# Capítulo 5

Java é muito mais do que linhas de comando e textos. Com o Java Swing deixaremos nossas aplicações mais atraentes e modernas. Aplicações que utilizam linhas de comando como interface com o usuário (que são as aplicações que criamos até então) foram bastante utilizadas antes da criação do mouse e hoje são utilizadas por programadores. Interfaces Gráficas enriquecem a aplicação e dão mais valor a ela. Neste capítulo você irá aprender como criar telas profissionais para aplicações Java. Além disso você verá que até em interfaces gráficas utiliza-se orientação a objetos.

## Aula 1 - Definindo as Views

E agora estamos chegando ao fim, nesta unidade finalizaremos o projeto com os detalhes finais que faltam ao nosso conhecimento. Vamos começar?!

Após definirmos as regras, funcionalidades e Models do aplicativo, as Views devem ser desenhadas.

Conforme dito na aula anterior, as Views são a interface com o usuário. Nela definimos a aparência do aplicativo, exibimos as informações e a coletamos do usuário. Estas informações exibidas e coletadas são processadas pelos Controllers, que iremos definir um pouco mais para frente.

*DICA: Um passo importante para que não haja retrabalho e para que se obtenha resultados rápidos e descartáveis (pois se o protótipo desenhado não agradar podemos descartá-lo sem muito custo) é a prototipagem.*

### 1.1 - Protótipo das Views

O intuito dos protótipos é validar de uma maneira rápida como os itens de interface se organizarão na tela. Para isto, não podemos perder muito tempo com ferramentas de design e edição de imagens. O ideal é que os primeiros protótipos sejam feitos usando papel e caneta grossa.

Dica: caneta grossa faz com que desenhemos mais rápido e com poucos detalhes.

Geralmente em uma empresa, esta tarefa não cabe ao programador, então desenhamos um protótipo que já foi validado para você. Foi definido que nosso app terá duas telas. Uma de visão geral e outra de Cadastro/Edição de contatos. Abaixo segue a tela de visão geral do protótipo que foi desenhado para você.

#### imgs/i1.png

Figura 5. – Protótipos

#### Vamos entender um pouco o que há nesse protótipo.

#### 1.1.1 - Tela principal

Nesta tela vemos duas divisões: **Contatos**, onde existe uma tabela com a lista de contatos e um botão “**Cadastrar contato**” (que abrirá uma nova janela para cadastro) e **Detalhes**, que mostra as informações de um contato selecionado da tabela.

Na tabela (não seria divisão?) de Contatos, iremos exibir somente o nome do contato. Na divisão Detalhes, teremos o botão de **Excluir** e o botão **Editar** que abrirá a janela de edição.

#### 1.1.2 - Tela de Cadastro

Esta é uma tela convencional de formulário. Nela estarão contidos todos os campos necessários para o cadastro do Contato, um botão **Salvar** e um botão **Cancelar**, para caso o usuário desista do cadastro.

#### 1.1.3 - Tela de Edição

Esta tela será reaproveitada da tela de cadastro, com as únicas diferenças de que no título da página que nos campos de texto já irão aparecer preenchidos os dados do Contato em edição.

Sugestão para parágrafo anterior:

Esta tela será reaproveitada da tela de cadastro, com as seguintes diferenças: o título da tela será e os campos de texto já irão aparecer preenchidos os dados do contato em edição.

### 1.2 - Diagrama de classes das Views

Fazer o diagrama de classes de Views é importante, pois este ajuda a interpretarmos o relacionamento que uma View tem com a outra. Se ao final desta unidade você analisar novamente o diagrama que criaremos a seguirAntes de começar a diagramar, devemos levar em conta a ferramenta a ser utilizada para criação das Views.

Utilizaremos a biblioteca Java Swing, que proporciona uma série de facilidades. Swing é uma biblioteca Java poderosa para criação de telas e componentes gráficos que daremos mais detalhes ao longo da aula e nas próximas aulas.

Mas como instalar o Swing? Não precisaremos, pois o Netbeans tem suporte para Java Swing Interface Builder, que permite a criação de telas com simples “clicar e arrastar”. Não é preciso fazer nenhum download ou instalação adicional, Swing já vem com o Netbeans.

Na prática o Swing é mais um pacote de classes, como os pacotes que temos criado, porém com muito mais classes, por exemplo..... Através disto, muitos códigos são autogerados nas classes destas Views. O diagrama de classe não precisa prever estes componentes autogerados.

Lembra quando dizemos, na unidade 4, que não precisamos mostrar os getters e setters no diagrama de classes pois já são implícitos? Para os componentes autogerados do Swing seguiremos a mesma regra, então vamos nos ater apenas aos botões, campos de texto e tudo que é mutável na View.

#### 1.2.1 - Tela principal

Os elementos abordados serão:

* Botão de Cadastrar, Editar e Excluir
* Tabela
* Rótulos de Nome, Email, Endereço e Grupo

Daremos o nome DashboardView para a nossa View principal, o diagrama ficará da seguinte maneira:

#### *imgs/i2.png*

Figura 5. – Diagrama da classe DashboardView

#### 1.2.2 - Tela de Cadastro e Edição

Os elementos abordados serão:

* Botão de Salvar e Cancelar
* Campos de texto Nome, Email, Endereço e Grupo

Daremos o nome RegisterOrEditContactView (esta tela será utilizada para cadastro e edição de contatos) para esta tela e o diagrama ficará da seguinte maneira:

imgs/i3.png

Figura 5. – Diagrama da classe RegisterOrEditContactView

#### 1.2.3 - Relacionamento entre as telas

A tela RegisterOrEditContactView será instanciada pela DashboardView no momento do clique no botão “**Cadastrar contato**” ou “**Editar contato**” e elas trocarão informações que serão definidas nos *Controllers*. O relacionamento entre as *Views* será de Associação, pois uma RegisterOrEditContactView pode ser independente de DashboardView e vice-versa.

imgs/i4.png

Figura 5. - Relação DashboardView e RegisterOrEditContactView

Agora que modelamos, o próximo passo é criar nossas telas.

### 1.3 - Criando nossas telas

Você poderia pensar: “Será que um programa em Java não passa de interações em linha de comando?”. A resposta é não, mudando essa perspectiva temos a Java Swing.

Softwares que regem empresas inteiras têm suas telas construídas em Java Swing. As aplicações Java em Desktop têm perdido força para aplicações Web e nativas do sistema operacional, por este motivo é um pouco difícil encontrar algum exemplo de aplicação Java Swing. Então mãos na massa.

O primeiro passo da mão na massa é entender que geralmente em empresas, existem os designers ou equipes de criação, responsáveis pelo *layout*, ou seja, a aparência que os aplicativos ou softwares e outros tipos de produtos deverão ter.

Ao contrário da fase de protótipo, estes desenhos devem ser fiéis ao produto final, para que o desenvolvedor replique o conceito desenhado na aplicação. Estes desenhos são chamados de Mockups, lembra-se?

E adivinhe, nossos Designers já criaram todos os mockups para você. Confira:

1. Tela principal:



Figura 5. – Mockup da DashboardView

1. Tela de Cadastro/Edição:

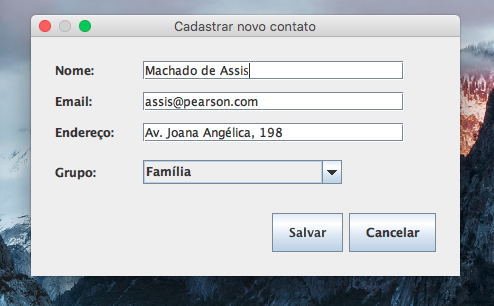


Figura 5. - Mockup da RegisterOrEditContactView

Como podemos ver, estruturalmente, elas não fugiram do que estava previsto nos protótipos do início desta aula (seção 1.1). Agora é hora de implementar. Começaremos com a tela mais simples, a tela de cadastro/edição.

#### 1.3.1 - Criando o projeto

Vamos começar o nosso projeto!

Crie um novo projeto com o nome ContactsApp, mas não crie a classe principal e logo iremos explicar o porquê.

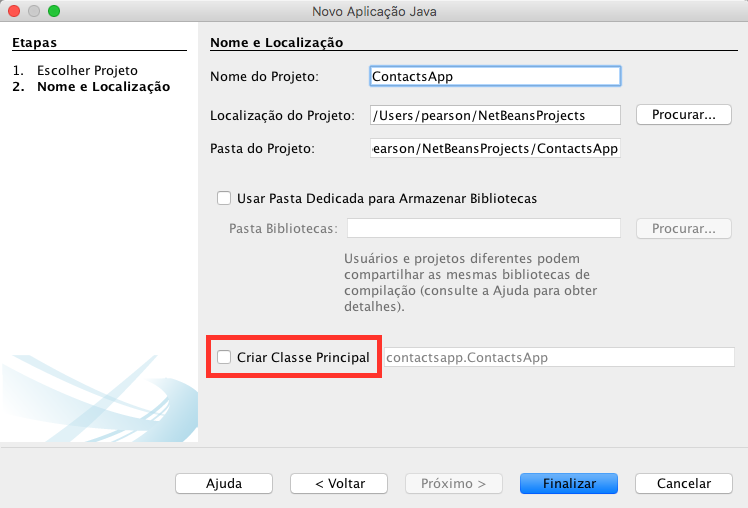


Figura 5. – Tela de criação de um novo projeto

Com o projeto criado, crie um pacote chamado Views. Clique com o botão direito do mouse sobre a pasta “**Pacotes de Códigos-fonte**” e em seguida clique em **Novo > Pacote Java**. Na nova tela que foi aberta, escolha um nome para o pacote, sugerimos que para acompanhar o exemplo do seu livro, você siga este exemplo e dê o nome do pacotede Views) e clique em **Finalizar.** A estrutura do seu projeto ficará como mostrado na Figura 5.8.

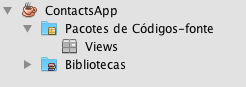


Figura 5. – Estrutura inicial do projeto

#### 1.3.2 - Criando a primeira tela

Agora vamos criar nossa tela. Ela será um JDialog. Primeiro vá em **Arquivo > Novo Arquivo**. Na tela que se abriu selecione a opção **Forms GUI Swing** e na divisória ao lado selecione **Form JDialog**. Clique em **Próximo**.

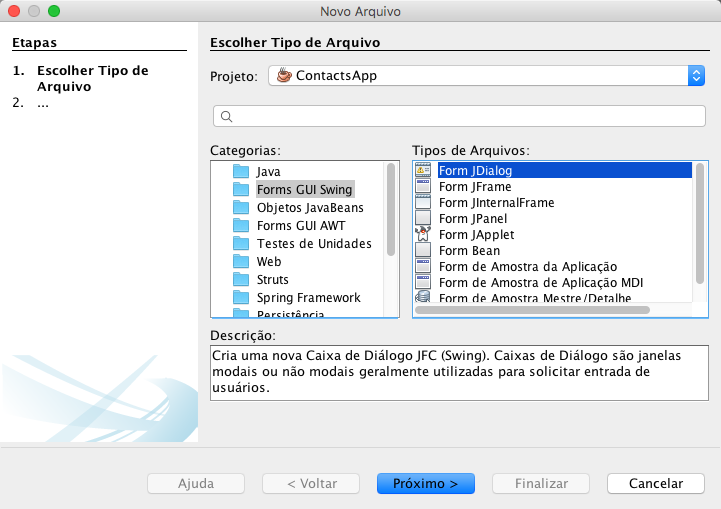


Figura 5. – Tela de seleção de tipo de arquivo

Nomeie nosso JDialog para RegisterOrEditContactView e selecione o pacote Views e clique no botão **Finalizar**:

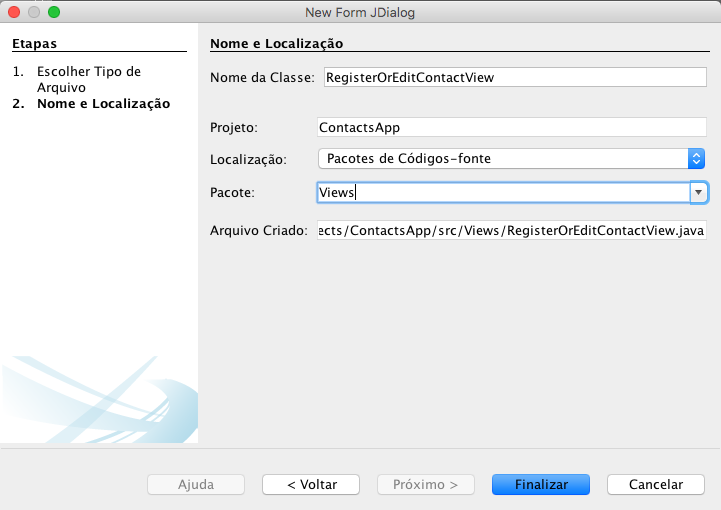


Figura 5. – Tela de Nomeação e Localização do novo arquivo

Pronto execute o programa e veja a beleza brilhar na frente dos seus olhos.

