Desenvolvimento de Sistemas

Interface *desktop*: construção de interface de usuário, manipulação de eventos, uso de controles, manipulação de janelas, construção de formulários e listagens (Parte 1)

Introdução

Comumente, quando se começa a programar, os primeiros *softwares* que são criados têm foco no console em formato de texto. Essa é uma ótima forma de se aprender a programar inicialmente, pois o foco está em conhecer a linguagem e desenvolver a lógica de programação. Mas, quando se precisa desenvolver um sistema que será usado por um usuário que não tem conhecimentos técnicos, é preciso apresentar uma interação mais apropriada que o console. É aí que são utilizadas as interfaces gráficas para criar elementos visuais que permitirão ao usuário interagir com o sistema. No decorrer desse conteúdo, você aprenderá como construir interfaces gráficas para *desktop*. Para isso, serão utilizados componentes de um *kit* de ferramentas chamado Swing, por meio do Apache NetBeans IDE.

Construção de interface de usuário

Falar de “interfaces gráficas” significa falar de uma forma visual de interagir com um sistema. Em programação, essas interfaces são chamadas de **GUI** (***g****raphical* ***u****ser* ***i****nterface* ou “interface gráfica do usuário”, em português).

O Apache NetBeans IDE contém um construtor de interface gráfica integrado chamado GUI Builder, que permite construir GUIs em projetos Java apenas arrastando e soltando elementos na tela. Para isso, o GUI Builder utiliza dois *toolkits* (*kits* de

ferramentas) com um rico conjunto de classes prontas: o AWT e o Swing.

AWT (abstract windowing toolkit)

Foi introduzido no JDK (Java Development Kit) 1.0. Sua proposta é utilizar a ambientação gráfica do sistema operacional que está sendo rodado no programa para renderizar elementos visuais que tenham a aparência idêntica, passando a sensação de "*software* nativo". Na implementação, o AWT usa as APIs (*application programming interfaces*, ou "interfaces de programação de aplicação", em português) de componentes visuais do sistema operacional, nas quais a JVM (Máquina Virtual Java) está rodando para criar componentes do sistema operacional. Por se tratar da primeira solução para construção de interfaces gráficas no Java, seu uso é extremamente limitado: a maioria dos seus componentes tornou-se obsoleta e eles devem ser substituídos por componentes de GUI mais atuais.

Swing

O Swing é um conjunto muito mais abrangente de bibliotecas gráficas que aprimora o AWT, trazendo componentes mais leves e flexíveis. Foi introduzido como parte do Java Foundation Classes (JFC) após o lançamento do JDK 1.1. Diferentemente do AWT, o Swing tem componentes que são independentes de plataforma, o que permite padronizar a exibição das interfaces em diferentes sistemas operacionais. Essa tecnologia é mais completa que o AWT, e por isso você aprenderá a construir interfaces gráficas para *desktop* utilizando-a.

Criando um projeto Java com interface gráfica para *desktop*

Para começar, você criará um novo projeto Java no Apache NetBeans IDE. Esse projeto será usado para testar e praticar os exemplos deste conteúdo. Logo, você poderá nomeá-lo como desejar.

1. Clique em **File** > **New Project**. Como alternativa, você pode clicar no ícone de **Novo Projeto**, localizado na barra de ferramentas do NetBeans IDE.
2. Selecione **Java with Ant** > **Java Application.e** e clique em **Next**.
3. Digite o nome do projeto no campo **Project Name** e clique em **Finish**.

Agora que o seu projeto foi criado, desenvolva sua primeira tela:

1. Clique com o botão direito do *mouse* sobre o pacote do projeto e selecione as opções **New** > **JFrame Form**.
2. Na nova janela que será aberta, especifique o nome da classe (**Class Name**). Para manter uma boa organização no projeto, o ideal é criar as classes com um nome referente ao conteúdo da tela. Depois de informar o nome da classe, clique em **Finish**.

Após concluir a criação da classe, uma nova tela será aberta no NetBeans IDE, na qual você será apresentado ao GUI Builder.

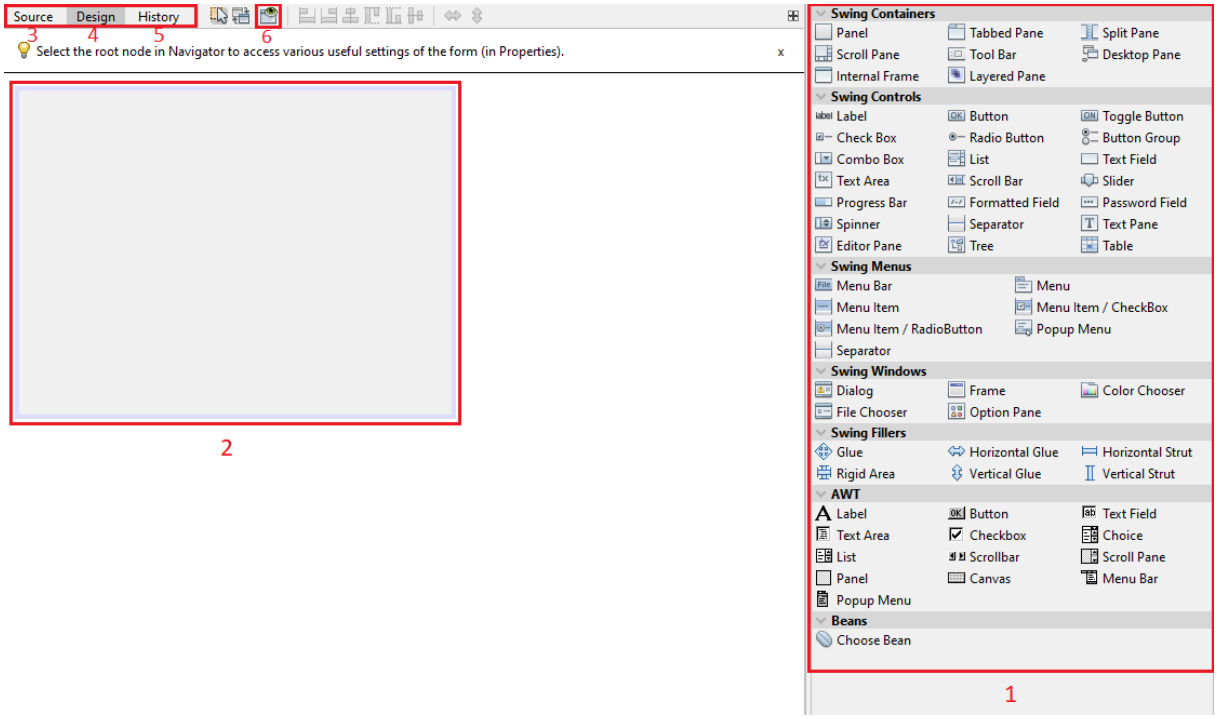


Figura 1 – Tela do GUI Builder

Fonte: Apache NetBeans IDE (2022)

Nessa tela, você pode identificar os seguintes recursos:

O GUI Builder do Apache NetBeans IDE consegue simplificar o fluxo de trabalho de criação de interfaces gráficas, liberando os desenvolvedores das complexidades dos gerenciadores de *layout* do Swing. Você pode criar seus formulários simplesmente

colocando os componentes onde quiser. O GUI Builder descobre quais atributos de *layout* são necessários e gera o código para você automaticamente – o código pode ser acessado na aba de código-fonte **Source code**.

Para desenvolver uma interface amigável para o usuário, são necessários diversos elementos, como caixas de texto, botões, locais para entrada de dados e muito mais. Esses elementos que compõem a interface gráfica do usuário são chamados de componentes.

O GUI Builder do Apache NetBeans IDE traz várias opções de componentes para você construir a interface gráfica da sua aplicação. Tudo o que precisa fazer é selecionar o componente e arrastar para a tela em branco. Ao fazer isso, um código Java será escrito na aba do **código-fonte** para o componente que você escolher ser criado na tela com todas as suas respectivas configurações. Você pode usar a aba de código-fonte para editar valores e chamar métodos para configurar cada componente. Outra opção é você clicar com o botão direito sobre um componente no GUI Builder e acessar a opção **Properties** para configurar os atributos e métodos do componente por meio de uma interface visual.

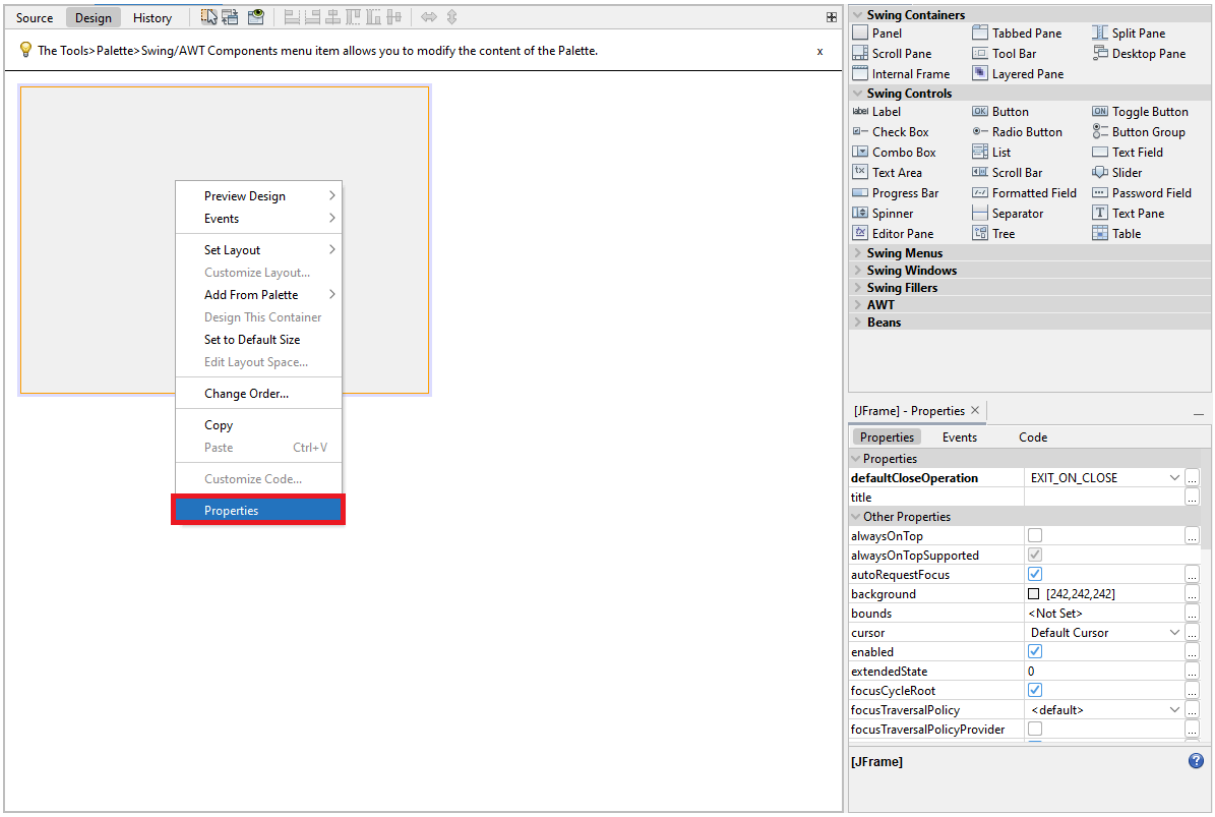


Figura 2 – Tela do GUI Builder
Fonte: Apache NetBeans IDE (2022)

Todos os componentes do Swing, como **JButton**, **JComboBox**, **JList**, **JLabel** são herdados da classe **JComponent**, que pode ser adicionada às classes contêineres. Os contêineres são as janelas ,como quadros e caixas de diálogo. Os componentes básicos do Swing são os blocos de construção de qualquer aplicativo de GUI. Métodos como **setLayout** substituem o *layout* padrão em cada contêiner. Contêineres como **JFrame** e **JDialog** só podem adicionar um componente a si mesmo.

Agora que você se familiarizou com a interface do construtor de GUI, confira alguns principais componentes disponíveis para construir interfaces gráficas.

Observação!

Apesar de o GUI Builder apresentar componentes do AWT, o uso desses componentes não é recomendado. Sendo assim, o foco deste estudo será o uso dos componentes do Swing.

