda abordagem parece mais vantajosa. Logo, vamos atualizar o código da nossa classe MainMenu de tal forma que esta classe fique responsável por garantir a criação de apenas um formulário de menu principal e que possa controlar a sua visibilidade, alternando seu estado entre visível e não visível. Para isto, introduzimos um objeto estático do próprio formulário MainMenu que inicialmente possui um valor nulo, além de dois novos métodos estáticos também – um para instanciar o formulário pela primeira vez, caso seja

necessário, e mostra-lo na tela e outro para ocultar o formulário se ele já estiver instanciado. Agora podemos utilizar o segundo método, aquele que oculta o formulário, no nosso método de evento *addPersonClicked()*. Além disso, precisamos alterar ligeiramente o nosso método *main()* que é responsável por mostrar a janela do menu principal quando o software é inicializado. Para isto, alteramos o código do thread que inicializa a parte gráfica.

Arquivo: MainMenu.java (atualização)

```
public class MainMenu extends JFrame {
    private static MainMenu mainMenuForm = null;

    /** Apenas a própria classe pode instanciar novo formulário */
    private MainMenu() { initComponents(); }
    public static void showForm(){
        if( mainMenuForm == null )   // formulário não foi criado
            mainMenuForm = new MainMenu(); // cria nova instância
        mainMenuForm.setVisible(true);     // mostra na tela
    }
    public static void hideForm(){
        if( mainMenuForm != null )
            mainMenuForm.setVisible(false);
    }
    //...
    private void addPersonClicked(java.awt.event.MouseEvent evt) {
        FormAddPerson formAdd = new FormAddPerson();
        formAdd.setVisible( true );
        MainMenu.hideForm();
    }
    //...
    public static void main(String args[]) {
       //...
       java.awt.EventQueue.invokeLater(new Runnable() {
          public void run() {
             MainMenu.showForm();
          }
       });
    //...
}
```

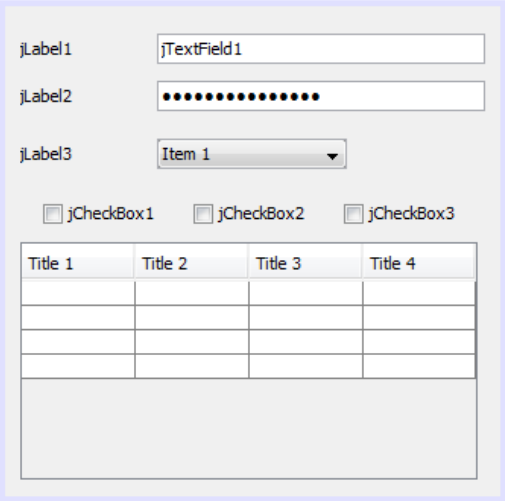
Para finalizar precisamos a ligação entre as janelas, precisamos ainda alterar o código do nosso primeiro formulário, de modo que quando esta janela for fechada ou então a operação cancelada, o formulário do menu principal seja mostrado novamente em tela, ANTES de liberar todos os recursos que o formulário de FormAddPerson estava usando – this.dispose();

Arquivo: FormAddPerson.java (adaptação)

```
public class FormAddPerson extends JFrame {
   //...
   private void windowClosing(java.awt.event.WindowEvent evt) {
       if( answer == JOptionPane.YES_OPTION ){
           MainMenu.showForm();
           this.dispose();
       }
   }
   //...
}
```

5. ALGUNS OUTROS ELEMENTOS DA JAVA SWING

Existem muitos detalhes a cobrir neste tópico referente a interfaces gráficas. Para finalizar este material, serão apresentados a seguir outros quatro elementos básicos de formulários que podem ser úteis em diversas situações. A intenção é que, com o conhecimento "mastigado" destes componentes, você seja capaz de buscar e entender a forma de trabalhar com outros elementos de interface gráfica.