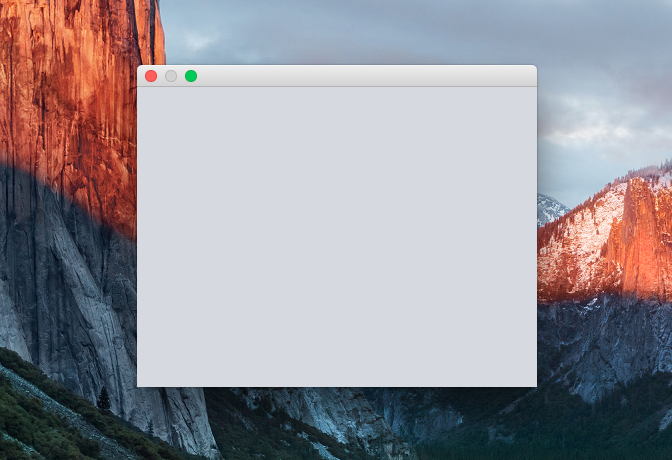


Figura 5. – Tela inicial construída

Uma tela em branco irá aparecer!

Não é muita coisa, mas já é um grande avanço comparado as linhas de comandos.

Esta tela é chamada de JDialog. Este tipo de tela tem seu próprio método main que faz com que a tela possa ser executada independente de outros serviços. A característica principal do JDialog é que quando ele é executado, ele sempre sobrepõe outras telas com a presença de uma nova janela. O JFrame, que será a classe da nossa tela principal, é uma outra janela, porém só é possível ter um JFrame na aplicação. É como se fosse a tela principal.

Quando se faz necessário trocar os componentes apresentados na tela, o JFrame apenas atualiza a si mesmo e apresenta os novos componentes sem a necessidade de apresentar uma nova janela por cima.

#### 1.3.3 - Entendendo o Interface Builder

Ao abrir o arquivo RegisterOrEditContactView.java você vai se deparar com algumas abas:

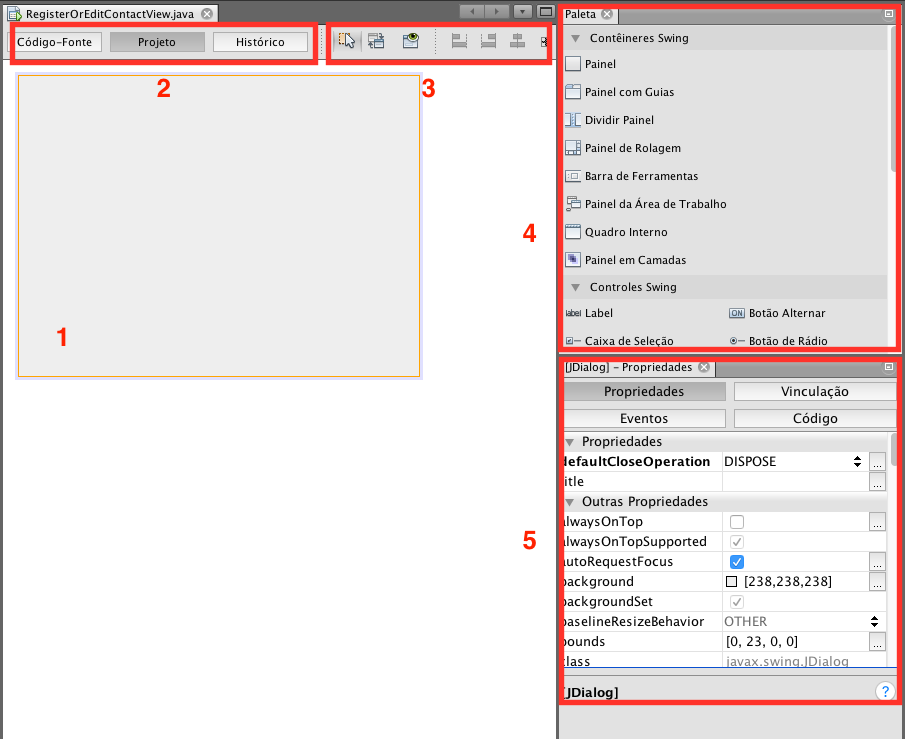


Figura 5. – Interface Builder

1. Esta é nossa área de trabalho. É aqui que pré-visualizamos nossa tela e inserimos componentes a ela.
2. Nesta aba alternamos o modo de visualização da classe. Existem três modos:
   1. **Projeto**: É a visualização no modo Interface Builder. Este é o modo ilustrado na Figura anterior.
   2. **Código-Fonte**: Mostra o código-fonte da tela. Sim, nós podemos criar a tela inteiramente via código.
   3. **Histórico**: Mostra um histórico de modificações no arquivo.
3. Alinhamento de elementos e pré-visualização da tela são encontrados nesta seção.
4. Paleta de elementos. Os principais elementos gráficos do Swing são encontrados aqui. Basta clicar e arrastar para a área de trabalho para utilizá-los.
5. Painel de propriedades. Todos os elementos gráficos do Swing possuem propriedades, como tamanho, cor, fontes e etc. Nesta seção podemos configurar tais propriedades.

#### 1.3.4 – Paleta de elementos

O Swing tem um leque enorme de componentes ao seu dispor. Com estes componentes você conseguirá compor qualquer tipo de aplicação. Existem alguns componentes que são essenciais e que estão presente em quase todos os aplicativos desktop. Eles se encontram na Paleta de elementos e iremos utilizar os seguintes elementos:

##### 1.3.4.1 - JLabel

É um rótulo ou um texto simples que exibe informações na tela. Ele não é interagível, ou seja, o usuário não pode exercer uma ação sobre ele (clicar, tocar, deslizar, selecionar, etc). A fonte (tamanho, cor, estilo, etc) do seu texto é personalizável.

##### 1.3.4.2 - JTextField

Um campo que tem o intuito de receber um texto digitado pelo usuário. Este componente é utilizado para preenchimento de formulários. Ele possui o atributo text, que contém o texto digitado pelo usuário e é atualizado a cada modificação do texto. A fonte do seu texto é personalizável.

##### 1.3.4.3 - JCombobox:

É uma caixa de combinação que reúne valores definidos para o usuário selecionar. Diferente do JTextField e do JCombobox, ele restringe a entrada do usuário a valores predeterminados pelo programador. Seus valores podem ser definidos no atributo model.

##### 1.3.4.4 - JButton

O JButton é um botão que recebe o evento de clique. Quando um evento de clique é realizado o JButton invoca um método. O Netbeans deixa este método pronto para ser implementado, e cabe a nós definir o conteúdo deste. É neste método que definimos a ação que o botão realizará.

##### 1.3.4.5 – JPanel

É um componente que agrupa outros componentes. Ele é utilizado para organizar o layout em camadas, pois podemos adicionar componentes sobre o JPanel. Como o JPanel também é um componente, podemos inseri-lo sobre um outro, assim criando uma estrutura de camadas.

##### 1.3.4.6 - JDialog

Uma JDialog é uma janela de diálogo ou subjanela independente que tem o papel de mostrar um aviso temporário, que sobrepõe a janela principal da aplicação Swing. A maioria das caixas de diálogo são utilizadas para apresentar uma mensagem de erro ou aviso de um usuário, mas estas janelas podem apresentar imagens, árvores de diretórios ou qualquer outra coisa que se faça necessário sobrepor a janela principal da aplicação.

##### 1.3.4. - JFrame

Essa classe é responsável por criar a tela em que iremos desenhar colocando botões, menus, caixas de texto e tudo mais que existe nas janelas de aplicativos. É a classe principal da aplicação e só pode haver um JFrame sendo executado por vez, ao contrário do JDialog.

##### 1.3.4.7 - JTable

O JTable é utilizado para visualizar dados em forma de grade, com cabeçalho, colunas e linhas e é um dos componentes mais complexos do pacote Swing. Seu trabalho é dividido em três partes:

* **Model**: Cuida dos dados da tabela, ou seja, é quem controla e distribui os mesmos. É implementado pela interface TableModel.
* **View**: Cuida da apresentação da tabela. É implementado pela interface CellRenderer. A apresentação é dada célula a célula da tabela.
* **Controller**: Controla a apresentação dos dados na camada view.

##### 1.3.4.8 - JMenuBar

É uma barra de menu que geralmente vemos no topo dos programas. Você pode adicionar objetos JMenu à barra de menu para construir um menu. Quando o usuário seleciona um objeto JMenu, uma janela popup é exibida, permitindo que o usuário selecione uma dos JMenuItems sobre ele.

##### 1.3.4.9 - JMenu

Uma janela pop-up contendo itens de menu (JMenuItem) que é exibido quando o usuário seleciona um item no JMenuBar. Além de itens do menu, o JMenu também pode conter separadores.

##### 1.3.4.10 - JMenuItem

Um item de menu é essencialmente um botão posicionado em uma lista. Quando o usuário seleciona o "botão", é executada a ação associada ao item de menu.

#### 1.3.4 - Inserindo elementos na tela

Antes de inserirmos os elementos na **área de trabalho** do Interface Builder, você pode ajustar o tamanho da área de trabalho clicando e arrastando as bordas do JDialog (que é o componente que já está inserido na área de trabalho).

Para a tela RegisterOrEditContactView iremos precisar de quatro JLabel (Label), três JTextField (Campo de texto), dois JButton (Botão) e um JComboBox (Combobox). Visite a **Paleta** e insira estes componentes no JDialog clicando e arrastando o componente da Paleta e soltando no JDialog.