Principais componentes do Java Swing

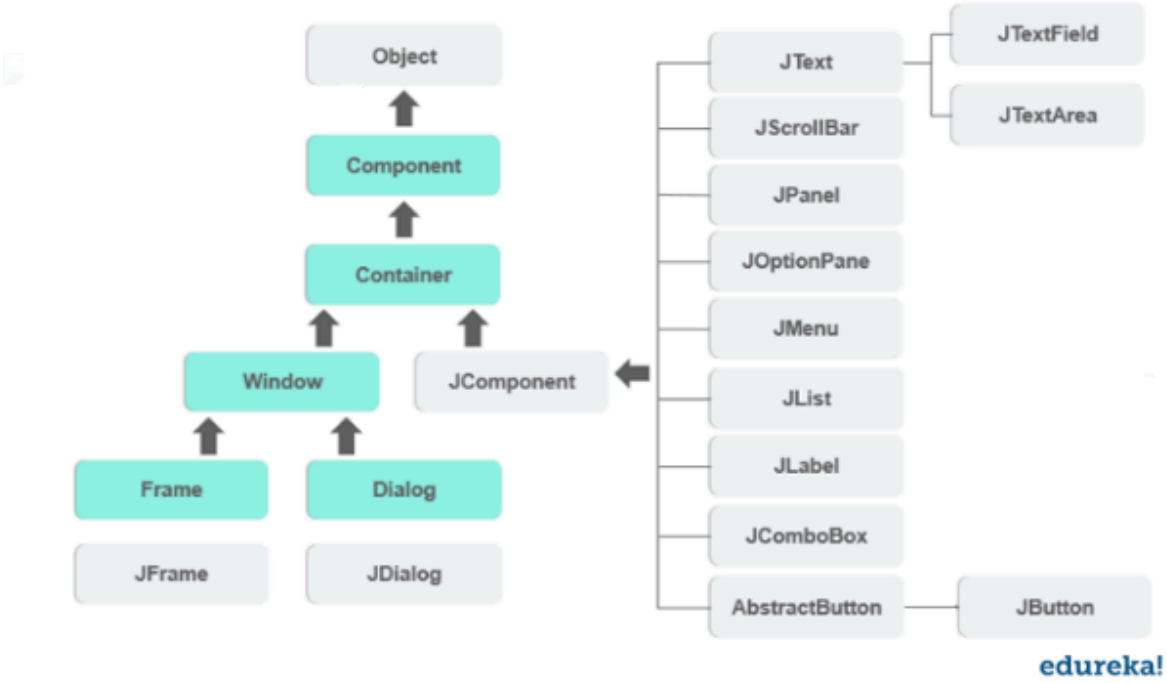


Figura 3 – Hierarquia de componentes do Java Swing
Fonte: Wassem (2021)

A imagem ilustra resumidamente a hierarquia de classes do Java Swing, deixando claro que todos os componentes são derivados direta ou indiretamente de **JComponent**. A seguir estão alguns componentes com exemplos para você entender como pode usá-los.

Frame

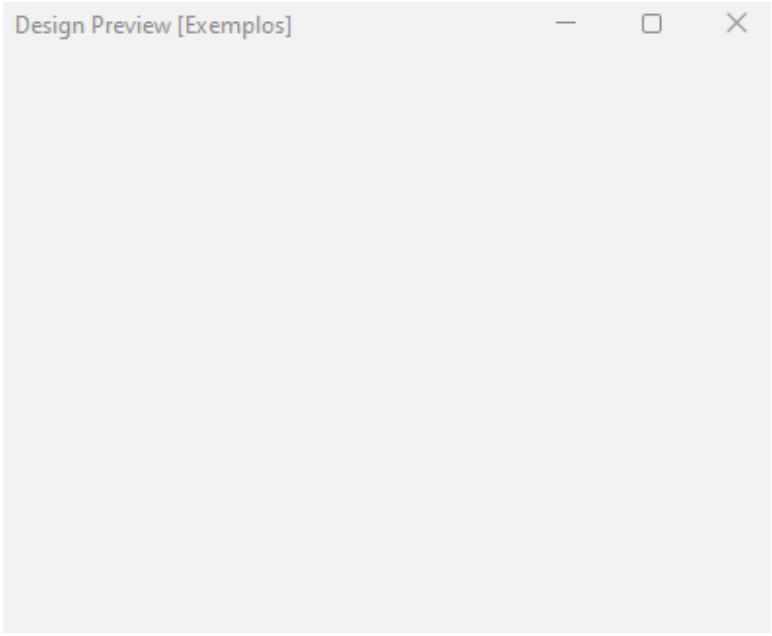


Figura 4 – Exemplo de **JFrame**
Fonte: Apache NetBeans IDE (2022)

Nome da classe: JFrame

Descrição: é a janela do programa na qual podem ser inseridos outros componentes, como botões de comando, ícones, textos etc.

Principais métodos:

- **pack()**: compacta a janela para o tamanho dos componentes.
- **setSize(int, int)**: define a largura e altura da janela.
- **setLocation(int, int)**: define a posição em que a janela do programa ficará quando estiver rodando.
- **setBounds(int, int, int, int)**: define a posição e o tamanho da janela.
- **setVisible(boolean)**: é o método para habilitar (*true*) ou desabilitar (*false*) a exibição da janela.
- **setDefaultCloseOperation()**: define o que ocorre quando o usuário tenta fechar a janela.

Panel

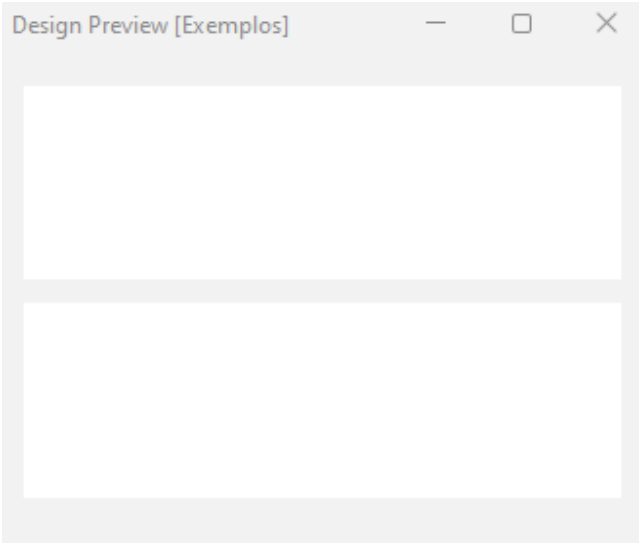


Figura 5 – Exemplo de **JPanel**
Fonte: Apache NetBeans IDE (2022)

Nome da classe: JPanel

Descrição: representa um contêiner básico para componentes a serem inseridos.

Principais métodos:

- **add(Component, int)**: adiciona um componente definindo a sua posição.
- **setLayout(LayoutManager)**: altera o tipo de *layout*.

Dicas de uso:

- Antes de inserir qualquer componente no **JFrame**, recomenda-se começar com um **JPanel** para garantir o posicionamento correto dos componentes.
- Caso você não utilize o **JPanel**, os componentes ficarão “soltos” e poderão se comportar de maneira inesperada quando o usuário aumentar ou diminuir a janela do programa.
- Você pode alterar a cor do **JPanel** clicando com o botão direito sobre o componente e acessando a opção **Properties**. Nela, você terá o campo **Background**, que permite você escolher uma cor para o seu **JPanel**.

Label

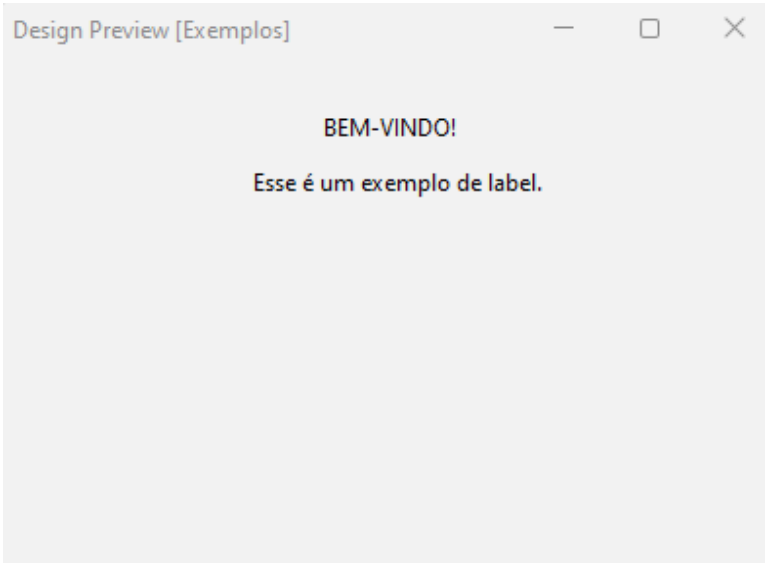


Figura 6 – Exemplo de **JLabel**
Fonte: Apache NetBeans IDE (2022)

Nome da classe: JLabel

Descrição: Permite exibir um texto simples não editável, podendo ser utilizada também para exibir ícones e imagens.

Principais métodos:

- **setText(String)**: altera o texto do componente.
- **getText()**: retorna o texto do componente.
- **setIcon(IconImage icon)**: adiciona um ícone ao **JLabel** ao passar como parâmetro o caminho da imagem.

Dicas de uso:

- Você pode alterar o texto das *labels* sem precisar ir até o código-fonte. Para isso, dê um duplo clique sobre a *label* inserida na tela e então digite a palavra que você deseja inserir.
- A maneira mais simples de alterar a cor, a fonte e o tamanho do texto na *label* é por meio da janela de propriedades da *label*, que pode ser acessada clicando-se com o botão direito sobre o componente inserido e selecionando a opção **Properties**. Nessa tela, você terá acesso às propriedades **Foreground** (para alterar a cor da fonte) e **Font** (para alterar fonte, estilo e tamanho do texto).

Text field

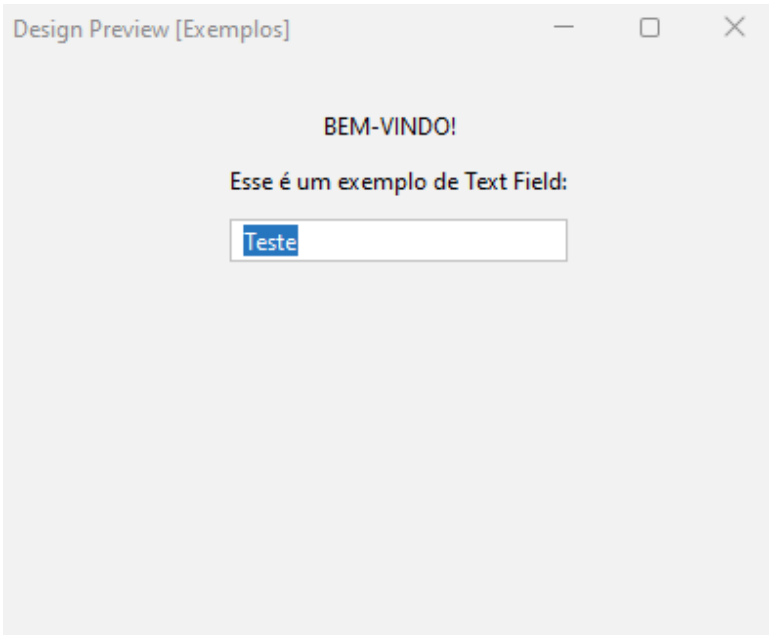


Figura 7 – Exemplo de **JTextField**
Fonte: Apache NetBeans IDE (2022)

Nome da classe: JTextField

Descrição: é o campo de texto no qual o usuário pode digitar um texto em uma linha.

Principais métodos:

- **setText(String)**: altera o texto do componente.
- **getText()**: retorna o texto do componente.

Password field

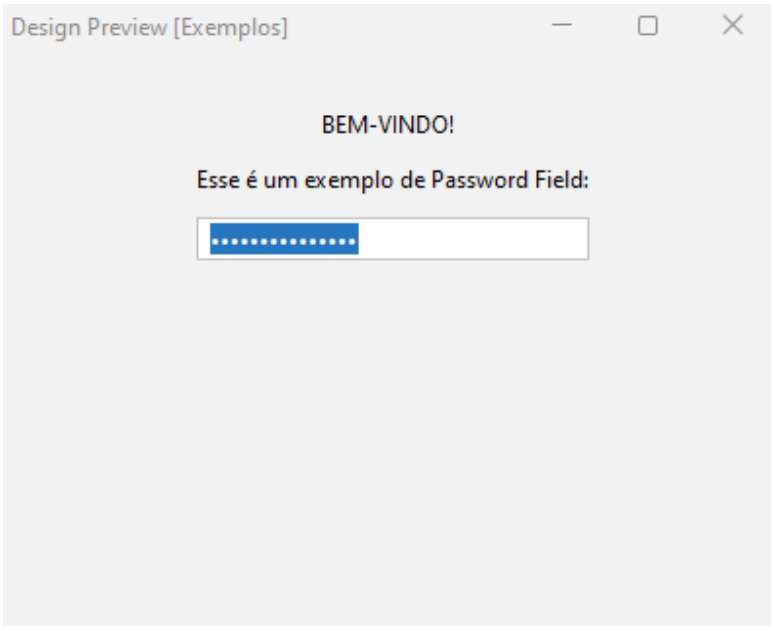


Figura 8 – Exemplo de **JPasswordField**
Fonte: Apache NetBeans IDE (2022)

Nome da classe: JPasswordField

Descrição: permite ao usuário inserir uma entrada de texto, mas que será exibida de modo camuflado, sendo cada caractere um asterisco.