5.1. Campo de Senha

O elemento de campo de senha – campo de texto onde os caracteres ficam ocultos – é um elemento implementado pela classe `JPasswordField`, que possui muitas semelhanças com o elemento campo de texto já visto, mas também possui as suas peculiaridades. A mais importante, que vale salientar, é que para resgatar o texto deste tipo de campo, recomenda-se o uso do método *getPassword()* que retorna um *array* de caracteres, ao invés de um objeto String. A documentação desta classe ainda reforça que, após o uso deste *array*, é importante para aspectos de segurança alterar todos os valores nele contido para não deixar resquícios da senha digitado.

5.2. Caixa de Combinação

Caixa de Combinação é outro elemento bastante usual em formulários. Estes são implementados pela classe `JComboBox`, que permite especificar quais são os itens da caixa de seleção, geralmente Strings. Após instanciar um objeto de caixa de combinação, deve-se definir qual modelo será utilizado. Por padrão, pode-se utilizar um objeto `DefaultComboBoxModel` que recebe em seu construtor um *array* de String, que identificam as opções disponíveis. Estas opções, entretanto, não são imutáveis; elas podem ser alteradas posteriormente com os métodos: `removeItemAt(int);` `removeAllItems();` e `addItem(String);` por exemplo.

Para verificar quando há uma alteração na seleção de um determinado elemento na caixa de combinação, deve-se adicionar um novo evento: *itemStateChanged*. No entanto, este evento é disparado duas vezes sempre que um novo item é selecionado – uma vez indicando que o antigo item foi "desmarcado" e outro indicando que um novo item foi selecionado. Para filtrar qual evento foi disparado e executar uma determinada operação apenas uma vez, pode-se utilizar o parâmetro do método como mostrado no quadro a seguir. Por fim, para verificar qual o item selecionado, podemos utilizar pelo menos dois métodos: `getSelectedIndex();` que retorna qual o índice do elemento selecionado, e `getSelectedItem();` que retorna o valor do item selecionado.

5.3. Caixa de Seleção

Caixas de Seleção são implementadas pela classe `JCheckBox`, a qual possui dois métodos principais para verificar ou definir o seu estado: `isSelected();` que retorna um valor booleano indicando se a caixa está marcada ou não, e `setSelected( boolean );` que marca ou

desmarca a caixa de seleção em acordo com o valor do booleano passado por parâmetro. Pode-se também adicionar um evento que dispara um método quando o estado de uma caixa de seleção é alterado. O evento em questão é o *itemStateChanged* (CUIDADO para não se confundir e usar o evento *stateChanged* que é ativado sempre que houver qualquer interação com a caixa de seleção, inclusive passar o mouse sobre este elemento gráfico).

JComboBox

```
jComboBox1.setModel( new javax.swing.DefaultComboBoxModel<>(
              new String[] { "Item 1", "Item 2", "Item 3", "Item 4"
} ));


private void itemStateChanged(java.awt.event.ItemEvent evt) {
   if( evt.getStateChange() == ItemEvent.SELECTED ){
      System.out.print( jComboBox1.getSelectedIndex() + " " );
      System.out.println( jComboBox1.getSelectedItem() );
   }
}
```

5.4. JTable

Por fim, um elemento que é muito útil para listar diversos registros em uma tela gráfica é a tabela, que na API *swing* é implementada pela classe `JTable`. Provavelmente este elemento é um dos que possui mais detalhes dentre aqueles que vimos. O primeiro passo para utilizar uma tabela deste tipo é instanciar um objeto `JTable`. A seguir, deve-se criar e atrelar um objeto auxiliar que define o modelo da tabela, isto é, como a tabela será exibida, quais serão as colunas e quais serão os valores das linhas. O modelo de tabelas padrão da API *swing* é o `table.DefaultTableModel`, o qual recebe como parâmetros de construção um *array* bidimensional de objetos – que representam os valores das linhas – e um *array* simples de String – que representam os nomes dos cabeçalhos das colunas. Os valores das linhas não precisam necessariamente ser preenchidos neste momento, podendo ser atribuído um *array* multidimensional vazio inicialmente (ou seja, zero linha).