#### 1.3.5 – Modificando as propriedades

No painel de propriedades (item 5 da Figura 5.12) podemos alterar diversas propriedades de um componente. Por exemplo, para alterar o texto da JLabel, basta selecioná-la, localizar o atributo text e alterá-lo, vide imagem abaixo:

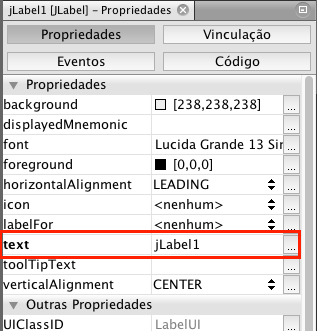


Figura 5. – Propriedade text no painel de propriedades

Agora que aprendemos alterar as propriedades, altere o texto de todas as quatro labels inseridas para corresponder as labels da imagem abaixo:

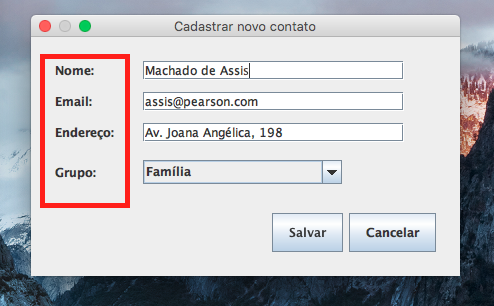


Figura 5. – Organização das Labels

Altere o atributo text de cada JButton também e organize-os:

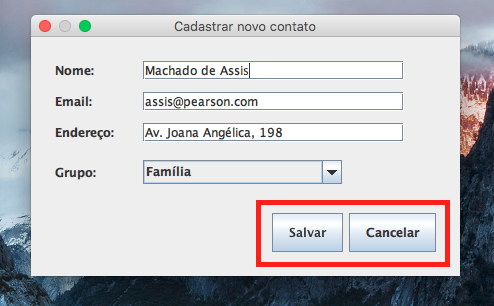


Figura 5. 15 – Organização dos Botõe

Altere a propriedade text cada JTextField e deixe-os vazios, sem texto algum e organize-os para corresponder coma a imagem abaixo:

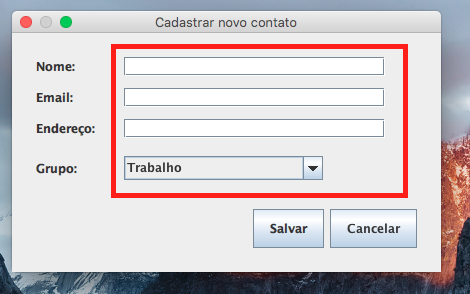


Figura 5. – Organização dos Campos de Textos e Combobox

*DICA: Para alterar qualquer propriedade de algum componente do Swing basta selecioná-lo e alterar a propriedade no painel de propriedades.*

Por fim organize o JCombox. Sua tela deverá se parecer com esta:

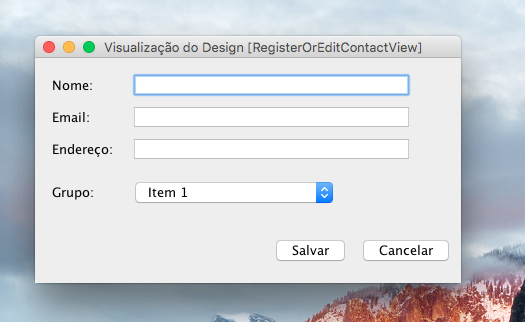


Figura 5. – Tela exibida em modo Preview

Por enquanto não se preocupe com o conteúdo do JComboBox. Ensinaremos futuramente.

Visualize o resultado clicando no ícone Preview da aba 3.

#### 

Figura 5. – Ícone de Preview

### 1.4 - Revisão

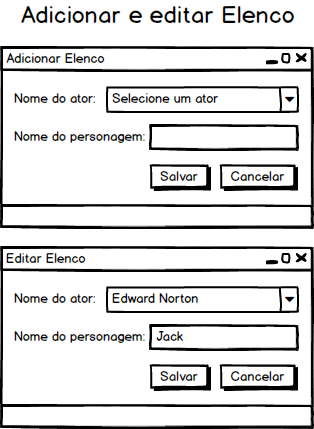
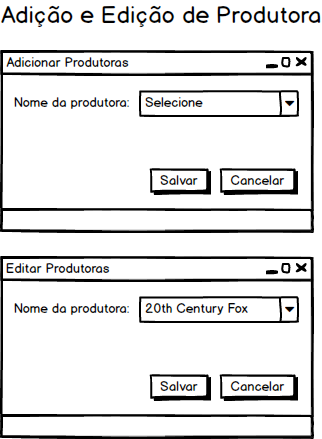
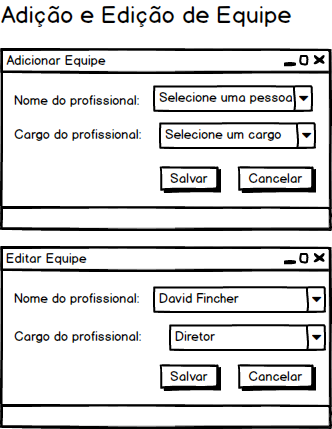
Descobrimos que o Java pode ser um pouco mais charmoso. Utilizando a biblioteca Swing podemos criar telas de nível profissional e deixar nossos sistemas um pouco mais atraente. Vimos que a fase de prototipagem e mockups podem esclarecer a mente do desenvolvedor na hora de entender o propósito de um software e, por fim, aprendemos a compor telas utilizando o Interface Builder do Netbeans.

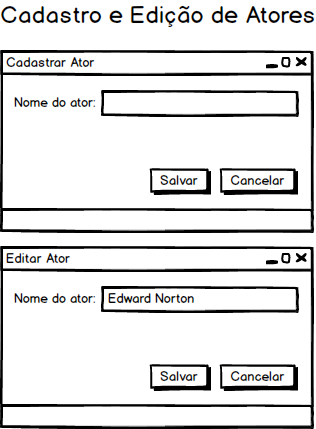
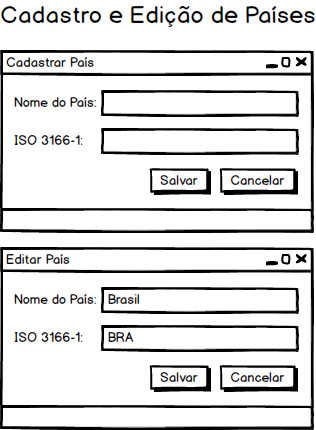
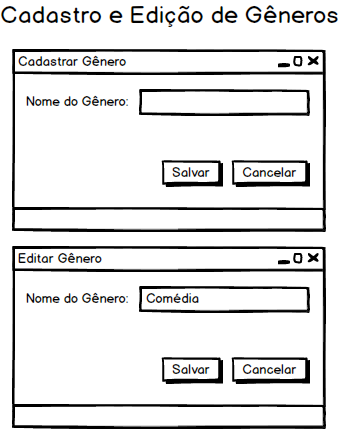
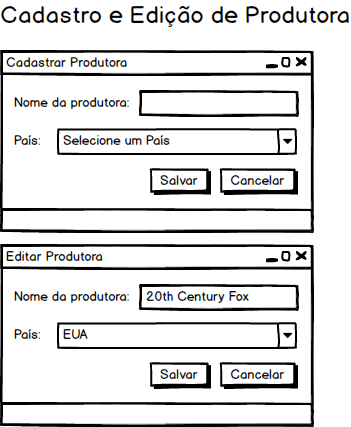
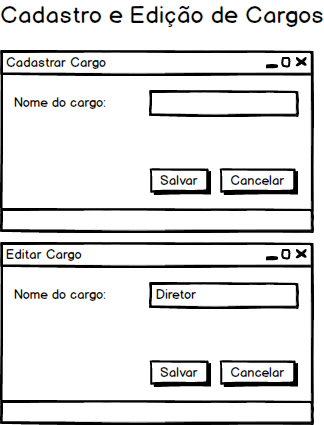
### 1.5 - Exercícios

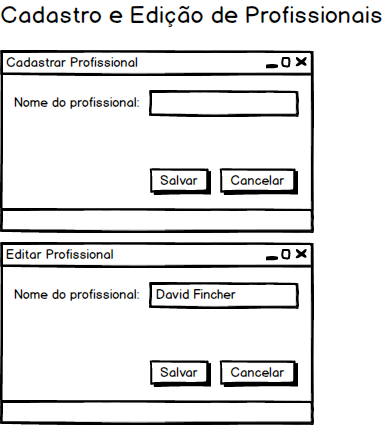
|  |  |
| --- | --- |
| 1. | Qual é o intuito dos protótipos? |
| 2. | O que são os Mockups? |
| 3. | Qual é o nível de detalhamento de um protótipo em comparação com os Mockups? |
| 4. | O que deve ser levado em consideração antes de diagramarmos as classes das views? |
| 5. | O que é o Swing? Existem outras bibliotecas com o mesmo intuito? |
| 6. | O que é um Interface Builder? |
| 7. | Qual é a diferença entre o JFrame e o JDialog? |
| 8. | O que são Layouts? Quais são os tipos de Layouts em Java Swing? Explique a diferença de cada um deles. |

### 1.6 – TDP

Seus protótipos abriram um leque de inspiração para Mariana. Com isso ela teve diversas ideias e um protótipo foi aprovado. Veja a primeira parte de protótipos:



#### Tarefa

Agora que temos os protótipos que definem a estrutura das Views.

E você está chegando ao fim deste projeto!

Você deverá criar suas próprias telas. Use cores, estilos e tamanhos que desejar. Apenas não deixe de seguir a estrutura que os protótipos sugerem. Não se preocupe com os dados por enquanto. Crie dados fictícios e apresente-os na tela.

## Aula 2 – Finalizando as Views e os Models

### 2.1 - A tela principal

Agora que nós já criamos a tela e inserimos os elementos principais, iremos dividir esta tela entre dois JPanels (Painel). O JPanel é um componente gráfico que permite agrupar outros elementos como se fossem parte de uma camada.

Primeiro crie uma nova tela no pacote Views com o mesmo procedimento da aula anterior, com a diferença de que agora não usaremos um JDialog, e sim um JFrame. De o nome a ela de DashboardView.

A tela DashboardView será segmentada em duas seções: **Contatos e Detalhes**. A seção Contatos irá listar resumidamente todos os seus contatos dentro de uma tabela. Já a seção Detalhes irá apresentar os detalhes de um contato selecionado na tabela da seção Contatos.

Estas seções terão botões de ação. A seção Contatos terá um botão para cadastro de novos contatos e a seção Detalhes terão botões de ações sobre os contatos (Excluir e Editar).

Na nova tela insira dois JPanels. Basta localizá-los na Paleta e arrastá-los (um por vez) para a área de trabalho. Estes terão um estilo diferente do padrão. Selecione o Painel e na aba de propriedades, localize a propriedade border e clique no botão ao lado direito “…”. Uma janela se abrirá:

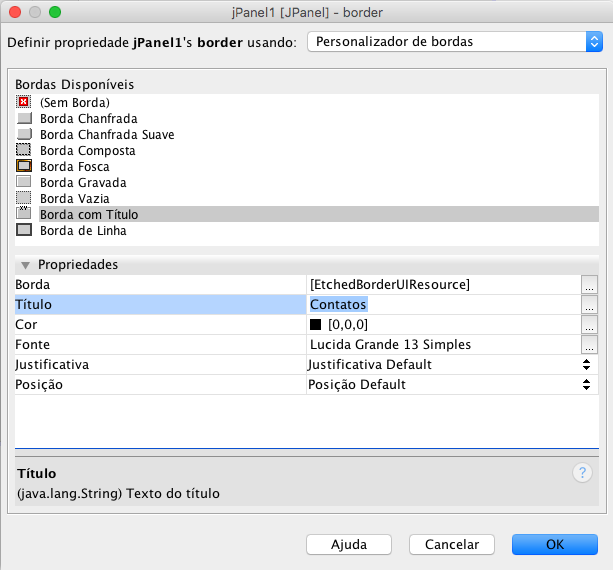


Figura 5. – Tela de propriedades do JPanel

Nesta tela selecione a opção **Borda com Título** e na seção abaixo edite a opção **Título**, e clique em **OK**. Faça isto para ambos painéis. Um deles deverá ter o título “Contatos” e o outro “Detalhes”. Organize os componentes da seguinte maneira:



Figura 5. 20 – Preview da tela com os JPanels inseridos

Temos agora as duas seções construídas.

O JPanel funciona muito bem quando o intuito é agrupar e organizar os componentes em camadas.

Agora iremos inserir os campos de dados. No Painel Detalhes, insira quatro Labels e edite a propriedade text de cada uma. Os valores de text devem ser: “Nome:”, “Email:”, “Endereço:” e “Grupo:”. Selecione cada Label e na propriedade font clique no botão do canto direito “**…**” e edite as fontes para **negrito**.

As Labels são ótimas para exibir textos que não receberão interações do usuário (toques, cliques, gestos, etc.).

Adicione dois botões no rodapé do Painel. O botão com text “Editar” e outro “Excluir. Por enquanto este Painel deverá ficar da seguinte maneira:



Figura 5. – Modo Preview com labels e botões inseridos

Temos quase tudo pronto. Os JButtons são utilizados para dispararmos algum evento para os controllers. Na aula seguinte aprenderemos como implementá-los.