Principais métodos:

- **setEchoChar(char)**: define o caractere que será exibido ao digitar.

Text area

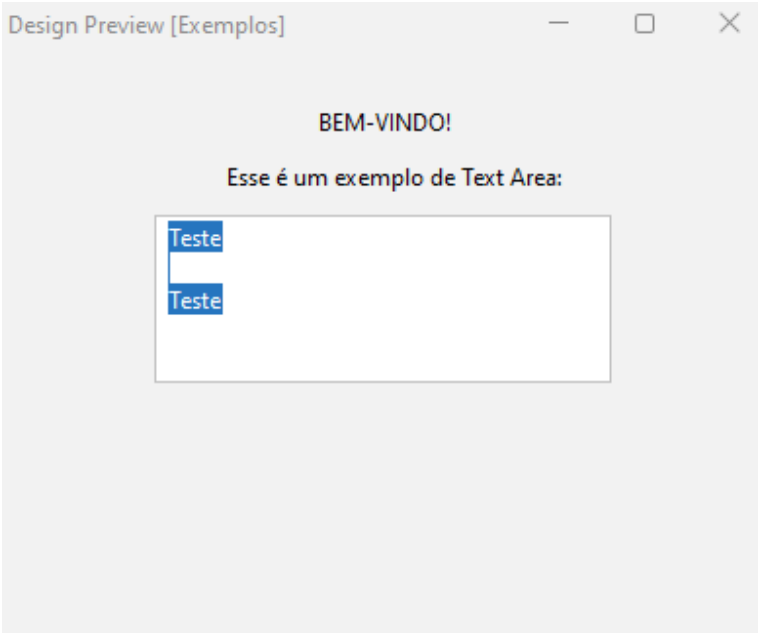


Figura 9 – Exemplo de **JTextArea**
Fonte: Apache NetBeans IDE (2022)

Nome da classe: JTextArea

Descrição: é similar à classe **JTextField**, que permite ao usuário digitar algo. Porém, ao invés de digitar apenas uma linha de texto, o **JTextArea** cria uma caixa de texto que permite a digitação em várias linhas.

Principais métodos:

- **setText(String)**: altera o texto do componente.
- **getText()**: retorna o texto do componente.
- **getSelectedText()**: retorna o texto selecionado pelo usuário.
- **insert(String, int)**: insere um texto na linha especificada.

Checkbox

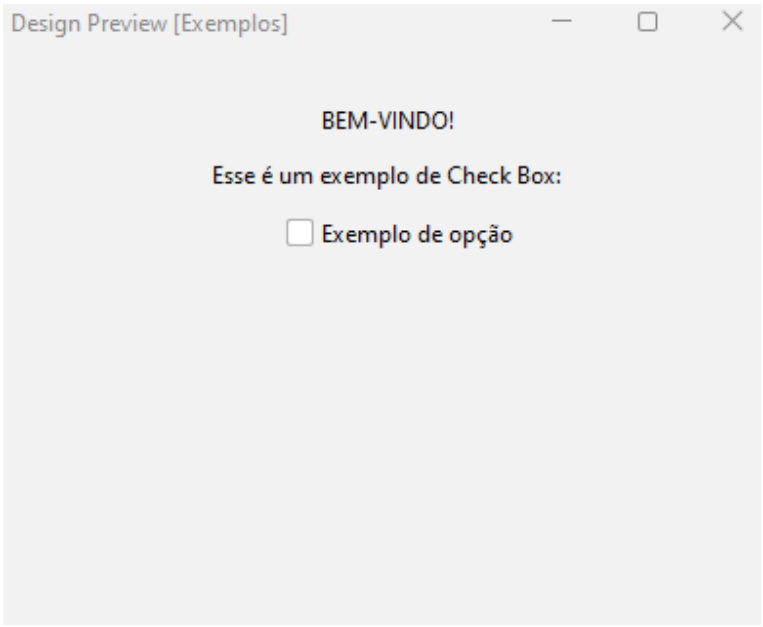


Figura 10 – Exemplo de **JCheckBox**
Fonte: Apache NetBeans IDE (2022)

Nome da classe: JCheckBox

Descrição: caixa de seleção que permite ao usuário selecionar ou não uma opção.

Principais métodos:

- **setText(String string)**: define o texto do item.
- **setSelected(Boolean b)**: se o parâmetro for *true*, o item fica selecionado.

Radio button

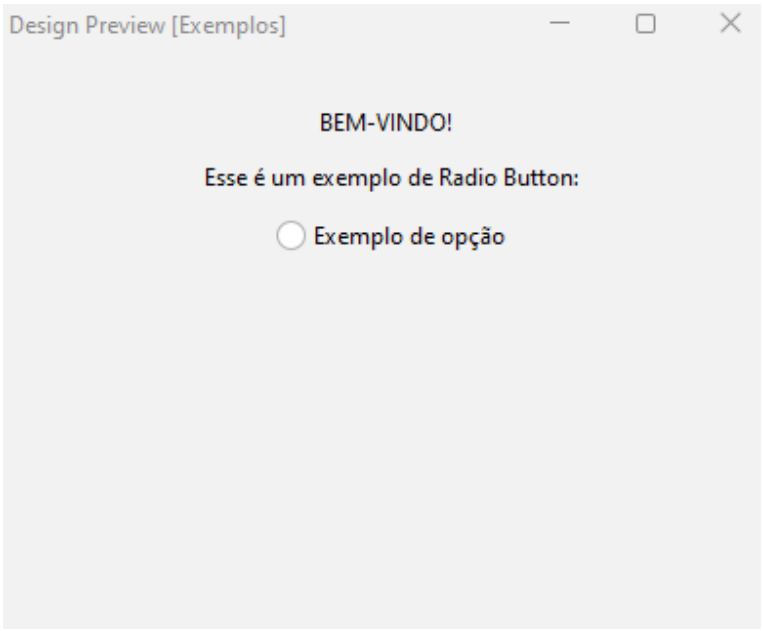


Figura 11 – Exemplo de **JRadioButton**
Fonte: Apache NetBeans IDE (2022)

Nome da classe: JRadioButton

Descrição: componente que permite selecionar uma entre várias opções. É semelhante à classe **JCheckBox** e por isso contém os mesmos construtores e métodos.

Principais métodos:

- **setText(String string)**: define o texto do item.
- **setSelected(Boolean b)**: se o parâmetro for *true*, o item fica selecionado.

Combobox

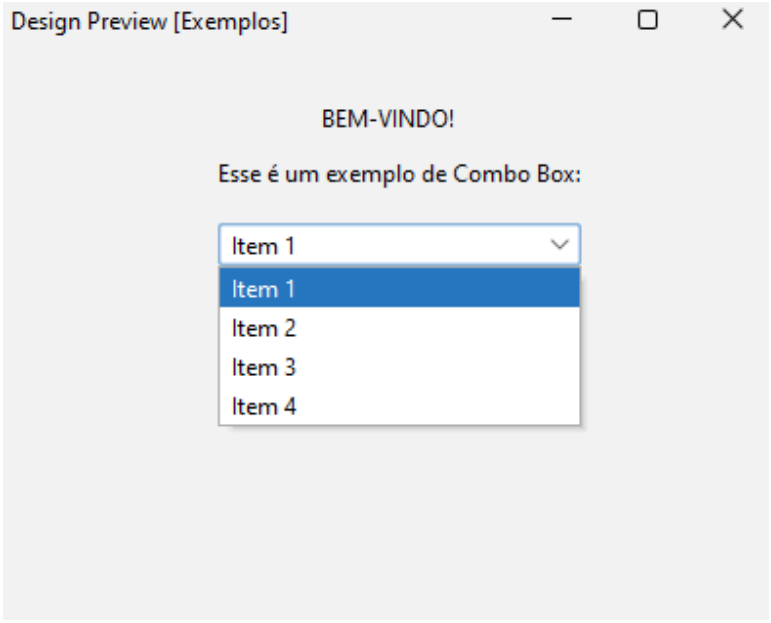


Figura 12 – Exemplo de **JComboBox**

Fonte: Apache NetBeans IDE (2022)

Nome da classe: JComboBox

Descrição: é uma caixa de combinação na qual o usuário pode selecionar uma opção.

Principais métodos:

- **addItem(Object obj)**: é utilizado para adicionar um item no *combo box*. Normalmente neste caso utiliza-se uma *string*.
- **setSelectedIndex(int index)**: define qual item do *combo box* começa selecionado.

Dica de uso:

- Você consegue alterar as opções apresentadas no **JComboBox** diretamente pelo GUI Builder do Apache NetBeans IDE. Para isso, clique com o botão direito sobre o componente e acesse as propriedades clicando em **Properties**. Uma nova janela abrirá e na *model* você terá acesso aos itens da lista – cada linha representa um item. Depois de fazer as alterações, basta clicar em **OK**.

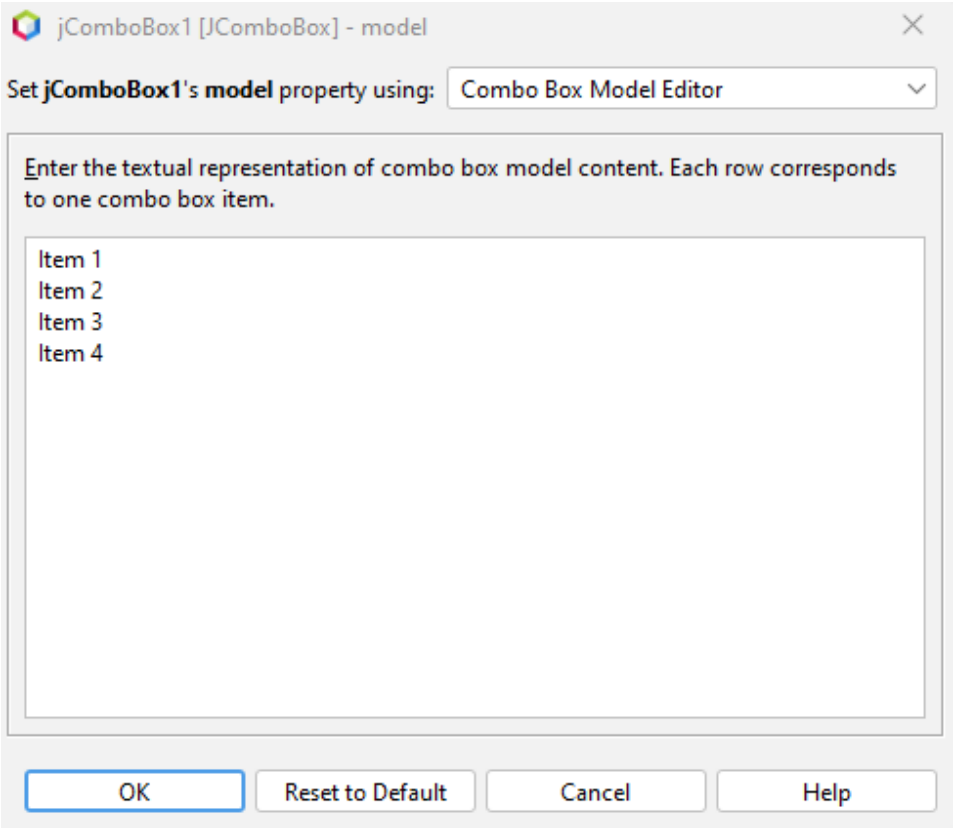


Figura 13 – Propriedades do componente **JComboBox**

Fonte: Apache NetBeans IDE (2022)

List

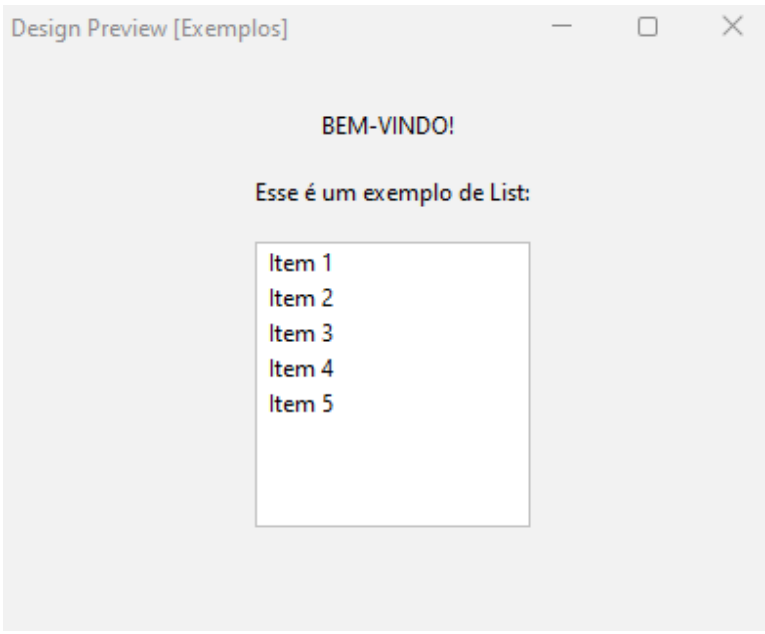


Figura 14 – Exemplo de **JList**
Fonte: Apache NetBeans IDE (2022)

Nome da classe: JList

Descrição: é uma lista de opções que permite a seleção de mais de um item simultaneamente.