O código referente à inicialização de uma `JTable` é gerado automaticamente pelo gerador de códigos da ferramenta de formulários do Netbeans. Para alterar a propriedade das colunas pela ferramenta de formulários, pode-se editar a propriedade *model* editando o número de linhas e colunas e especificando as colunas da tabela. A Figura 6 ilustra onde esta propriedade e a janela que simplifica estas modificações. Note que existe uma coluna denominada “editável”, que indica se os campos desta coluna podem ser editados ou não. O código gerado neste processo segue o padrão mostrado no quadro de código abaixo:

JComboBox

```
import javax.swing.JTable;
import javax.swing.table.DefaultTableModel;
import javax.swing.table.TableColumn;
//...
DefaultTableModel model = new DefaultTableModel(
   new Object [][] {},
   new String [] {"Nome", "ID", "RG", "Descrição"}
));
jTable1.setModel( model );
```

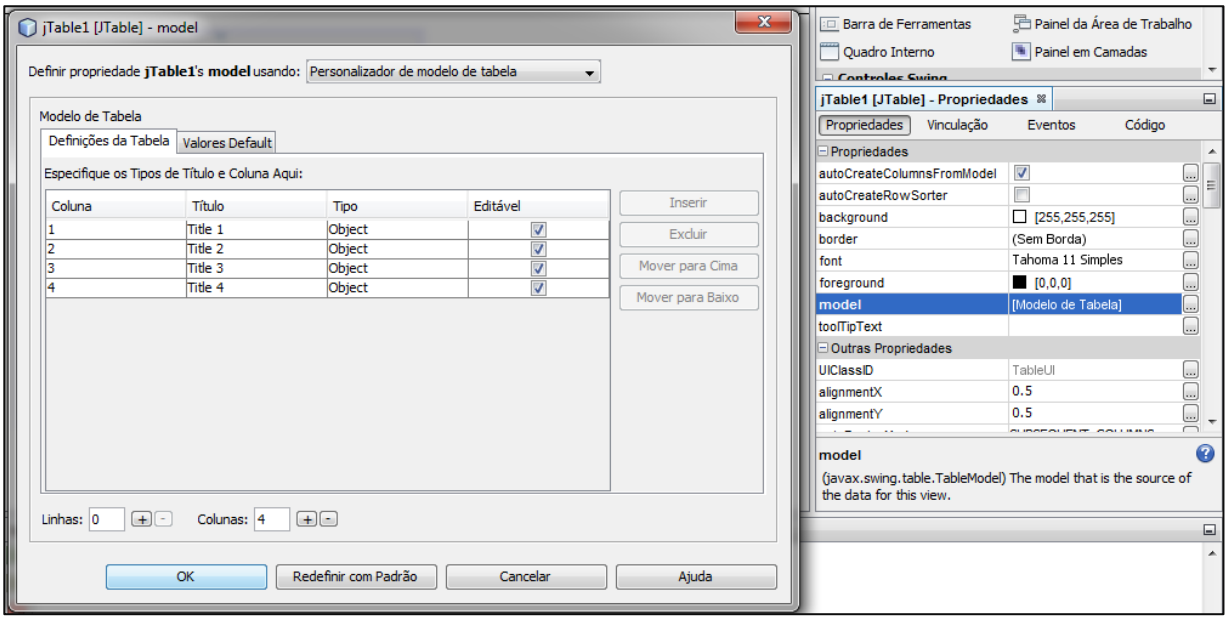


FIGURA 6 – Propriedade “model” de um elemento JTable no editor de formulários do Netbeans e sua respectiva janela de edição de propriedades.