Agora insira mais quatro labels no Painel. Estas labels conterão os valores de cada campo de um Contato. Defina valores aleatórios de acordo com respectivo campo. Veja a seguir como deverá ficar sua tela quando houverem valores:

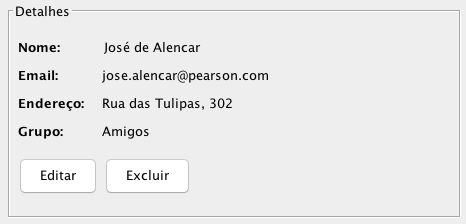


Figura 5. – Modo preview final do painel de detalhes

O que achou? Está como o mockup?

Criar as telas se equipara a desenhar. Quanto mais observadores somos, mais precisa será nossa reprodução do que estamos copiando. O tamanho padrão do JFrame é inadequado com o que é proposto nos mockups, então clique e arraste as bordas do JFrame da área de trabalho para obter um tamanho que se aproxime dos propostos nos mockups.

A seção de detalhes está pronta!

Temos a visualização dos detalhes de um contato. Quando clicarmos no botão **Editar** ou **Excluir**, realizaremos uma ação sobre o contato mostrado nesta seção.

Agora vamos para o Painel Contatos. Insira uma JTable (Tabela) e um JButton (valor de text sendo “Cadastrar contato”) abaixo da tabela de modo que sua tela fique parecida com a figura abaixo após realizar o preview (pré-visualização).

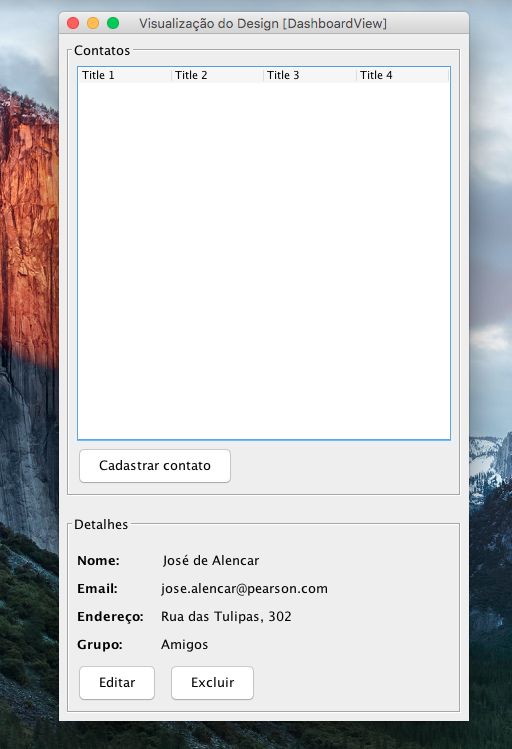


Figura 5. – Visualização da DashboardView

Nesta tabela que acabamos de criar é suficiente mostrar apenas uma coluna contendo o Nome do contato, senão não sobraria informações do contato para serem mostradas no painel Detalhes. Para diminuir a complexidade do exemplo, a tabela não será editável pelo usuário. Siga os seguintes passos:

1. No Interface Builder clique com o botão direito em cima da tabela e em seguida clique na opção **Conteúdo da Tabela.**
2. Nesta etapa estamos definindo quais serão as colunas apresentadas, quais tipos de dados elas exibirão e alguns tipos de restrições, como na imagem abaixo. Por padrão a tabela possui quatro colunas. Vamos excluí-las.

Na nova tela que foi aberta, entre na aba **Colunas.** Selecione todas as linhas pressionando a tecla SHIFT e clique em excluir:

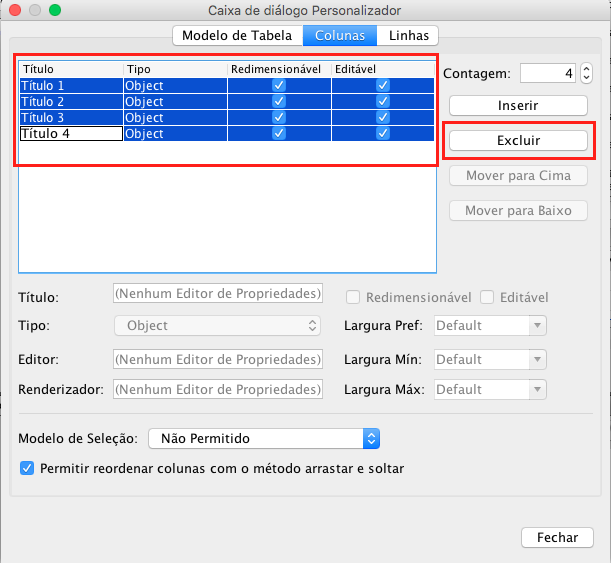


Figura 5. - Personalizar Tabela

1. Agora vamos inserir nossa coluna Nome. Clique em Inserir. Clicando duas vezes no campo null na coluna **Título**, apague a palavra null (desta nova inserção para-retirar) e escreva “Nome”, clique no campo Object na coluna **Tipo** e mude-o para String e desmarque o campo da coluna **Editável,** como na figura abaixo:

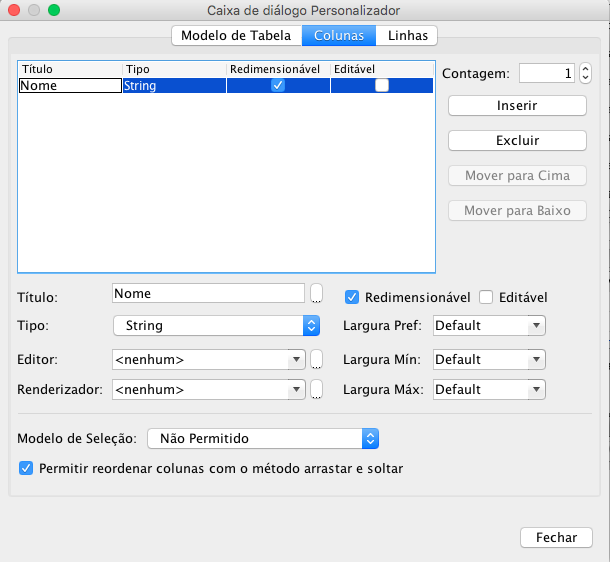


Figura 5. - Configuração desejável da tabela

Estas configurações são o bastante para xxxxx.

Nossa coluna Nome será do tipo String por motivo óbvio, nome é um texto simples. Se mantivéssemos com o tipo Object o programa funcionaria, mas ao invés de apresentar o nome do contato o programa mostraria a representação toString() do Object que fosse designado a aparecer nesta coluna, ou seja, apresentaria não o nome do Contato mas um número hexadecimal por exemplo; pois como nos nossos Models o atributo nome será String, o toString() de uma String é o próprio texto. Por este motivo deixe como String para que possamos interpretar melhor o que está acontecendo.

Clique em fechar visualize como o projeto ficou. Deverá estar parecido com a seguinte imagem:

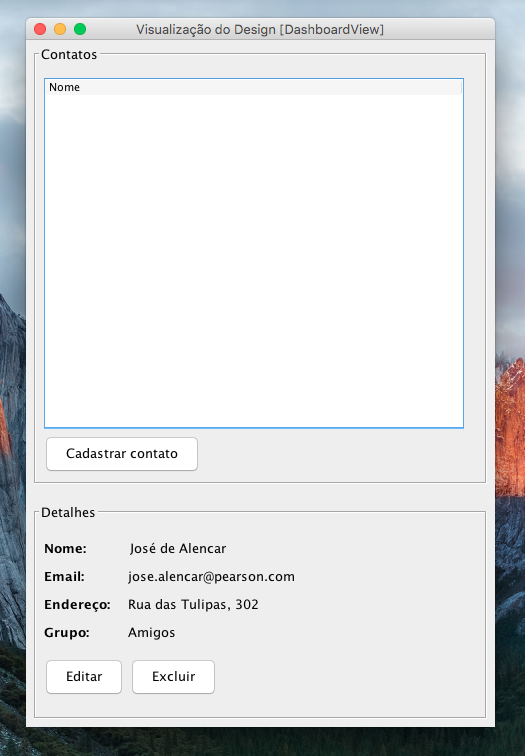


Figura 5. – Preview da DashboardView

Finalmente finalizamos nossas telas. Na aula seguinte iremos terminar nossa camada Model e dar início a camada Controller.

### 2.2 - Codificando nossos Models

Antes de mergulhar na camada de Controllers precisamos finalizar nossa camada Model, codificando o enum e a classe Contact. Lembre que o Controller é o que faz o intermédio entre a camada Model e View, ou seja, a comunicação entre elas. Já finalizamos nossa camada View, agora terminaremos a camada Model para só então criarmos a ligação entre elas: os Controllers. Relembre o diagrama dos nossos Models:

### 

Figura 5. – Relacionamento Contact e Group

Primeiramente vamos criar nosso enum. Crie um novo pacote chamado Models e clique no menu **Arquivo > Novo Arquivo**.

Ao invés de criar uma classe. Crie um enum:

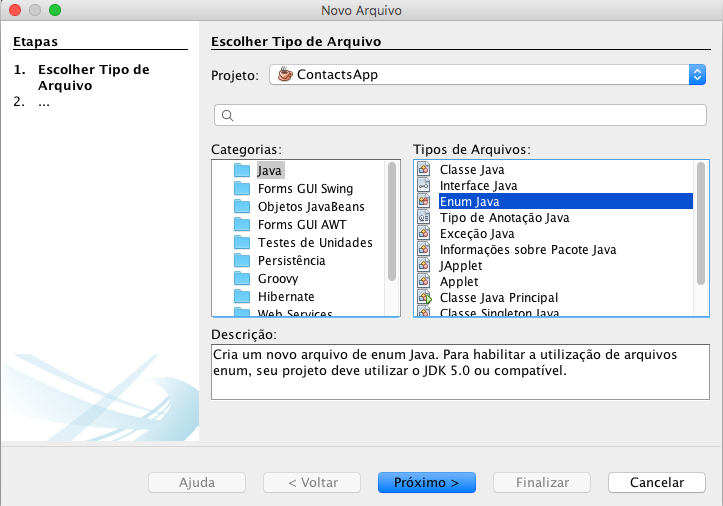


Figura 5. – Tela Novo Arquivo

De o nome Group para ele e coloque-o no pacote Models.

#### imgs/p21.png

Figura 5. – Tela configuração do arquivo

#### 2.2.1 - Enum Group

Como dito anteriormente o enum é um conjunto de valores enumerados. Em Java podemos utilizar enums como Strings ao invés de inteiros. Isso facilita o trabalho pois além de restringir os valores de Group, poderemos popular o Combobox criado na tela de Cadastro/Edição com os valores do enum Group.

Abaixo temos a nossa classe Group. Verifique seu código e **insira os novos trechos** para que sua classe fique como esta:

1. **package** Models;
3. **public** **enum** Group {
4. CoWorkers("Trabalho"),
5. Family("Família"),
6. Friends("Amigos"),
7. School("Escola"),
8. Acquaintance("Conhecidos");
9. **private** **final** String text;
11. /\*\*
12. \* @param text
13. \*/
14. **private** Group(**final** String text) {
15. **this**.text = text;
16. }
18. @Override
19. **public** String toString() {
20. **return** text;
21. }
22. }

Listagem 5.1

Após termos alterado nossa classe Group para aceitar valores String ao invés de Integer, **vamos analisar o código** por partes, começando pela declaração da classe:

**public** **enum** Group

O trecho acima é a declaração dos enum. Semelhante à declaração da classe, aqui especificamos o nome do enum.

1. CoWorkers("Trabalho"),
2. Family("Família"),
3. Friends("Amigos"),
4. School("Escola"),
5. Acquaintance("Conhecidos");

Listagem 5.2

Na listagem acima definimos os casos do enum. Em outras palavras, são os possíveis valores que uma variável do tipo Group poderá assumir. Neste exemplo estamos utilizando enums com valores String.

Até aqui temos o básico para criação de um enum, mas o propósito maior é preencher o Combobox com os possíveis valores deste enum. Para isto devemos implementar um construtor privado e sobrepor o método toString(). Esta implementação é utilizada internamente na biblioteca Swing.