Principais métodos:

- **setModel(ListModel list)**: permite adicionar/atualizar os dados na lista, ou seja, os tipos de dados passados como parâmetros em um **ListModel**. Para adicionar valores ao **ListModel**, utilize o método **addElement()**.

Dica de uso:

- Assim como em **JComboBox**, é possível definir os itens que estarão presentes no componente **JList** clicando com o botão direito do mouse sobre o componente, selecionando item "Properties" e depois a propriedade "model".

Button

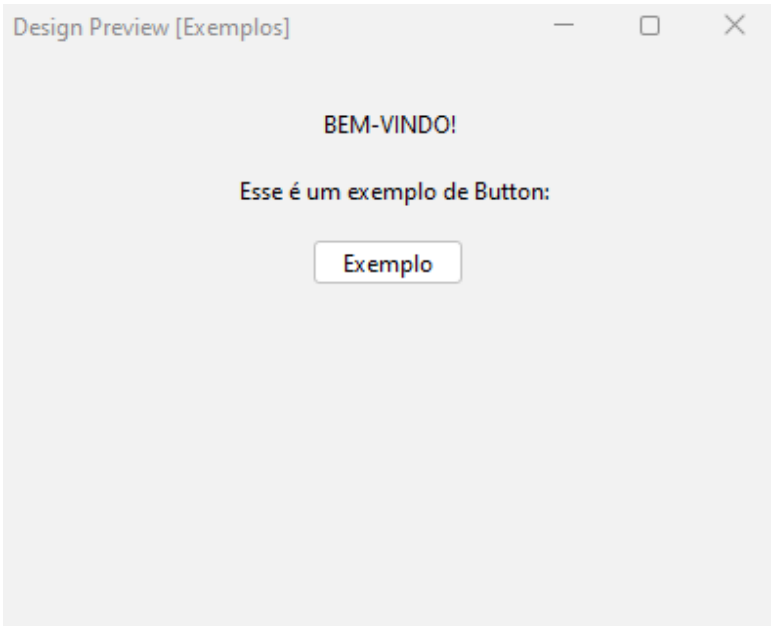


Figura 15 – Exemplo de **JButton**

Fonte: Apache NetBeans IDE (2022)

Nome da classe: JButton

Descrição: é um botão com um texto, que permite a interação do usuário por meio de um clique.

Principais métodos:

- **setText(String string)**: define o texto que será exibido no botão.
- **setEnabled(Boolean enabled)**: se o parâmetro for *true*, significa que o usuário pode clicar no botão. O botão ficará bloqueado, caso o valor seja *false*.
- **setIcon(Icon defaultIcon)**: define um ícone para o botão – deve-se passar o caminho do ícone.

Table

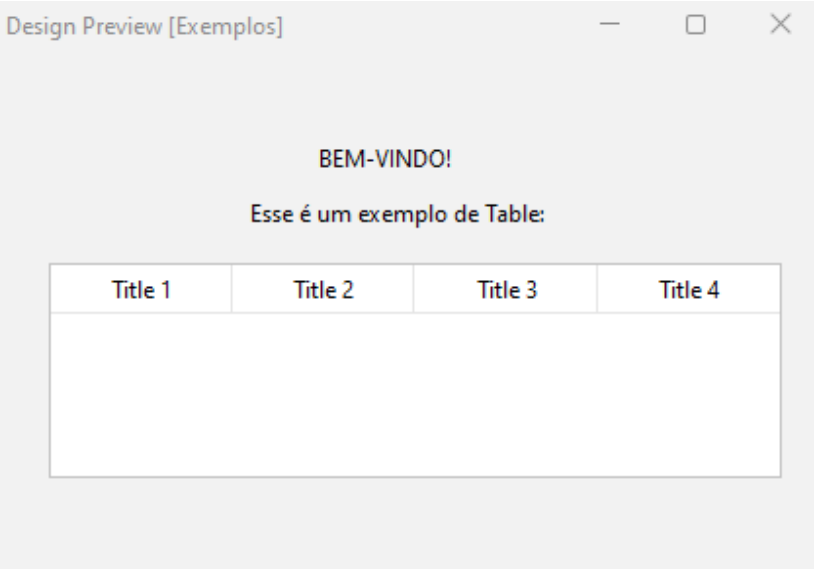


Figura 16 – Exemplo de **JTable**

Fonte: Apache NetBeans IDE (2022)

Nome da classe: JTable

Descrição: cria uma tabela composta por cabeçalho, linhas e colunas.

Principais métodos:

- **addColumn(TableColumn[] column)**: adiciona colunas ao fim da tabela.
- **clearSelection()**: limpa seleção da tabela.
- **setValueAt(int linha, int coluna)**: altera o valor da célula na linha e coluna informadas.

Dica de uso: normalmente, as tabelas são responsáveis por exigir uma grande quantidade de dados. Sendo assim, o tamanho padrão da janela do programa nem sempre será o suficiente para exigir todas as linhas. Por isso, recomenda-se que as tabelas sejam inseridas dentro do componente **Scroll Pane**, que automaticamente criará uma barra de rolagem quando os dados não couberem na tela.

Por exemplo:

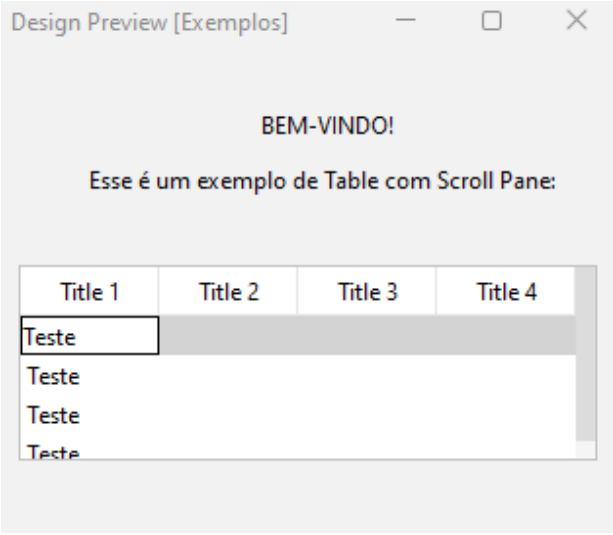


Figura 17 – Exemplo de **JScrollPane**
Fonte: Apache NetBeans IDE (2022)

Dentro do GUI Builder do Apache NetBeans IDE existem muitos outros componentes à disposição para você construir suas telas. Agora que você já sabe como usar os principais componentes, sinta-se livre para experimentá-los e, sempre que possível, confira as atualizações diretamente na documentação do Java.

Manipulação de eventos e uso de controles

Para o funcionamento completo de um *software*, não basta ter uma interface gráfica bonita, é preciso que o sistema responda às interações do usuário corretamente. Para isso, considere o seguinte algoritmo:

1. Enquanto o programa estiver rodando, verifica-se se houve alguma interação do usuário. Por exemplo: se o *mouse* foi movido ou se alguma tecla foi pressionada.
2. Caso tenha havido interação, realiza-se alguma ação.
3. Segue-se repetindo essa verificação enquanto o programa estiver rodando.

Em Java, para esse caso, haveria mais ou menos a seguinte estrutura de código:

```
while(programaRodando == true) {
  boolean resposta = verificaSeHouveInteracao();

  if(resposta == true) {
      // Realizamos alguma ação
  }
}
```

Porém, essa solução não é considerada eficiente, pois seria necessário explicitamente “ler” os periféricos (*mouse* e teclado) para tratar as ações do usuário dentro do programa. Caso você siga essa ideia, perderia muito tempo e

processamento para fazer essas leituras, causando lentidões nas outras tarefas que deveriam ser feitas pelo sistema. Felizmente, a linguagem de programação Java contém um paradigma no qual se pode executar trechos de códigos com o disparo de eventos. Isso se chama “programação orientada a eventos”.

A linguagem de programação orientada a eventos tem o usuário como o papel principal na execução das rotinas que compõem um programa. Essas ações ou rotinas são disparadas por ele por meio da interação com o programa. Após isso, os eventos são capturados pelo ambiente para acionar funções escritas pelo programador. Na prática, isso ocorreria da seguinte forma:

1. Sempre que o usuário interage com o *mouse* ou o teclado, o sistema operacional gera um **evento**.
2. Se a aplicação (imagine uma aplicação Java) estiver interessada em um evento específico (por exemplo, clique do *mouse* no botão **Salvar**), deve solicitar ao sistema operacional para **escutar** o evento.
3. Se aplicação não estiver interessada, seu processamento continuará normalmente.

Perceba que a aplicação não é responsável por identificar a ocorrência de eventos, pois o responsável é o sistema operacional.

Esse paradigma da programação orientada a eventos em que o sistema operacional avisa o programa que determinado evento aconteceu é baseado no Princípio de Hollywood: “*don’t call us, we’ll call you*”, ou, em português, “não nos chame, nós chamaremos você”.

Alguns exemplos de eventos são:

- Clicar sobre um botão: evento *click*
- Digitar caracteres dentro de uma caixa de texto: evento *change*
- Passar o ponteiro do *mouse* sobre um componente: evento *MouseMove*
- Pressionar uma tecla: evento *KeyPress* ou evento *KeyDown*
- Liberar um botão do *mouse* após tê-lo pressionado: evento *MouseUp*
- Focalizar um componente: evento *SetFocus*

Na linguagem de programação Java, há diversas classes para se trabalhar com o tratamento de eventos. Essas classes são chamadas de interfaces ouvintes de eventos e fazem parte do pacote **java.awt.event**.

As interfaces ouvintes de eventos são chamadas assim porque são responsáveis por “escutar” se determinado evento ocorreu. Se o evento ocorrer, o sistema operacional irá informá-la e outro evento poderá ocorrer. Em outras palavras, elas exercem o papel de um ***listener*** ou “ouvinte”. Cada *listener* utiliza uma classe específica para representar o evento, de acordo com o tipo que será gerado.

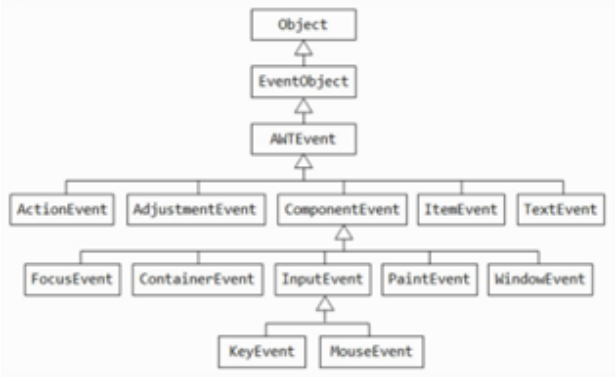


Figura 18 – Hierarquia de classes do pacote **AWTEvent**
Fonte: https://www.falkhausen.de/Java-8/java.awt/event/Event-Hierarchy.html

Observação

O pacote **AWTEvent** oferece várias interfaces para ouvintes, mas, para fins didáticos, este estudo focará apenas nas principais interfaces.