Como uma tabela pode ser maior do que o tamanho da janela é interessante sempre associá-la a um painel de *scroll*. Tal objeto é implementado na API swing pela classe `JScrollPane`. Para efetivar o vínculo entre tabela e o painel de scroll devemos utilizar o método `setViewportView`.

```
jScrollPane1.setViewportView(jTable1);
```

Para alterar as principais propriedades de uma tabela, alteramos o seu modelo. Para exemplificar, podemos definir a largura preferida de uma determinada coluna recuperando o modelo de colunas da tabela, selecionando então o índice da coluna e seu valor da largura:

```
jTable1.getColumnModel().getColumn(0).setPreferredWidth( 150 );
```

Para alterar as linhas de conteúdo da tabela, devemos resgatar o modelo da tabela através do método *getModel();* que retorna um objeto que implementa a interface `TableModel`. Normalmente as `JTable` são criadas utilizando-se o modelo padrão `DefaultTableModel`, mas como isso é apenas uma possibilidade, devemos antes de mais nada garantir isto usando o operador *instanceof*. Feito isso, podemos fazer um *casting* seguro deste `TableModel` para um `DefaultTableModel`. Através deste objeto, podemos utilizar dois métodos de edição de linhas:

- `removeRow( int )`: método utilizado para remover uma linha especifica;
- `addRow( Object[] )`: método utilizado para adicionar uma linha ao final. O tamanho do *array* deve ser compatível com o número de colunas.

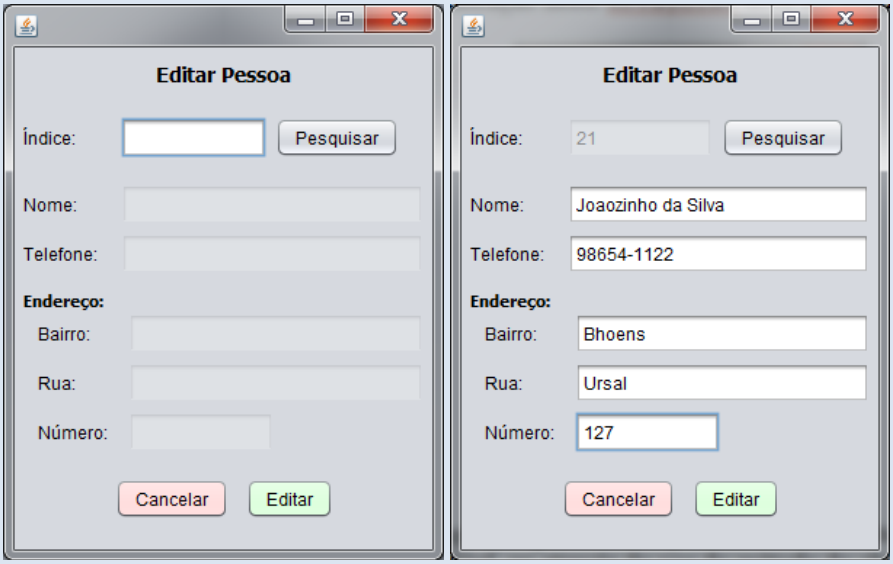
JComboBox

```
if( jTable1.getModel()  instanceof  DefaultTableModel ){
  DefaultTableModel model = (DefaultTableModel) jTable1.getModel();
  model.addRow( new Object[]{ "Cavalo", 11, 500, "Alasan" }  );
  model.removeRow( 0 );
}
```

Exercícios

1) Termine o projeto da agenda eletrônica criando os dois formulários restantes: "Ver Contatos" e "Editar Contato". No formulário de edição de contato, deve-se criar algum mecanismo para que os dados de um determinado contato sejam recuperados e possam ser editados, sobrescrevendo os valores antigos. Pode-se utilizar o modelo em que o usuário deve digitar inicialmente o índice do contato que deseja editar e pressiona um botão. Isto dispara um evento que procura pelo registro de pessoa com o índice informado e preenche os campos de texto, permitindo a edição (ver Figura abaixo).

Lembre-se que os dados dos indivíduos já cadastrados são mantidos em um `ArrayList` (*contactBook*) na classe `Main`. Será necessário criar outros métodos para acessar este `ArrayList` de tal forma que seja possível editar e resgatar os dados ali presentes. Toda modificação neste `ArrayList` deve ser realizada nesta classe.



OBS: perceba os campos bloqueados. Isto pode ser feito no editor de formulários através da propriedade *enabled*, ou através do uso do método do objeto de campo de texto:

```
setEnabled( boolean );
```