**private** **final** String text;

Este atributo assumirá o valor textual do enum, que será requerido pelo componente Combobox.

1. **private** Group(**final** String text) {
2. **this**.text = text;
3. }

Listagem 5.3

Este é o método construtor. Diferente das classes, o enum necessita de um construtor privado nestes casos especiais. O modificador de acesso deste enum deve ser privado porque devemos impedir que o atributo text seja inicializado, isso poderia abrir a possibilidade de atribuirmos qualquer valor ao enum, e este não é o objetivo dele. Então restringimos o construtor para mantermos a integridade do enum.

1. @Override
2. **public** String toString() {
3. **return** text;
4. }

Listagem 5.4

Como aprendemos o toString() converte o objeto para uma forma textual.

#### 2.2.2 - Classe Contact

Agora vamos entender um pouco sobre a classe Contact.

A classe Contact não sairá do padrão que aprendemos. Crie todos os atributos e encapsule-os com getters, setters e um método construtor

*DICA: Relembre encapsulamento na aula 1.4 da Unidade 3.*

Lembre-se de criar o atributo id. O construtor não deve receber o id via parâmetro, mas devemos inicializá-lo utilizando uma String randômica única.

Neste caso, id é um atributo totalmente privado. Não é possível realizar leitura de seu valor, nem atribuir um valor novo. Este atributo é usado apenas para controle.

Após isso, sobreponha o método equals(), que pode ser autogerado clicando com o botão direito no código, em seguida Inserir Código. Selecione apenas o id ao lado e equals(). O código deverá seguir desta maneira:

1. **package**Models;
3. **import** java.util.Objects;
4. **import** java.util.UUID;
6. /\*\*
7. \*
8. \* @author pearson
9. \*/
10. **public** **class** Contact {
12. **private** String id;
13. **private** String name;
14. **private** String email;
15. **private** String address;
16. **private** Group group;
18. **public** Contact(String name, String email, String address, Group group) {
19. **this**.name = name;
20. **this**.email = email;
21. **this**.address = address;
22. **this**.group = group;
23. id = UUID.randomUUID().toString();
24. }
26. **public** Group getGroup() {
27. **return** group;
28. }
30. **public** **void** setGroup(Group group) {
31. **this**.group = group;
32. }
34. **public** String getName() {
35. **return** name;
36. }
38. **public** **void** setName(String name) {
39. **this**.name = name;
40. }
42. **public** String getEmail() {
43. **return** email;
44. }
46. **public** **void** setEmail(String email) {
47. **this**.email = email;
48. }
50. **public** String getAddress() {
51. **return** address;
52. }
54. **public** **void** setAddress(String address) {
55. **this**.address = address;
56. }
58. **public** String getId() {
59. **return** id;
60. }
62. **public** **void** setId(String id) {
63. **this**.id = id;
64. }

67. @Override
68. **public** **boolean** equals(Object obj) {
69. **if** (**this** == obj) {
70. **return** **true**;
71. }
72. **if** (obj == **null**) {
73. **return** **false**;
74. }
75. **if** (getClass() != obj.getClass()) {
76. **return** **false**;
77. }
78. **final** Contact other = (Contact) obj;
80. **if** (!Objects.equals(**this**.id, other.id)) {
81. **return** **false**;
82. }
83. **return** **true**;
84. }
86. }

Listagem 5.5

Neste ponto sua classe deve estar exatamente como o código da listagem acima.

Repare que dois alertas (warnings) apareceram. Para esclarecimento, os warnings não são erros de código, mas devem ser evitados pois indicam que o trecho causará um erro de execução ou pode ser escrito de uma maneira diferente, ou algo está faltando.

O primeiro warning, está apontando para a declaração do método equals(). Se posicionarmos o ponteiro do mouse sobre ele uma mensagem “Gerar hashCode() ausente” aparecerá dizendo que é preciso sobreescrever o método hashCode() também.

Se você clicar no alerta, ele sugere algumas soluções e se clicar em uma delas, a mesma é aplicada automaticamente.

*DICA: Tome cuidado pois nem sempre o Netbeans está com a razão, algumas vezes ele sugere correções que são irrelevantes ou que não surtirão efeitos positivos na implementação da classe. Aplique esta sugestão se você desejar.*

O segundo warning, que está apontando para a última instrução if do método, alega que “A instrução if é redundante”. Realmente é. Vamos entender.

O método equals, que está sendo utilizado como condição da instrução if, retorna um booleano (true ou false), e caso true, o método retorna true caso contrário, retorna false.

Isso ocorre pois se o tipo de retorno do método é um boolen, a instrução de retorno poderá retornar resultado de uma condição. Para resolver a questão em casos como este, você deverá aplicara sugestão a seguir:

**return Objects.equals(this.id, other.id);**

**Informação importante!**

A classe UUID pertence ao pacote java.util e é responsável por gerar uma String única que identificará nossos Contatos. A sobreposição do método equals() é feita para que mantenhamos um controle maior na comparação entre dois Contatos no momento da Exclusão e Edição.

Nossa camada Model está finalmente pronta. Lembre-se sempre de importar as novas classes utilizadas. Neste caso um alerta aparecerá mostrando a necessidade de importação da classe UUID.

### 2.3 – Revisão

Nesta aula continuamos a criação das Views e aprendemos a configurar uma JTable. Também consolidamos nossos conhecimentos sobre os enums. Eles serão bastante utilizados na sua vida de programador. Por fim codificamos a camada Model e aprendemos como comparar objetos utilizando o método equals().

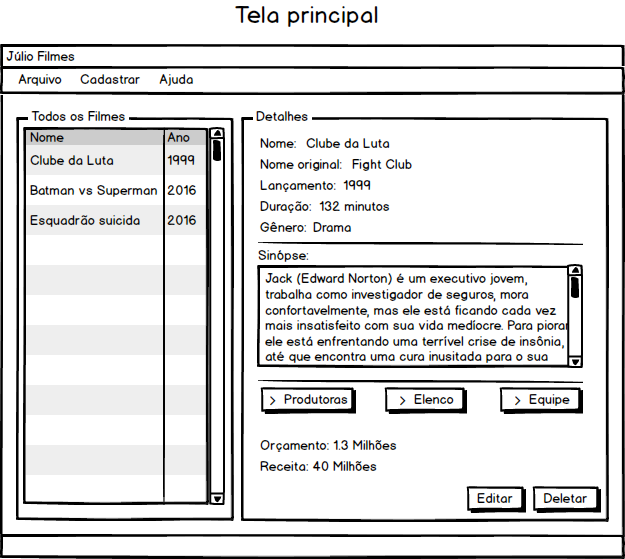
### 2.4 – Exercícios

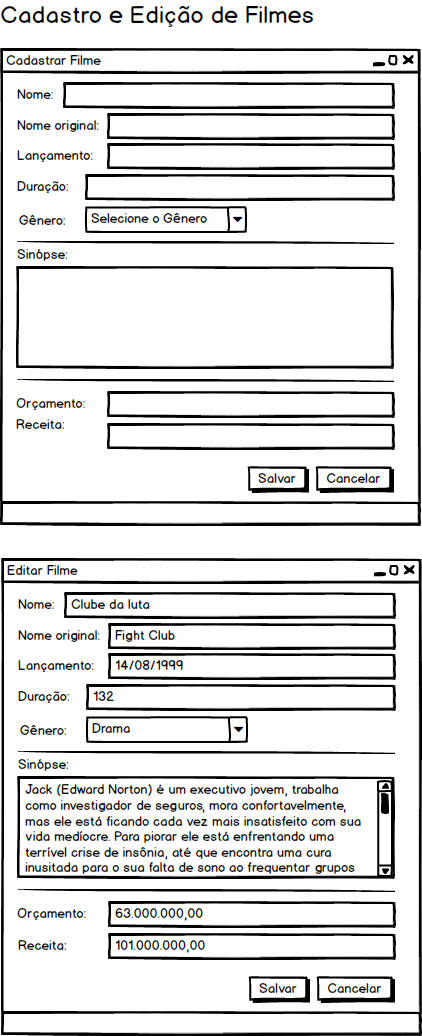
|  |  |
| --- | --- |
| 1. | O que é um JPanel? |
| 2. | Como realizar uma ação quando clicarmos em um JButton? |
| 3. | Explique qual é a funcionalidade do método equals(). |
| 4. | Para que serve a Classe UUID? |
| 5. | O que deve ser modificado em um enum para que ele atribua valores do tipo String? |
| 6. | Por que devemos especificar o tipo de dado de uma coluna de uma JTable? |

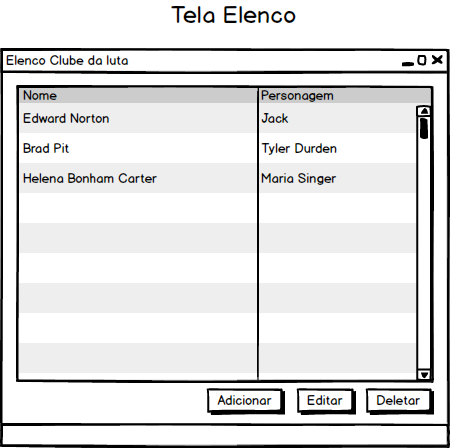
### 2.5 – TDP

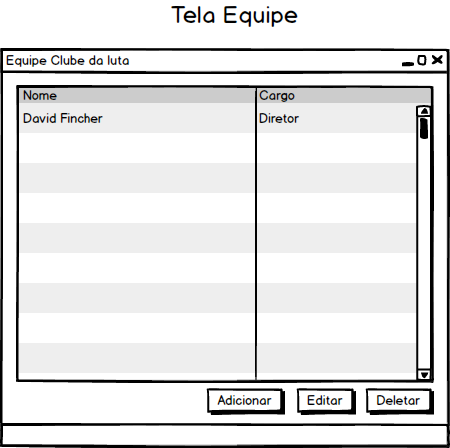
#### 2.5.1 - A criação das telas

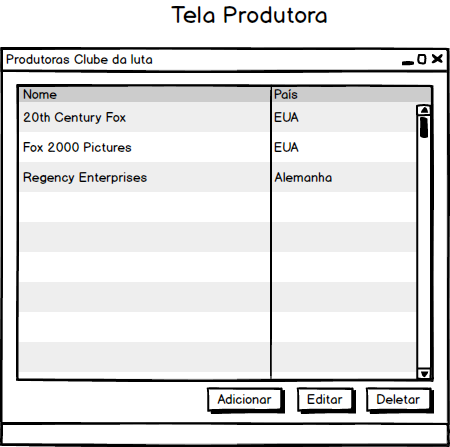
Pode ser um pouco cansativo criar as telas. Agora sobraram as telas principais do aplicativo. São as mais desafiadoras. Veja a segunda parte de protótipos:











Agora você e seu amigo deverão analisar bem esta segunda demanda de protótipos e dar vida a eles também.

#### Tarefa

Estamos chegando ao FIM!

Continue a criação das Views como você fez na aula anterior. Crie as Views representadas nos protótipos acima. Lembre-se que você está livre para usar o estilo que quiser na criação das View, apenas tente seguir a estrutura proposta nos protótipos.

## Aula 3 - Controllers

UAU! Parabéns por ter chegado até aqui.

Estamos em nossa última aula de POO. Vamos finalizar com chave de ouro!

### 3.1 - A Camada Controller

Quando adicionamos um novo arquivo Swing, como por exemplo nossos JDialogs e o JFrame, o Netbeans cria o arquivo .java que já conhecemos e um novo arquivo com extensão .form. O arquivo .form não é visível na árvore de arquivos do projeto no Netbeans, mas quando estamos compondo uma View através do interface builder é nestes arquivos que estamos mexendo.

Neste caso nossos Controllers seriam os arquivos .java que você criou. DashboardView.java e RegisterOrEditContactView.java. Se você preferir, pode renomear o pacote Views com o nome ViewControllers.