Ação do usuário	Evento disparado	Ouvinte do evento
Clicar em um componente	**ActionEvent**	**ActionListener**
Clicar em um item de escolha (**JCheckBox**)	**ItemEvent**	**ItemListener**
Pressionar uma tecla do teclado	**KeyEvent**	**KeyListener**
Clicar em um componente (**JComponent**)	**MouseEvent**	**MouseListener**
Abrir, fechar, minimizar ou maximizar uma janela	**WindowEvent**	**WindowListener**

Para que você compreenda como cada recurso funciona na prática, será criado um novo projeto Java no Apache NetBeans e serão apresentados alguns casos de uso para cada evento.

Observação

Para fins didáticos, você verá exemplos em código Java que não utilizam o construtor de interface (GUI Builder) do Apache NetBeans IDE. Para isso, uma classe chamada **Exemplo** será criada e utilizada para escrever os códigos dos exemplos a seguir.

Encerramento

Com isso, foi concluída a primeira parte do conteúdo sobre a criação de interfaces gráficas em Java para *desktop*. Até aqui, você aprendeu a teoria e a prática referentes a como construir telas com o GUI Builder e manipular eventos com seus respectivos controles. Na próxima parte, você colocará em prática tudo que aprendeu em um projeto completo com janelas, formulários, listagens e funcionalidades de cadastro e listagem de dados.

Desenvolvimento de Sistemas

Interface *desktop*: construção de interface de usuário, manipulação de eventos, uso de controles, manipulação de janelas, construção de formulários e listagens (Parte 2)

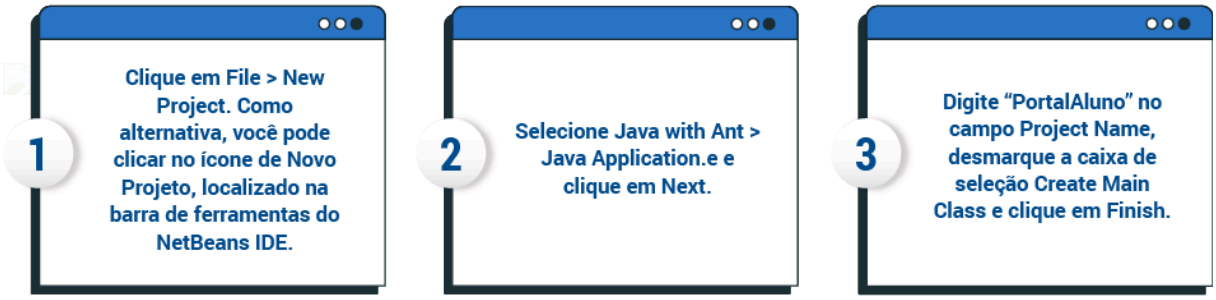
Construção de telas, formulários e listagens

Agora que você já conhece os principais componentes do Swing e sabe como trabalhar com eventos, será preciso colocar em prática tudo o que aprendeu. Para isso, você criará um projeto que englobará todos os conhecimentos desenvolvidos até este momento.

Nota: caso não tenha estudado o conteúdo **Interface *desktop*... (Parte 1)**, recomenda-se que faça isso antes de prosseguir seu estudo neste conteúdo.

Criando um projeto Java com interface gráfica para *desktop*

Para começar, você criará um novo projeto Java no Apache NetBeans IDE. Esse projeto será um sistema para cadastrar e consultar alunos matriculados em uma instituição de ensino. Então, esse projeto será chamado de “PortalAluno”.



Para esse projeto, você deve criar três telas diferentes:

a. Tela inicial

b. Tela de cadastro

c. Tela de listagem

Passo 1

Depois de criar o projeto, crie um novo pacote Java com o nome **br.com.senacead.portalaluno.telas**, pressionando o botão direito do *mouse* sobre o pacote **portalaluno** e selecionando **New > Java Package**. Após isso, o projeto terá a seguinte estrutura:

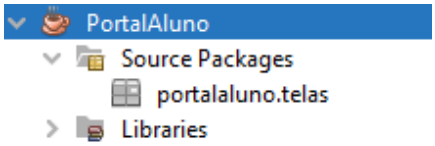


Figura 1 – Estrutura de pastas do projeto PortalAluno
Fonte: Apache NetBeans IDE (2022)

Passo 2

Agora, crie uma tela. Para isso, clique com o botão direito do *mouse* sobre o pacote recém-criado e selecione as opções **New** > **JFrame Form**.

Passo 3

Uma nova janela será aberta, na qual você deverá especificar o nome da classe (*class name*). Para manter uma boa organização no projeto, o ideal é criar as classes com um nome referente ao conteúdo da tela. Crie uma tela inicial para seu projeto. Então, você poderá chamar a classe de **TelaInicial**. Após informar o nome da classe, clique em **Finish**.

Passo 4

Repita o processo e crie mais duas telas: **Cadastro** e **Listagem**.

No final, você terá a seguinte estrutura de arquivos no Apache NetBeans IDE:

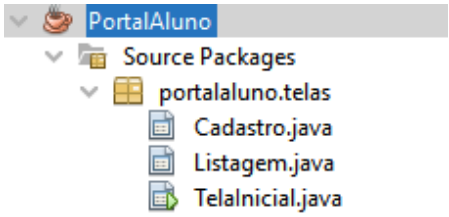


Figura 2 – Estrutura de pastas do projeto “PortalAluno”
Fonte: Apache NetBeans IDE (2022)

Construindo telas do projeto

Embora o GUI Builder do IDE simplifique o processo de criação de GUIs Java, geralmente é útil esboçar a aparência da interface antes de começar a organizá-la por meio de *wireframes*, por exemplo. Muitos *designers* de interface consideram essa uma técnica de "prática recomendada", no entanto, para os propósitos deste conteúdo, será deixada uma prévia das telas para você ter uma ideia do que deve ser construído.

Navegação entre telas

Tela inicial

O objetivo neste momento é simples:

- Quando o usuário clicar no botão **Listagem de alunos**, a tela **Listagem.java** deverá se abrir.
- Quando o usuário clicar no botão **Cadastro de alunos**, a tela “Cadastro.java” deverá se abrir.
- Quando o usuário clicar no botão **Sair**, a tela atual deverá se fechar.

Por padrão, o GUI Builder cria cada componente com um nome de variável que seja facilmente identificado no código-fonte. Para isso, utiliza-se o nome do próprio componente. No caso dos botões nos quais você está trabalhando, existem os seguintes nomes:

- Botão de listagem: **jButton1**
- Botão de cadastro: **jButton2**
- Botão de sair: **jButton3**

Dependendo da ordem na qual você criou os seus botões, é possível que eles tenham outro nome. Para confirmar, clique no item que aparece na listagem do canto inferior esquerdo do Apache NetBeans IDE e repare qual componente o GUI Builder destacará.

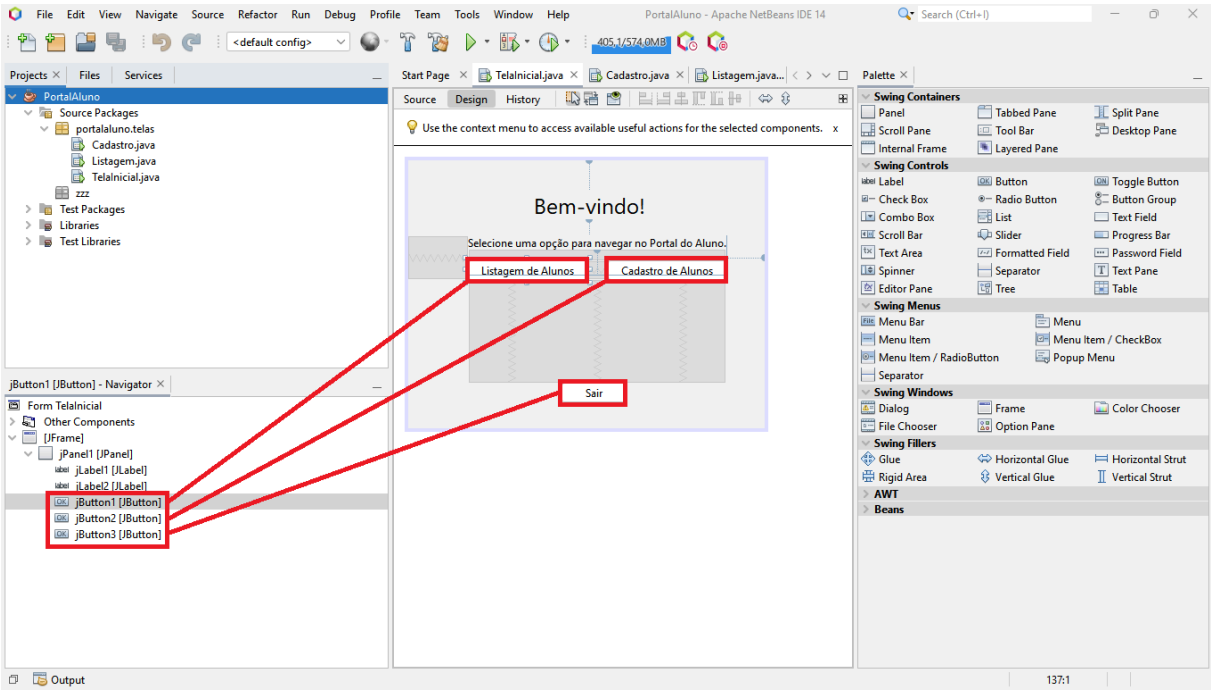


Figura 8 – GUI Builder do Apache NetBeans IDE
Fonte: Apache NetBeans IDE (2022)

Para que você não perca o controle do seu código, é interessante definir os nomes das suas variáveis ao invés de seguir o que o GUI Builder definiu. Afinal, futuramente, ficaria muito difícil diferenciar o **jButton3** do **jButton66**. Então, altere o nome da variável de cada botão. Para isso, clique com o botão direito do *mouse* sobre o componente, selecione a opção **Change Variable Name...** e defina os seguintes nomes para cada botão:

- Listagem de alunos: **botaoListagem**
- Cadastro de alunos: **botaoCadastro**
- Sair: **botaoSair**

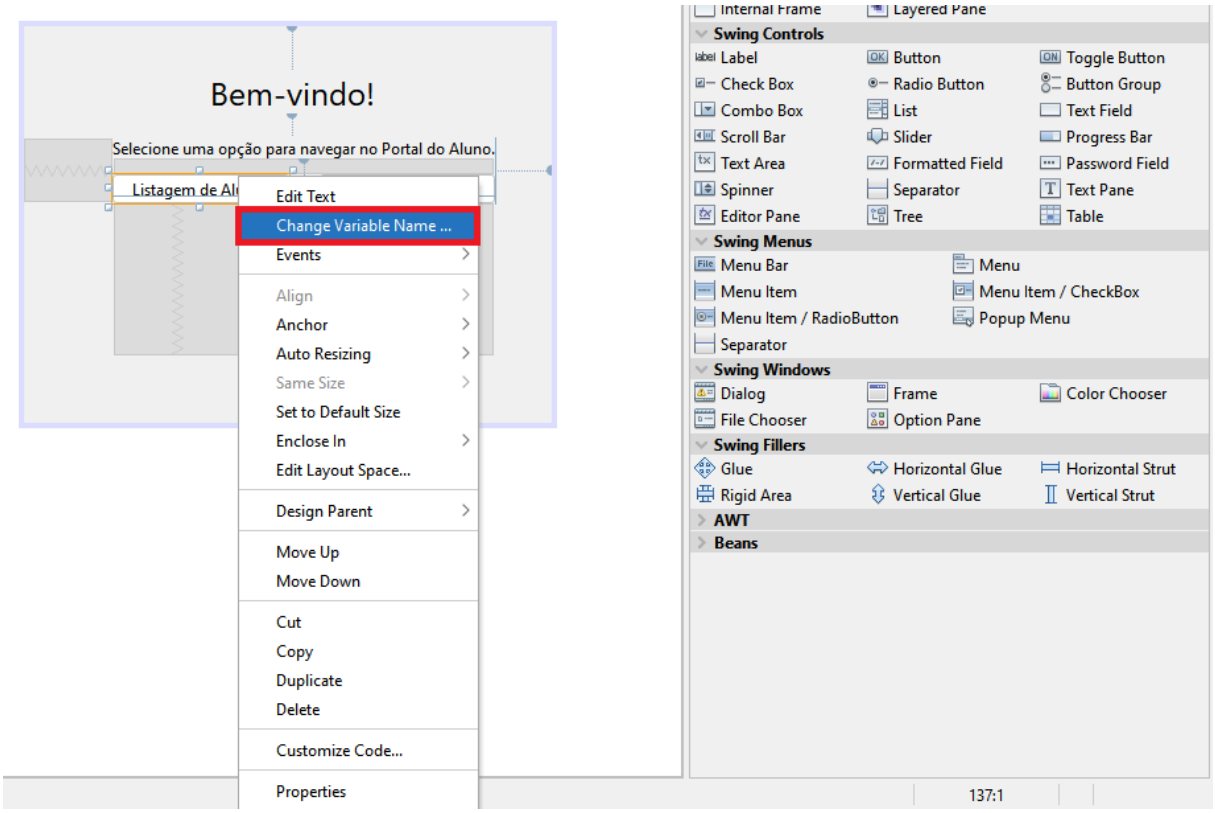


Figura 9 – Acessando a opção **Change Variable Name...** no GUI Builder
Fonte: Apache NetBeans IDE (2022)

Agora, você já tem uma definição clara para cada botão e sabe diferenciar um do outro.