Além destes arquivos, teremos um outro Controller, que irá gerenciar (salvar, editar e deletar) a lista de contatos. Então crie um novo pacote clicando em **Arquivo > Novo Arquivo,** selecione **Pacote Java** na seção **Tipos de Arquivos,** clique em **Próximo** , nomeie-o de “Controllers”e clique em **Finalizar.**

Agora vamos criar nosso gerenciador de contatos. Neste gerenciador armazenaremos a lista de todos os contatos salvos em uma seção (execução do app). Também iremos adicionar, remover e alterar os itens desta lista através deste gerenciador. Este gerenciador chamaremos de ContactsController. Crie uma classe para ele e insira o seguinte código:

1. **public** **class** ContactsController {
3. // Lista de contatos salva
4. **public** **static** ArrayList<Contact> contactList = **new** ArrayList<>();
5. // Contato que será selecionado na tabela
6. **private** **static** Contact selectedContact = **null**;
7. // Construtor
8. **public** ContactsController() {
10. // Se a lista estiver vazia ainda, inserimos alguns contatos como teste
11. **if** (contactList.isEmpty()) {
12. contactList.add(**new** Contact("Machado de Assis", "assis@pearson.com", "Av. Joana Angélica, 198", Group.Family));
13. contactList.add(**new** Contact("Carlos Drummond de Andrade", "drummond@pearson.com", "Av. Joana Angélica, 198", Group.Family));
14. contactList.add(**new** Contact("Clarice Linspector", "linspector@pearson.com", "Av. Joana Angélica, 198", Group.School));
15. contactList.add(**new** Contact("Jorge Amado", "jorge@pearson.com", "Av. Joana Angélica, 198", Group.School));
16. contactList.add(**new** Contact("Graciliano Ramos", "ramos@pearson.com", "Av. Joana Angélica, 198", Group.Friends));
17. contactList.add(**new** Contact("Monteiro Lobato", "lobato@pearson.com", "Av. Joana Angélica, 198", Group.CoWorkers));
18. }
20. }
22. // Getters e setters
23. **public** **static** Contact getSelectedContact() {
24. **return** selectedContact;
25. }
27. **public** **static** **void** setSelectedContact(Contact selectedContact) {
28. ContactsController.selectedContact = selectedContact;
29. }
31. // Manipulação da lista
32. // Salva um contato
33. **public** **static** **void** save(Contact newContact) {
35. contactList.add(newContact);
36. }
38. // Edita um contato
39. **public** **static** **void** edit(Contact contactToBeEdited, Contact newValues) {
41. // Atribui os novos valores ao contato a ser editado
42. contactToBeEdited.setName(newValues.getName());
43. contactToBeEdited.setEmail(newValues.getEmail());
44. contactToBeEdited.setAddress(newValues.getAddress());
45. contactToBeEdited.setGroup(newValues.getGroup());
47. // Percorre a lista procurando o item em edição e o atualiza
48. **for** (Contact contact : contactList) {
49. **if** (contactToBeEdited.equals(contact)) {
50. contactList.set(contactList.indexOf(contact), contactToBeEdited);
51. **return**;
52. }
53. }
55. }
57. // Deleta um contato
58. **public** **static** **void** delete(Contact contactToBeDeleted) {
60. // Percorre a lista procurando o item a ser e o deleta
61. **for** (Contact contact : contactList) {
62. **if** (contactToBeDeleted.equals(contact)) {
63. contactList.remove(contact);
64. **return**;
65. }
66. }
67. }


71. }

Listagem 5.6

Vamos analisar:

**public** **static** ArrayList<Contact> contactList = **new** ArrayList<>();

Primeiro criamos uma lista que armazenará todos os nossos contatos. Deixamos como estática pois ela será única para toda a aplicação.

**private** **static** Contact selectedContact = **null**;

Quando selecionarmos um contato na tabela, precisaremos armazenar este dado para o momento da Exclusão e Edição. Este deve ser estático pois principalmente na Edição, que é mostrada outra tela, precisamos deste dado atualizado. Iniciamos com nenhum Contato selecionado.

No nosso consultor, repare que já inserimos alguns contatos na lista. Apenas para nível de teste.

1. // Construtor
2. **public** ContactsController() {
4. // Se a lista estiver vazia ainda, inserimos alguns contatos como teste
5. **if** (contactList.isEmpty()) {
6. contactList.add(**new** Contact("Machado de Assis", "assis@pearson.com", "Av. Joana Angélica, 198", Group.Family));
7. contactList.add(**new** Contact("Carlos Drummond de Andrade", "drummond@pearson.com", "Av. Joana Angélica, 198", Group.Family));
8. contactList.add(**new** Contact("Clarice Linspector", "linspector@pearson.com", "Av. Joana Angélica, 198", Group.School));
9. contactList.add(**new** Contact("Jorge Amado", "jorge@pearson.com", "Av. Joana Angélica, 198", Group.School));
10. contactList.add(**new** Contact("Graciliano Ramos", "ramos@pearson.com", "Av. Joana Angélica, 198", Group.Friends));
11. contactList.add(**new** Contact("Monteiro Lobato", "lobato@pearson.com", "Av. Joana Angélica, 198", Group.CoWorkers));
12. }
14. }

Listagem 5.7

Os próximos métodos são responsáveis por manter o controle na inserção, edição e exclusão de contatos. Veja os comentários no código da listagem 5.6 para maior esclarecimento.

Para eliminar os erros não se esqueça de importar a classe Group, Contacts e ArrayList.

*DICA: Para facilitar use sempre a sugestão que o Netbeans oferece nos alertas de erro. Outro atalho é utilizar Cmd + Shift + I, para MacOS ou Ctrl + Shift + I para Linux e Windows.*

#### 3.1.1 - Preparando lista para a Tabela

Nosso objetivo é mostrar esta lista manipulada no Controller em uma tabela. Para isto, o Java disponibiliza alguns procedimentos a serem seguidos. Na lista anterior criamos nosso ContactController básico e agora vamos transformá-lo em um recuso???? para preencher a JTable **extendendo a classe** AbstractTableModel. Com bastante atenção compare seu código com o código abaixo e faça as modificações necessárias para ficar idêntico a listagem abaixo:

1. **package** Controllers;
2. **import** Models.\*;
3. **import** java.util.ArrayList;
4. **import** javax.swing.table.AbstractTableModel;
5. /\*\*
6. \*
7. \* @author pearson
8. \*/
9. **public** **class** ContactsController **extends** AbstractTableModel{
11. // Atributos para tabela
12. **private** **static** ArrayList rows;
13. **private** String[] cols = {"Nome"};
15. // Lista de contatos salva
16. **public** **static** ArrayList<Contact> contactList = **new** ArrayList<>();
17. **private** **static** Contact selectedContact = **null**;
19. // Construtor
20. **public** ContactsController() {
22. // Se a lista estiver vazia ainda, inserimos alguns contatos como teste
23. **if**(contactList.isEmpty()) {
24. contactList.add(**new** Contact("Machado de Assis", "assis@pearson.com", "Av. Joana Angélica, 198", Group.Family));
25. contactList.add(**new** Contact("Carlos Drummond de Andrade", "drummond@pearson.com", "Av. Joana Angélica, 198", Group.Family));
26. contactList.add(**new** Contact("Clarice Linspector", "linspector@pearson.com", "Av. Joana Angélica, 198", Group.School));
27. contactList.add(**new** Contact("Jorge Amado", "jorge@pearson.com", "Av. Joana Angélica, 198", Group.School));
28. contactList.add(**new** Contact("Graciliano Ramos", "ramos@pearson.com", "Av. Joana Angélica, 198", Group.Friends));
29. contactList.add(**new** Contact("Monteiro Lobato", "lobato@pearson.com", "Av. Joana Angélica, 198", Group.CoWorkers));
30. }
32. updateRows();
33. }
35. **public** **static** Contact getSelectedContact() {
36. **return** selectedContact;
37. }
39. **public** **static** **void** setSelectedContact(Contact selectedContact) {
40. ContactsController.selectedContact = selectedContact;
41. }
43. // Salva um contato se não for existente
44. // A verificação de unicidade é feita por comparação de emails
45. **public** **static** **void** save(Contact newContact) {
47. contactList.add(newContact);
48. updateRows();
49. }
51. **public** **static** **void** edit(Contact contactToBeEdited, Contact newValues) {
53. contactToBeEdited.setName(newValues.getName());
54. contactToBeEdited.setEmail(newValues.getEmail());
55. contactToBeEdited.setAddress(newValues.getAddress());
56. contactToBeEdited.setGroup(newValues.getGroup());
58. **for**(Contact contact: contactList) {
59. **if**(contactToBeEdited.equals(contact)) {
60. contactList.set(contactList.indexOf(contact), contactToBeEdited);
61. **return**;
62. }
63. }
65. updateRows();
66. }
68. **public** **static** **void** delete(Contact contactToBeDeleted) {
70. **for**(Contact contact: contactList) {
71. **if**(contactToBeDeleted.equals(contact)) {
72. contactList.remove(contact);
73. **return**;
74. }
75. }
76. updateRows();
77. }


81. // Métodos de AbstractTableModel
83. **public** ArrayList getRows() {
84. **return** rows;
85. }
87. **public** **void** setRows(ArrayList rows) {
88. ContactsController.rows = rows;
89. }
91. **public** String[] getCols() {
92. **return** cols;
93. }
95. **public** **void** setCols(String[] cols) {
96. **this**.cols = cols;
97. }

100. **private** **static** **void** updateRows() {
101. rows = **new** ArrayList();
103. contactList.stream().forEach((contact) -> {
104. rows.add(**new** Object[] {contact.getName()});
105. });
106. }
107. @Override
108. **public** **int** getColumnCount() {
109. **return** cols.length;
110. }
112. @Override
113. **public** **int** getRowCount() {
114. **return** rows.size();
115. }
117. @Override
118. **public** String getColumnName(**int** numCol) {
119. **return** cols[numCol];
120. }
122. @Override
123. **public** Object getValueAt(**int** numRow, **int** numCol) {
124. Object[] row = (Object[]) getRows().get(numRow);
125. **return** row[numCol];
126. }
127. }

Listagem 5.8

Analisando…

**public** **class** ContactsController **extends** AbstractTableModel

O AbstractTableModel é o nosso facilitador. Para que a nossa lista possa ser usada na tabela, implemente os métodos desta classe.

1. **private** **static** ArrayList rows;
2. **private** String[] cols = {"Nome"};

Listagem 5.9

Os atributos cols e rows são auxiliares para implementação dos métodos de AbstractTableModel. Em cols deveremos informar todas as colunas que aparecerão na tabela. Nosso caso, somente a coluna “Nome”.

1. **private** **static** **void** updateRows() {
2. rows = **new** ArrayList();
4. **for**(Contact contact: contactList) {
5. rows.add(**new** Object[] {contact.getName()});
6. }
7. }

Listagem 5.10

Este método mantém rows atualizada com as novas modificações de nomes dos contatos da lista. Ela é estática pois está sendo utilizada por outros métodos estáticos e isto é requerido.

Por fim, o restante dos métodos são getters e setters e implementações padrões das sobreposições requeridas de AbstractTableModel.

Desta maneira nosso Controller está criado. Só precisamos fazer a interligação das Views com os Models.

### 3.2 - Interligando a Views com o Models

A primeira coisa que vamos integrar é o enum com o Combobox. Vá na tela RegisterOrEditContactView e selecione o ComboBox (no modo de visualização “Projeto”).

Para que a manipulação dos componentes seja mais fácil, renomeie todos os componentes que serão mutáveis.

Selecione o componente, clique com o botão direito e em seguida “**Alterar o Nome da Variável…”.**

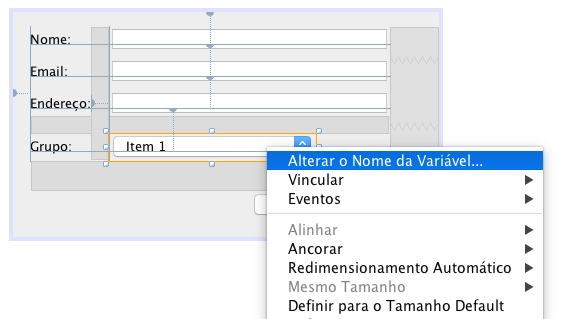


Figura 5. – Alteração do nome de variável dos componentes de UI

Os nomes devem seguir da seguinte forma:

#### imgs/p23.png

Figura 5. – Referência para nomes dos componentes gráficos

#### 3.2.1 - Integrando o Combobox

Agora que criamos toda nossa camada de Controller e Views vamos interligá-las começando pelo combobox da tela RegisterOrEditContactView.