Observação

Outra forma de alterar o nome da variável de um componente é acessando **Properties** > **Code** por meio do GUI Builder e alterando o campo **Variable Name**.

Com as variáveis definidas, agora você pode se concentrar nos eventos de cada botão. Para adicionar o evento de *action*, clique com o botão direito do *mouse* sobre o botão no qual você quer adicionar o evento e selecione as opções **Events** > **Action** >

ActionPerformed. Comece com o botão **Listagem de alunos**.

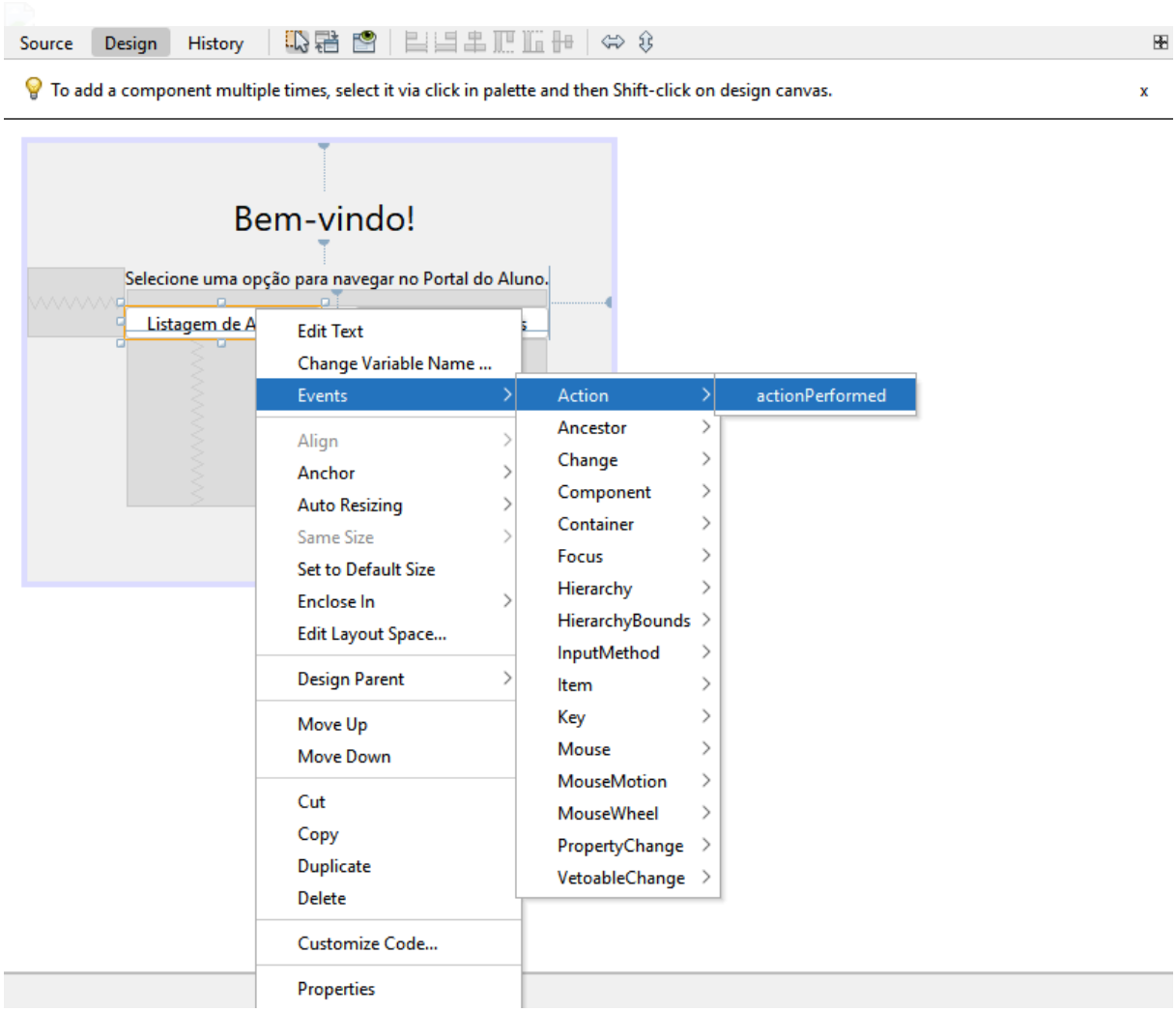


Figura 10 – Acessando eventos do componente no GUI Builder

Fonte: Apache NetBeans IDE (2022)

Isso levará você até a aba de código-fonte na qual terá o seguinte trecho de código focado:

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

Perceba que o próprio GUI Builder já se encarregou de criar e configurar o *listener* e o método para se acionar um evento utilizando a variável do componente para nomear o método. Logo, tudo que você precisa fazer é adicionar o trecho de código que abrirá a tela **Listagem.java**. O trecho em questão terá a seguinte estrutura:

```
private void botaoListagemActionPerformed(java.awt.event.ActionEvent evt) {
  Listagem telaListagem = new Listagem();
  telaListagem.setVisible(true);
}
```

O que você fez aqui foi iniciar a tela na memória, instanciando-a como o objeto **telaListagem**. Por meio desse objeto, você aciona o método **setVisible(true)** para que a tela seja aberta para o usuário.

Para o botão de cadastro, repita o procedimento. Porém, neste caso, não se quer abrir a tela de listagem, e sim a de cadastro. Logo, seu código sofrerá alterações e ficará da seguinte maneira:

```
private void botaoCadastroActionPerformed(java.awt.event.ActionEvent evt) {
  Cadastro telaCadastro = new Cadastro();
  telaCadastro.setVisible(true);
}
```

Por fim, adicione o trecho de código que “fechará” a tela inicial. Para isso, você deve estar imaginando usar o método **setVisible(false)**, correto? Mas, se fizermos isso, a tela ainda existirá na memória do computador sem o usuário ver. Para fechar a tela e encerrar o programa, você deve utilizar o método **System.exit(0)**, da seguinte maneira:

```
private void botaoSairActionPerformed(java.awt.event.ActionEvent evt) {
  System.exit(0);
}
```

Agora que sua tela inicial está 100% funcional, implemente a navegação nas outras telas.

Listagem de alunos e Cadastro de alunos

Para ambas as telas, você realizará o mesmo procedimento, já que que ambas contêm apenas um botão voltado para a navegação:

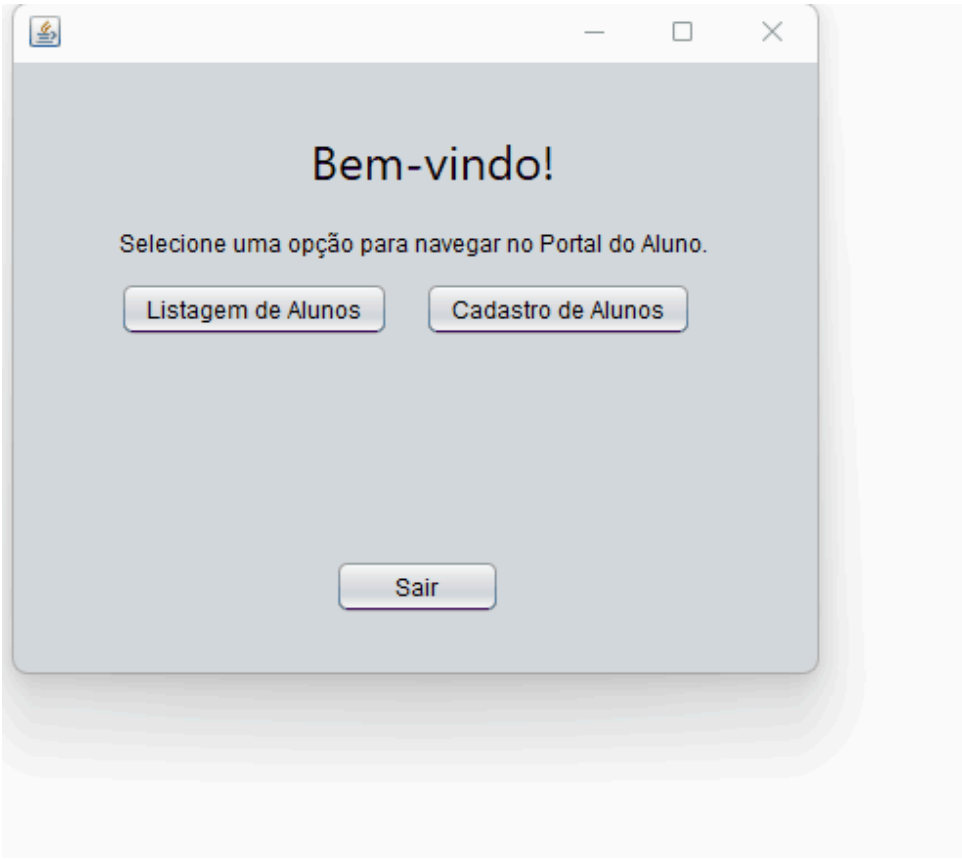
1. Comece alterando o nome da variável do botão **Voltar** para **botaoVoltar**.
2. Após essa alteração, no código-fonte, adicione a ação ao **ActionEvent**.
3. Adicione o seguinte trecho de código:

```
private void botaoVoltarActionPerformed(java.awt.event.ActionEvent evt) {
  dispose();
}
```

Diferentemente do método **setVisible()**, que apenas oculta a tela do usuário enquanto continua na memória, o método **dispose()** fecha a tela e a “limpa” da memória. Perceba também que, nesse caso, não é preciso instanciar a classe da tela como um objeto para acessar o método, porque você já está na tela que quer manipular. Portanto, basta acionar o método direto.

Após implementar esse trecho em ambas as telas, você terá todas as telas com a navegação funcional.

Analise o GIF a seguir:



Implementando as classes de manipulação de dados

Para cadastrar o aluno, são necessárias duas classes Java. A primeira será a classe **Aluno**, responsável por guardar os dados “nome”, “e-mail” e “curso”. Já a segunda, será a classe **ListaAluno**, na qual se pode guardar vários objetos da classe **Aluno** por intermédio de uma lista.

1. Primeiramente, crie um novo pacote dentro de **portalaluno** chamado *model*. Dentro desse pacote, você criará as classes de manipulação de dados do seu sistema.

2. Após a criação do pacote, crie duas novas classes Java: **Aluno** e **ListaAluno**.

A sua estrutura de arquivos no Apache NetBeans IDE ficará assim:

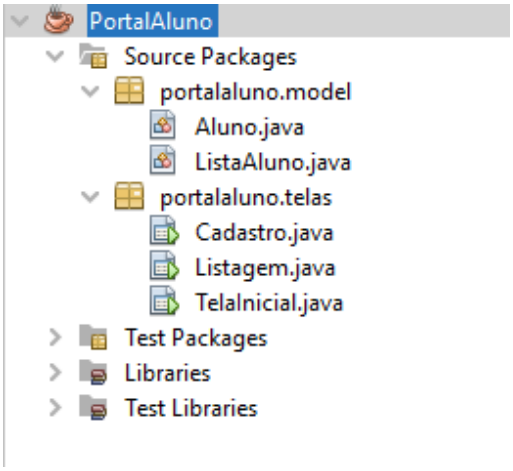


Figura 11 – Estrutura de arquivos do projeto “PortalAluno”

Fonte: Apache NetBeans IDE (2022)

A classe **Aluno** terá o seguinte código:

```
package portalaluno.model;

public class Aluno {

    // Declaração de variáveis
    private String nome;
    private String email;
    private String curso;

    // Métodos para acessar e atualizar dados das variáveis
    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public String getCurso() {
        return curso;
    }

    public void setCurso(String curso) {
        this.curso = curso;
    }

}
```

Já a classe **ListaAluno** terá o seguinte código:

```
package portalaluno.model;

// Importação dos pacotes necessários para usarmos o List e ArrayList
import java.util.ArrayList;
import java.util.List;

public class ListaAluno {
    // Declaração de variáveis
    private static final List<Aluno> lista = new ArrayList<>();

    // Métodos para acessarmos a lista e adicionarmos novos itens
    public static List<Aluno> Listar() {
        return lista;
    }

    public static void Adicionar(Aluno aluno) {
        lista.add(aluno);
    }

}
```

Acessando as classes de manipulação de dados

Estando as classes de manipulação de dados prontas, agora você poderá integrá-las às suas telas para tornar seu sistema funcional para o usuário.

Cadastro de aluno

Comece pela tela **Cadastro.java**. A primeira coisa que precisa fazer nessa tela é atualizar os nomes dos seus componentes para conseguir acessá-los corretamente na aba **Source Code**. Novamente, clique com o botão direito do *mouse* sobre o componente, selecione a opção **Change Variable Name...** e defina os seguintes nomes:

- Altere o nome da **JTextField** abaixo de Nome do **aluno** para **nome**.
- Altere o nome da **JTextField** abaixo de **Email** para **email**.
- Altere o nome da **JComboBox** abaixo de **Curso** para **curso**.

No final, você terá a seguinte configuração:

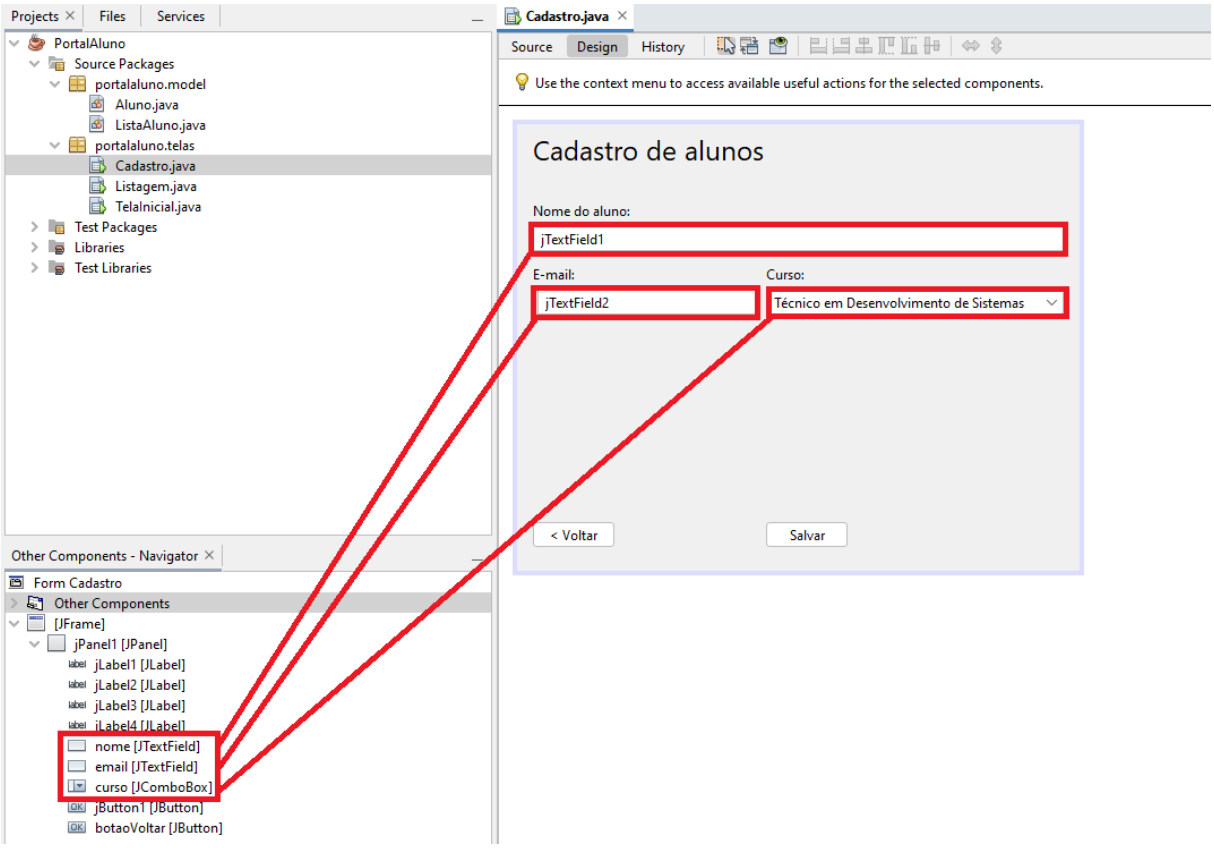


Figura 12 – GUI Builder do Apache NetBeans IDE

Fonte: Apache NetBeans IDE (2022)

Como o que se quer é que o usuário digite os dados em “Nome do aluno” e “E-mail”, convém deixar esses campos limpos para que recebam a entrada de dados do usuário. Então, aproveite e limpe os textos usados como padrões dos componentes **JTextField** clicando com o botão direito sobre cada componente e selecionando **Edit Text...**

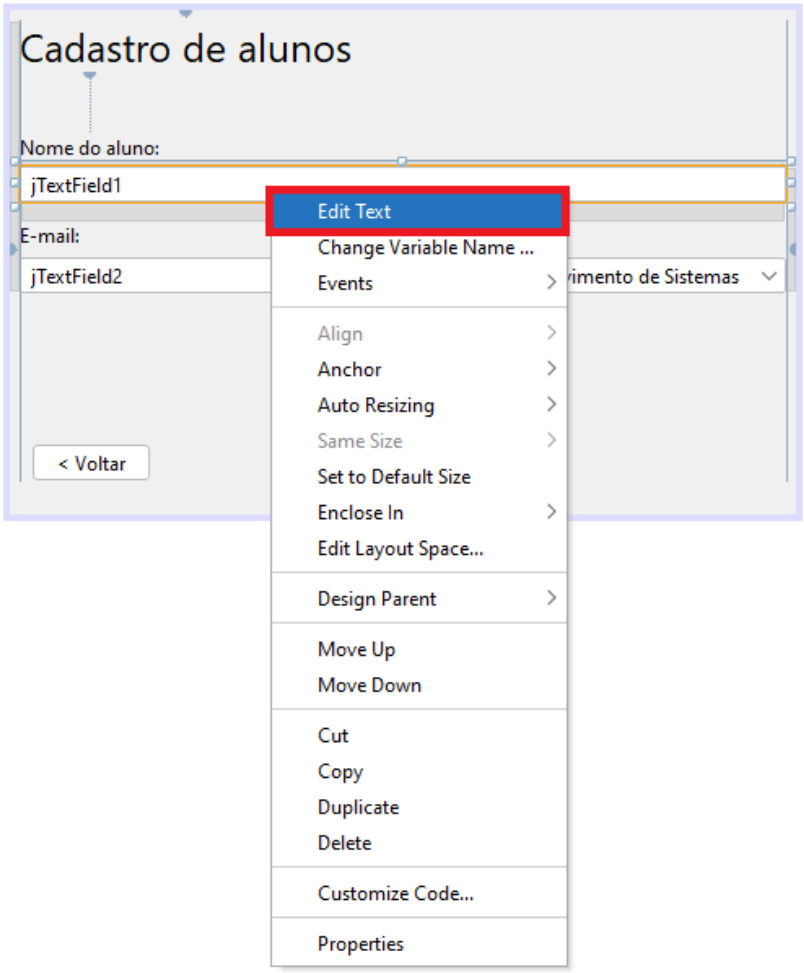


Figura 13 – Acessando o **Edit Text** do componente **JTextField**
Fonte: Apache NetBeans IDE (2022)

Agora, você pode atualizar o nome da variável do botão **Salvar**. Clique com o botão direito do *mouse* sobre o componente, selecione a opção **Change Variable Name...** e altere o nome para **Salvar**. Depois disso, clique com o botão direito do *mouse* sobre o componente e escolha as opções **Events** > **Action** > **ActionPerformed** para escrever o código que será acionado quando o usuário clicar no botão. Você será levado à aba **Source Code** na exata linha para escrever o código do método **SalvarActionPerformed()**.

O código que se quer implementar deve fazer o seguinte:

1. Criar um objeto “aluno” vazio
2. Atualizar os atributos do objeto com os dados do formulário

3. Adicionar o objeto com os atributos definidos a lista de alunos
4. Mostrar uma mensagem para o usuário informando que os dados foram cadastrados com sucesso

Logo, o método **SalvarActionPerformed()** terá o seguinte código:

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

    // Criamos o objeto "aluno" com seus respectivos dados
    Aluno aluno = new Aluno();
    aluno.setNome(nome.getText());
    aluno.setEmail(email.getText());
    aluno.setCurso(curso.getSelectedItem().toString());

    // Adicionamos o aluno a lista
    ListaAluno.Adicionar(aluno);