No modo **Interface Builder**, selecione o combobox que nomeamos de groupCombobox e localize a propriedade **model** no painel de **Propriedades** e clique no botão da direita “**…**”. Uma nova janela se abrirá:

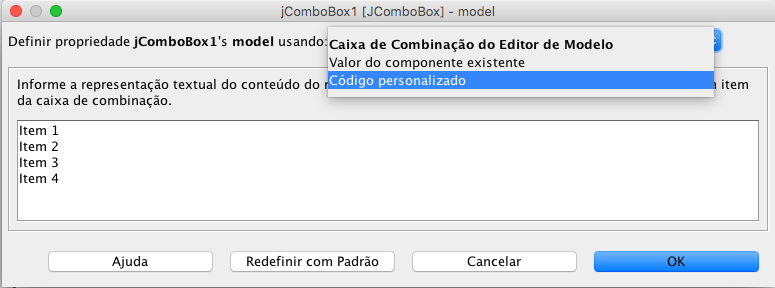


Figura 5. – Personalização de Model da JCombobox

Nesta tela podemos alterar os possíveis valores de um combobox. Poderíamos simplesmente lista-los no campo de texto desta tela, mas vamos fazer do jeito sofisticado, utilizando os valores do enum. Nesta janela é possível injetar código de maneira mais simples. Escolha a opção **Código personalizado** e insira o seguinte código no novo campo de texto apresentado:

**new** DefaultComboBoxModel(Group.values())

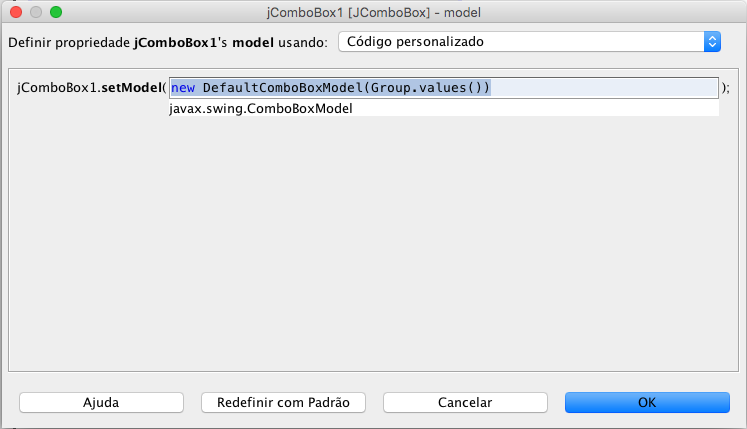


Figura 5. - Personalização de Model da JCombobox (passo 2)

Clique em OK.

O Java Swing gera alguns códigos automaticamente. Estes códigos não podem ser alterados por nós, então eles são mantidos ocultos. Após realizar o passo acima, um erro foi causado dentro deste código autogerado. Para que possamos identificá-lo, entre na classe RegisterOrEditContactView em modo Código-fonte e localize o trecho apresentado na imagem abaixo:

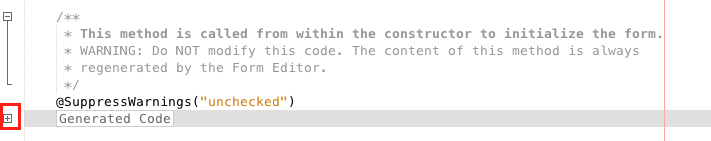


Figura 5. – Identificação do botão que revela o código auto-gerado

Clique no botão “+” para revelar o código autogerado e localize a seguinte instrução com erro:

imgs/p26.png

Figura 5. – Resultado da personalização de model para JCombobox

Importe todas as classes e execute o arquivo RegisterOrEditContactView para ver o resultado, com muita emoção:

*DICA: Como temos mais de uma classe executável, ou seja, ambas Views possuem um método main, para executar uma classe em específico clique com o botão direito do mouse sobre ela e escolha a opção “Executar Arquivo”*

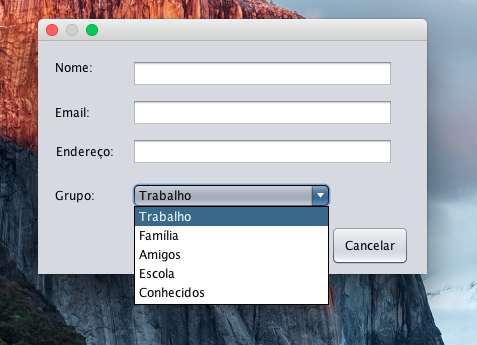


Figura 5. – JCombobox com resultados esperados

Ufa, funcionou! Agora acabamos de consolidar o funcionamento de um enum.

#### 3.2.2 - Fechando a tela

A ação do botão Cancelar na tela RegisterOrEditContactView é simplesmente fechar a mesma. Para realizarmos esta façanha, vá a interface builder desta tela e clique duas vezes no botão **Cancelar**.

Adicione a chamada de dois métodos no local que você foi redirecionado:

1. **private** **void** cancelButtonActionPerformed(java.awt.event.ActionEvent evt) {
2. setVisible(**false**);
3. dispose();
4. }

Listagem 5.11

Primeiro repare que estamos no modo Código-Fonte da classe RegisterOrEditContactView. Este método foi autogerado e é disparado quando o evento de clique no botão Cancelar é realizado. O método setVisible() faz com que a tela fique visível ou não. Neste caso queremos que ela desapareça. O método dispose() faz com que a tela, já invisível, libere espaço da memória, ou seja, a tela é destruída. Estes métodos são próprios do JDialog.

O dispose() é necessário pois o consumo de memória se torna muito elevado quando o app estiver em uso por uma quantidade de tempo moderada, pois todas as telas que imaginamos que estão fechadas ainda estão ocupando espaço de memória se não usarmos o dispose().

Execute o código e confira o resultado.

#### 3.2.3 - Salvando um novo contato

Agora vamos implementar a ação do botão Salvar.

Como dizemos anteriormente, a tela RegisterOrEditContactView servirá para edição e cadastro de usuário, então nosso botão Salvar deve estar pronto para ambas situações.

No interface builder de RegisterOrEditContactView, clique duas vezes no botão **Salvar** e deixe o método da seguinte forma:

1. **private** **void** saveButtonActionPerformed(java.awt.event.ActionEvent evt) {
3. // Resgata os valores dos campos
4. String name = nameTextField.getText();
5. String email = emailTextField.getText();
6. String address = addressTextField.getText();
7. Group group = (Group)groupComboBox.getSelectedItem();
9. // Instancia um novo contato
10. Contact contact = **new** Contact(name, email, address, group);
12. // Prepara a ação para edição
13. Contact selectedContact = ContactsController.getSelectedContact();
14. **if**(selectedContact == **null**) {
15. // Se não existir um contato selecionado, salvamos
16. ContactsController.save(contact);
17. }**else**{
18. // Se existir um contato selecionado, editamos
19. ContactsController.edit(selectedContact, contact);
21. //Após editar, removemos o contato selecionado
22. ContactsController.setSelectedContact(**null**);
23. }
25. // Fecha a tela
26. setVisible(**false**);
27. dispose();
28. }

Listagem 5.12

Os comentários do trecho de código são autoexplicativos. Não se esqueça de importar a classe ContactsController.

Ao executar a tela, não teremos resultado visível, mas acredite, vai dar certo.

O próximo passo é alterar o título da tela de acordo com o contexto. No modo Código-Fonte, ache o método construtor da classe e preencha-o com o código abaixo:

1. **public** RegisterOrEditContactView(java.awt.Frame parent, **boolean** modal) {
2. **super**(parent, modal);
3. initComponents();
5. **if** (ContactsController.getSelectedContact() == **null**) {
6. setTitle("Cadastrar contato");
7. }**else**{
8. setTitle("Editar contato");
9. fillFields();
10. }
11. }
13. **public** **void** fillFields() {
14. Contact contact = ContactsController.getSelectedContact();
16. nameTextField.setText(contact.getName());
17. emailTextField.setText(contact.getEmail());
18. addressTextField.setText(contact.getAddress());
19. groupComboBox.setSelectedItem(contact.getGroup());
21. }

Listagem 5.13

Aproveite e crie o método fillFields(). Quando estivermos em modo edição, ele irá mostrar os valores do contato em edição nos campos de texto.

Nossa tela RegisterOrEditContactView está praticamente pronta. Vamos integrar a tela inicial.

#### 3.2.4 - Integrando a tabela

Estamos nas etapas finais!

Agora acesse o arquivo DashboardView. Como procedimento, vamos renomear as variáveis como na tela anterior, seguindo a imagem guia abaixo:

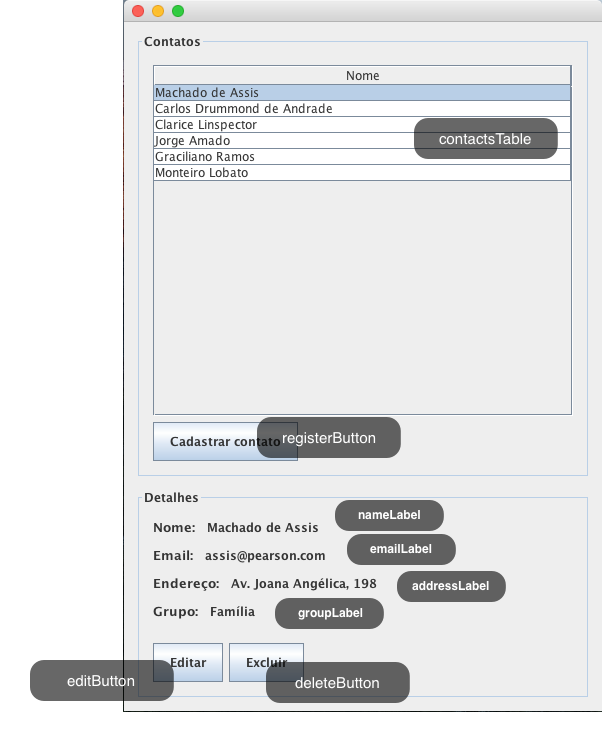


Figura 5. – Referência para nomes das variáveis (tela DashboardView)

Feito isto, entre no modo código fonte, localize o método construtor e insira o seguinte código:

1. **public** DashboardView() {
2. initComponents();
4. contactsTable.setModel(**new** ContactsController());
5. contactsTable.setRowSelectionInterval(0, 0);
6. }

Listagem 5.14

Não se esqueça de importar o ContactsController.

Após iniciar os componentes da View (initComponents() é autogerado e obrigatório para o funcionamento da tela), nos atribuímos um Model para a tabela. Apesar de considerarmos a classe ContactsController um Controller, ela possui a lista de Contatos que servem como Model da tabela, além disso estendemos esta classe de AbstractTableModel. E na linha seguinte fazemos com que a primeira linha da tabela seja selecionada.

Execute a DashboardView e veja o resultado.



Figura 5. – Programa sendo executado

E agora você esta emocionado?

Os contatos pré-adicionados pelo nosso controller estão sendo exibidos na tabela, olhe onde você chegou! Estamos nos aproximando da reta final. O próximo passo é exibir os detalhes do contato ao selecionar uma linha da tabela.

#### 3.2.5 - Exibindo os detalhes de um contato

Para exibir os dados do Contato no Painel de detalhes são necessários dois passos.

* Interceptar o evento de seleção de linha da tabela.
* Atualizar os campos.

Esta interceptação é feita através da implementação de métodos Listeners que serão explicados logo mais. Primeiro vamos criar o método de atualização dos campos. Na classe DashboardView crie o método updatePanel() e implemente-o para corresponder com a listagem a seguir:

1. **private** **void** updatePanel() {
2. // Obtém a célula selecionada no ato da seleção
3. **int** selectedRow = contactsTable.getSelectedRow();
4. // Só atualiza os campos quando existe uma célula selecionada
5. **if** (selectedRow >= 0)  {
6. Contact contact = ContactsController.contactList.get(selectedRow);
7. ContactsController.setSelectedContact(contact);
9. // Atualiza os campos do painel
10. nameLabel.setText(contact.getName());
11. emailLabel.setText(contact.getEmail());
12. addressLabel.setText(contact.getAddress());
13. groupLabel.setText(contact.getGroup().toString());
14. }**else**{
15. // Limpa os campos do painel para quando não houver nada selecionado
16. nameLabel.setText("");
17. emailLabel.setText("");
18. addressLabel.setText("");
19. groupLabel.setText("");
20. }
22. }

Listagem 5.15

Os comentários do trecho de código são autoexplicativos

Para interceptar o evento de seleção da tabela, usaremos o que chamamos de Listener.

Um listener é utilizado para monitorar alguma atividade e avisar quando um evento esperado acontece. Localize o método construtor e adicione este trecho de código ao final do método:

1. contactsTable.getSelectionModel().addListSelectionListener(**new** ListSelectionListener(){
2. @Override
3. **public** **void** valueChanged(ListSelectionEvent event) {
4. updatePanel();
5. }
6. });

Listagem 5.16

Pode parecer estranho, mas é assim que implementamos os listeners. Inserimos um método via parâmetro e este método se chama Callback, que funciona da seguinte maneira:

A contactsTable esta monitorando o evento ListSelection (seleciona um item da lista), mas ela não sabe o que fazer quando este evento for capturado, então ela delega para a classe que a instanciou a responsabilidade de dizer o que deve ser feito quando este evento ocorrer. Esta é uma parte mais avançada do Java, se você se interessou por este recurso pesquise por Java Listeners.

Chame o updatePanel() após a seleção de linha na tabela para termos dados preenchidos no painel.

O método construtor deverá ficar como mostra a listagem abaixo:

1. **public** DashboardView() {
2. initComponents();
4. contactsTable.setModel(**new** ContactsController());
5. contactsTable.setRowSelectionInterval(0, 0);
6. updatePanel();
8. contactsTable.getSelectionModel().addListSelectionListener(**new** ListSelectionListener(){
9. @Override
10. **public** **void** valueChanged(ListSelectionEvent event) {
11. updatePanel();
12. }
13. });
15. }

Listagem 5.17

Execute a tela e selecione diversas linhas da tabela e veja que o painel de detalhes troca seus dados.

#### 3.2.6 - Deletando um contato

Entre no Interface Builder (modo Projeto) da classe DashboardView e clique duas vezes no botão Excluir. Insira o seguinte código:

1. **private** **void** deleteButtonActionPerformed(java.awt.event.ActionEvent evt) {
2. // Obtém contato selecionado na tabela
3. **int** selectedRow = contactsTable.getSelectedRow();
5. // Deleta contato da lista
6. **if** (selectedRow >= 0) {
7. Contact contact = ContactsController.contactList.get(selectedRow);
8. ContactsController.delete(contact);
9. }
11. // Atualiza a tabela
12. contactsTable.setModel(**new** ContactsController());
14. // Elimina contato selecionado
15. ContactsController.setSelectedContact(**null**);
16. }

Listagem 5.18

O comentário no trecho de código é autoexplicativo, mas não se esqueça de importar a classe ContactsController na nossa DashboardView.

Execute a tela, selecione um contato e clique em excluir.

#### 3.2.7 - Interligando as telas

E agora, finalmente a Última etapa é interligar as Views.

Ainda no modo Código-fonte da class DashboardView, crie um método chamado openNewWindow() para ser reutilizado como ação do botão Cadastrar e do Editar:

1. **private** **void** openNewWindow() {
2. // Instancia um novo JDialog padrão
3. RegisterOrEditContactView registerView = **new** RegisterOrEditContactView(**this**, rootPaneCheckingEnabled);
5. // Deixa a nova tela visÃ­vel
6. registerView.setVisible(**true**);
8. // Listener para atualizar a tabela quando um novo contato for salvo ou editado
9. registerView.addWindowListener(**new** WindowAdapter() {
10. @Override
11. **public** **void** windowClosed(WindowEvent e) {
12. contactsTable.setModel(**new** ContactsController());
13. }
14. });
15. }

Listagem 5.19

Importe a classe RegisterOrEditContactView na nossa DashboardView para que ela possa ser chamada ao tocar em Editar ou Cadastrar.

Entre no modo Projeto de DashboardView clique duas vezes no botão Cadastrar contato e insira o seguinte código.

1. **private** **void** registerButtonActionPerformed(java.awt.event.ActionEvent evt) {
3. ContactsController.setSelectedContact(**null**);
4. openNewWindow();
5. }

Listagem 5.20

E para o botão Editar, deixe apenas o método updateNewWindow():

1. **private** **void** editButtonActionPerformed(java.awt.event.ActionEvent evt) {
2. openNewWindow();
3. }

Listagem 5.21

Fim. Nossa aplicação está com todas as funcionalidades.

### 3.3 - Look & Feel

Repare que quando executamos o aplicativo, ele apresenta uma interface com cor predominante azul, que é diferente da interface apresentada quando utilizamos a funcionalidade de preview. Este tema azul é chamado de Nimbus, e é exclusivo de aplicações feitas em Java Swing, porém a maioria dos desenvolvedores não se agradam dele e acabam removendo este tema do projeto. O tema apresentado quando utilizamos o preview é o tema nativo do sistema operacional, ou seja, aplicações Java Swing executadas em sistema operacional MacOS, ficarão com aparência padrão de programas para Mac, e se executadas em sistema operacional Windows, eles ficarão com cara de programas para Windows.

O Swing possui alguns temas disponíveis para a nossa aplicação. Estes temas são chamados de Look & Feel. O tema proposto nos mockups é o chamado “Metal” e para que o Look & Feel da nossa aplicação seja igual ao proposto, vá no modo Código-fonte da tela DashboardView e localize o método main.

Expanda a parte de Look & Feel pelo ícone destacado na imagem abaixo:

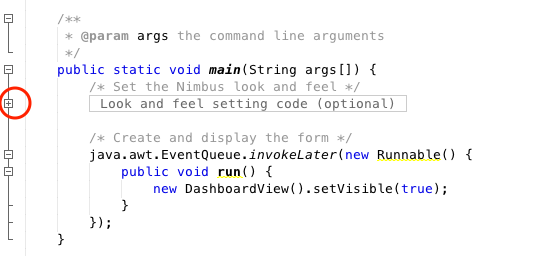


Figura 5. – Botão que revela códigos do Look&Fell

No nosso caso, sugerimos utilizar o tema “Metal” para que você possa acompanhar seu projeto de acordo com o que está vendo em seu livro. Localize a string “Nimbus” e troque por “Metal”. Você pode também optar por utilizar o tema nativo do sistema operacional. Basta remover toda cláusula try catch. Na imagem abaixo, vemos o tema “Metal” ao lado de uma janela com tema Nativo (No caso do exemplo, o sistema operacional é MacOS). Para utilizar o tema nativo, remova todo o código encontrado na seção Look and Feel. Veja como ficará na imagem a seguir:

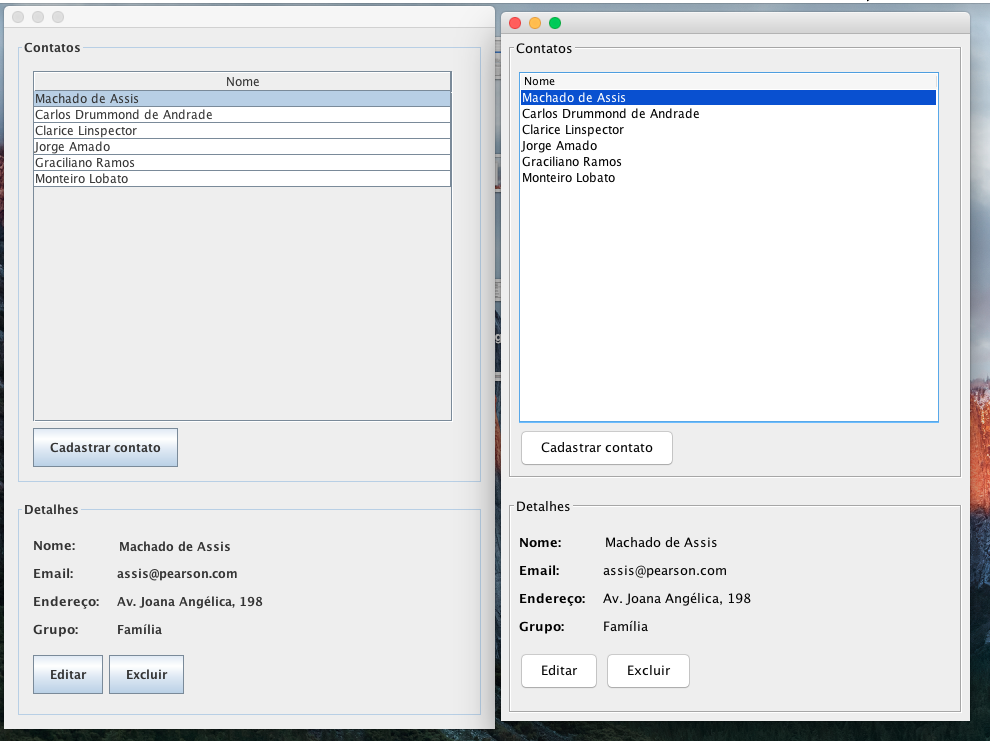


Figura 5. – Comparação entre os Look&Feels

### 3.4 – Resumo

Nesta aula você consolidou seus conhecimentos em Java. Vimos como de fato, funciona uma arquitetura MVC, aprendemos a criar adaptações em nossas classes para servir como modelo em nossos componentes de View. E também conseguimos criar eventos em botões e interceptar eventos interessantes., de agora em diante iremos trabalhar outros aspectos para o desenvolvimento de Aplicativos

Nossas aulas sobre Swing e Java terminam por aqui. Há muito mais sobre Swing para aprender envolvendo outros tipos de aplicações, continue se aprimorando!. Não fizemos tratamento de erros, bloqueio do botão excluir e editar quando não há nenhum contato selecionado. A caminhada do programador é longa, e sempre existe algo mais para ser aprendido. Siga firme, e que você tenha um futuro promissor na área!

### 3.5 – Exercícios

|  |  |
| --- | --- |
| 1. | O que é um Controller? |
| 2. | Sobre os enums, o que é necessário ser feito para que um JCombobox receba-o como Model? |
| 3. | Você utilizou um ArrayList. Pesquise e responda. O que é um ArrayList? Para que serve? |
| 4. | Qual método é necessário para que uma view se torne visível? E qual é o método para destruir uma view? |
| 5. | O que é um Listener? Cite um exemplo. |
| 6. | O que é Look&Feel? |

### 3.6 – TDP

Esta é a atividade final onde colocará à prova todo conhecimento adquirido no curso.

#### 3.6.1 - Controllers

Após ter feitos as Views, você precisa dar vida a elas, interligando-as com os Models, e persistindo os dados salvos.

Primeiro você deverá interligar as Views com os Models através dos Controllers. Muito similar ao aprendido em aula.

#### Tarefa

Chegamos a última etapa!

Como se sente? Realizado com certeza, veja onde chegou!

Para finalizarmos o projeto, nessa etapa final você deverá finalizar as funcionalidades de cadastrar, editar, visualizar e deletar, para isso utilize Controllers para realizar a interação entre View e Models

#### Tarefa extra

Existem diversos temas do Java Swing na internet. Pesquise e instale algum de seu gosto.

#### Fim

E após tudo isso como está o seu projeto?!

Agora chegou o grande momento de apresentar a sua turma e ao seu educador! Parabéns! Você está craque em Java!!