    // Mostramos os dados para o usuário através de um JOptionPane
    JOptionPane.showMessageDialog(null, "Os seguintes dados foram cadastrados com s
ucesso: \n"
            + "\nNome: " + nome.getText()
            + "\nE-mail: " + email.getText()
            + "\nCurso: " + curso.getSelectedItem().toString()
    );
  }
```

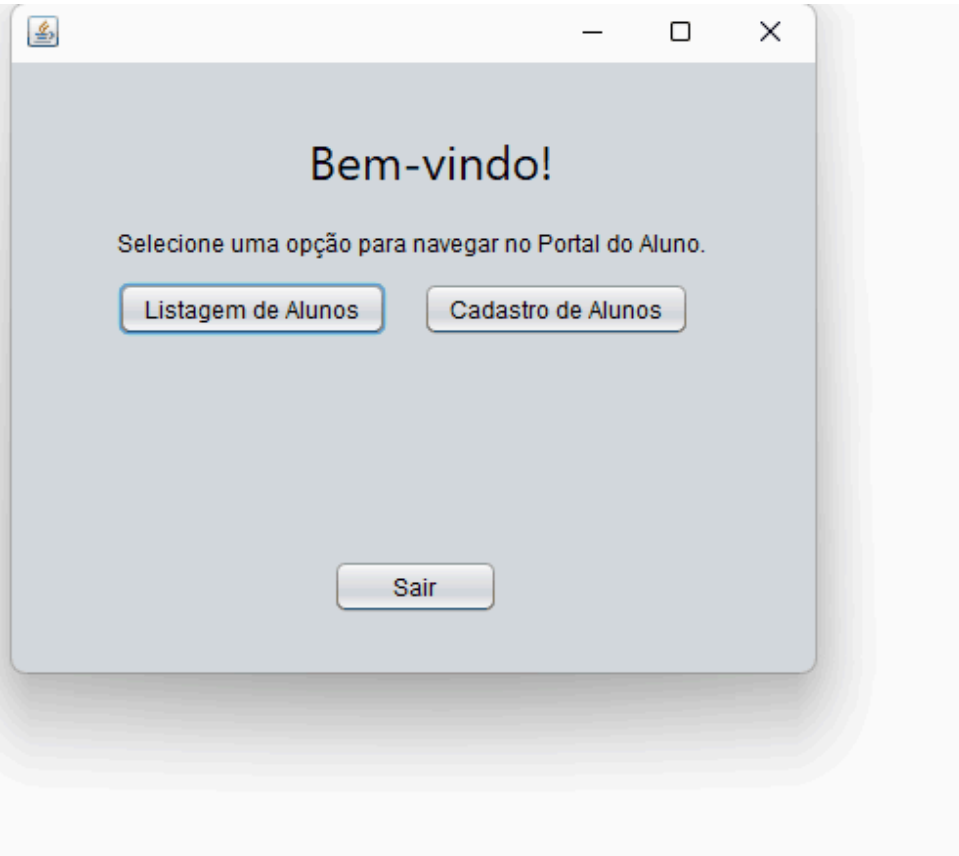
Atenção!

Lembre-se de fazer as importações necessárias no início do código para que as classes Java sejam encontradas e funcionem corretamente.

Figura 14 – **Source Code** do GUI Builder

Fonte: Apache NetBeans IDE (2022)

Se você testar a aplicação agora, verá o resultado mostrado no seguinte GIF:



Esta aplicação está concluída no conteúdo **Estrutura de dados: sintaxe, bibliotecas de linguagem e aplicabilidade**, desta unidade curricular, utilizando **JTable** para a listagem de itens cadastrados. Neste momento, caso queira verificar se a inserção foi realizada corretamente, você pode usar um laço *for*, percorrer a lista e imprimir cada um dos itens na lista em tela usando um **JOptionDialog** ou mesmo em console **(System.out.println())**.

Encerramento

Assim, conclui-se a segunda parte do conteúdo sobre interfaces *desktops*, no qual foram apresentados e explorados os principais componentes visuais da biblioteca Swing de Java. Com o exemplo, foi possível colocar em prática tudo que você aprendeu na primeira parte e construir um sistema capaz de cadastrar dados da memória.

É importante que você siga testando os outros componentes disponíveis e experimente as funcionalidades e configurações deles. Lembre-se de que somente com a prática você ganhará experiência.

Desenvolvimento de Sistemas

Tratamento de exceções em linguagem de programação: comandos, classes, aplicabilidade

Por vezes, programas de computador resultam em erro, e esse erro pode terminar com a execução do programa ou o travamento do sistema por completo. Para contornar isso, algumas linguagens de programação, como Java, contêm o **tratamento de exceção**, que é uma maneira segura de tratar desses erros sem a quebra total do programa.

Imagine os seguintes exemplos que poderiam causar problemas em um programa de computador:

- ➔ Um novo usuário, ao executar o seu cadastro em um sistema, acaba inserindo uma entrada de dados anormal em um campo de cadastro, como um caractere especial em um campo de data ou um hífen em um campo que não aceita esse tipo de caractere.

- ➔ Um arquivo que deve ser lido pelo programa para gravar novos dados encontra-se ausente ou corrompido
- ➔ Um programa tenta executar uma divisão por 0 (zero).

Nos casos mencionados, o tratamento de exceção evitaria com graciosidade que o programa quebrasse totalmente e, ou, parasse de responder. Ou seja, o tratamento de exceção observa e trata erros previsíveis, mostrando,

por exemplo, uma mensagem amigável ao usuário ao invés de encerrar a execução do programa abruptamente.

Como funcionam as exceções

Uma exceção em Java nada mais é do que um **objeto** de uma classe específica (que, como você verá a seguir, é descendente de **Throwable**). Esse objeto especial pode ser criado em situações específicas em que se detecta uma situação de falha. Por exemplo, imagine que você está programando um *software* que lê arquivos no formato TXT; caso algum usuário acabe enviando um arquivo de extensões DOC ou XLS, que não são suportadas, é possível criar e **lançar** uma exceção para sinalizar à execução do programa que ocorreu um problema.

Lançar exceção significa basicamente interromper a execução do código atual, do código que chamou esse código, do código que invocou este outro e assim por diante, até alcançar algum código que faça o **tratamento** dessa exceção.

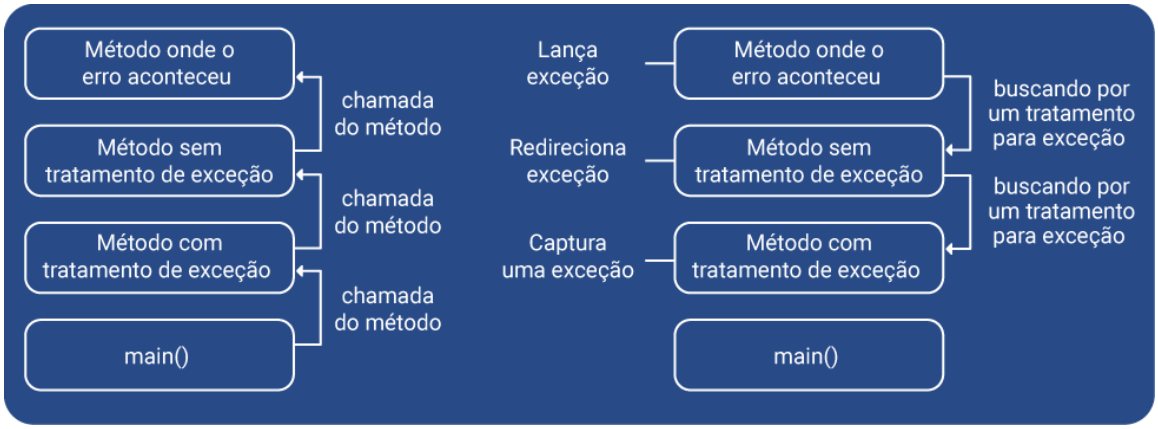


Figura 1 – Fluxo de uma exceção

Fonte: Senac EAD (2022)

Tratar exceção é basicamente proteger um código suscetível à falha (neste exemplo, poderia ser o código responsável por realizar a leitura do arquivo que ele espera que seja TXT) e realizar uma ação específica para essa falha – por exemplo, mostrando uma mensagem ao usuário ou registrando em arquivo log o problema que ocorreu.

Caso **não haja tratamento** para uma exceção lançada, ela alcançará o nível mais básico do programa (o método **main()** em Java), mostrará em console a exceção com mensagens compreensíveis ao programador, mas pouco amigáveis ao usuário, e poderá encerrar a execução do programa. Além de situações em que você mesmo cria objetos de exceção no código desenvolvido, existem casos em que Java cria exceções internamente em operações implementadas na linguagem. Por exemplo, quando ocorre a manipulação de um vetor e utiliza-se um índice inválido, internamente Java cria um objeto de exceção para alertar que não é possível acessar a posição inexistente do vetor.

Hierarquia de exceções em Java

Na linguagem Java, todas as exceções são representadas por classes e todas as classes de exceção derivam de uma classe chamada **Throwable** sendo assim, ao ocorrer uma exceção, algum objeto de algum tipo será instanciado (criado). Existe também na linguagem dois tipos de subclasses que são **Throwable**: as classes de exceção e as de erro. Exceções do tipo erro dizem respeito a erros que ocorrem na JVM (Java Virtual Machine, ou em português, Máquina Virtual Java) e não no programa em questão. Esse tipo de exceção está fora do controle do programador e não será tratado por ele. Logo, este texto tratará da subclasse de exceção (**Exception**), que são as exceções

que podem, e devem, ser tratadas pelo programador, como nos casos citados anteriormente. Uma importante subclasse de exceção é a **RuntimeException**, que é usada para representar diversos erros comuns durante a execução de um programa em Java.

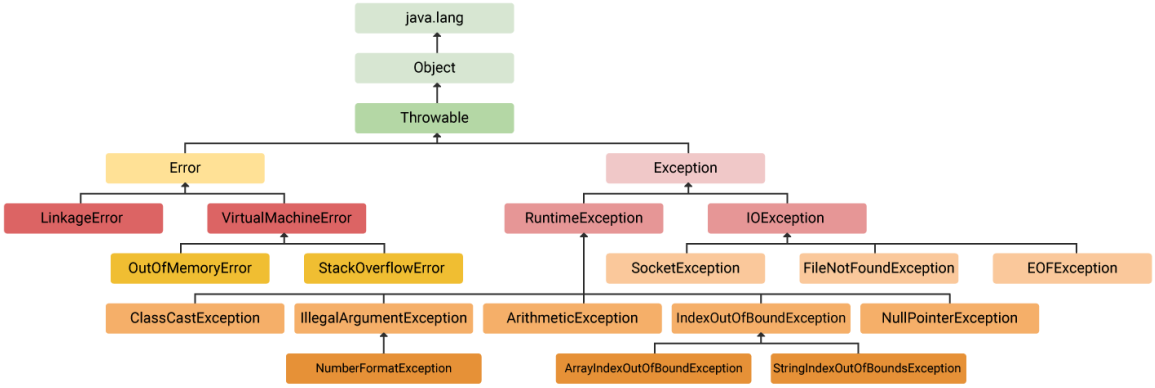


Figura 2 – Hierarquia de classes de exceção e de erros

Fonte: Adaptado de Punchihewa (2021)

Comandos e aplicabilidade

Todo o tratamento de exceção em Java ocorre por meio de cinco palavras-chave: ***try***, ***catch***, ***throw***, ***throws*** e ***finally***. Estas formam uma espécie de subsistema em que uma acaba referenciando a outra.

Partes do programa e trechos de código que você desejar monitorar por exceções deverão sempre conter um bloco de código ***try***. Caso ocorra uma exceção nesse bloco, o seu código poderá capturar essa exceção usando o comando ***catch***. Para “lançar” manualmente uma exceção, use o comando ***throw***. Em alguns casos, será preciso lançar uma exceção fora de um método e, para isso, utiliza-se o comando ***throws***. Além disso, todo código que precisa ser executado ao sair de um bloco try será colocado no bloco ***finally***.

Classes

Algumas classes do Java são exclusivas para tratar certos tipos de exceção.

Confira uma pequena lista com as mais classe mais comuns e um breve exemplo de utilização delas.

1. NullPointerException

Uma exceção do tipo **NullPointerException** é lançada quando um programa Java tenta processar um objeto com um valor do tipo **null**.

Por exemplo:

```
public class Exemplo {
  public void fazAlgo() {
    Integer numero = null;

    if (numero > 0) {
      System.out.println("O numero e positivo");
    }
  }
}
```

Nesse exemplo, por conta de o objeto **número** conter o valor **null**, uma exceção será disparada pela classe **NullPointerException**.

2. ArrayIndexOutOfBoundsException

Uma exceção do tipo **ArrayIndexOutOfBoundsException** ocorre quando um programa Java tenta processar (inserir ou pesquisar) em um vetor fora do seu índice, ou seja, tenta pesquisar ou adicionar um valor em uma posição do vetor que não existe.

Já foi apresentado um exemplo dessa situação neste conteúdo, mas confiram mais um:

```
public class Exemplo {
  public void processArray() {
    List names = new ArrayList <>();
    names.add("João");
    names.add("Maria");
    return names.get(5);
  }
}
```

Nesse exemplo há um **arraylist** que contém duas posições, mas tenta retornar um valor na sua quinta posição, que, neste caso, não existe.

3. IllegalStateException

O **IllegalStateException** é lançado quando um método está sendo chamado em um momento ilegal ou inadequado. Uma ocorrência comum dessa exceção é lançada ao tentar remover um item da lista enquanto você está processando essa lista, como demonstrado a seguir:

```
public class Exemplo {
  public void sendoprocessadoArray() {
    List names = new ArrayList<>();
    names.add("Eric"); //adicionando na lista
    names.add("Sydney");

    Iterator iterator = names.iterator();

    while (iterator.hasNext()) {
      iterator.remove();
    }
  }
}
```

Nesse exemplo, uma exceção será disparada, pois está sendo chamado o método de remover da lista dentro de um *while* de incluir na lista.

4. ClassCastException

A classe **ClassCastException** é lançada quando se tenta incluir um objeto dentro de outro, sendo que eles não pertencem à mesma classe hierárquica. Pode acontecer por exemplo quando se tenta colocar um objeto do tipo *long* dentro de um do tipo *string*.

Observe:

```
public class Exemplo {
  public void forcandoUmCastErrado() {
    Long value = 1967L;
    String name = (String) value;
  }
}
```

5. ArithmeticException

A classe **ArithmeticException** será chamada quando uma condição aritmética não permitida for executada, por exemplo, no caso de uma tentativa de divisão por zero.

Confira:

```
return numero / 0;
```

Nesse exemplo, um retorno em uma variável que tenta ser dividida por zera lançará a exceção.

Essas foram as cinco classes mais comumente utilizadas e disparadas em Java.

Lançando suas próprias exceções

Até agora você viu exceções que foram geradas e captadas (*catch*) pela JVM, no entanto, você pode lançar (*throw*) suas próprias exceções.

Veja como lançar uma **ArithmeticException** manualmente:

```
// Lançando uma exceção manualmente com o throw

class ThrowDemo{
  public static void main(String args[]){
try{
      System.out.println("Antes de lançar a exceção");
      throw new ArithmeticException(); // Lancando a excecao
    }
    catch (ArithmeticException exc){
      //Capturando a exceção
      System.out.println("Antes de lançar a exceção");
    }

    System.out.println("Aqui já estamos fora do bloco try/catch");
    }
}
```

A saída desse programa seria a seguinte:

```
Antes de lançar a exceção

Exceção capturada

Aqui já estamos fora do bloco try/catch

------------------------------------------------------------------------

BUILD SUCCESS

------------------------------------------------------------------------
```

Algumas vezes, pode ser preciso executar uma rotina após uma exceção ser executada, por exemplo, fechar um arquivo, desconectar de um servidor etc. Para esses casos, use o bloco ***finally***.

O bloco ***finally*** será executado sempre que um bloco ***try/catch*** terminar sua execução, tenha ou não ocorrido uma exceção.

Analise o exemplo:

```
//Usando o bloco finally para garantir a execução depois de um try/catch

class UseFinally{
  public static void geraExcecao(int escolha){
    int t;
    int nums[] = new int[2];

    System.out.println("Recebendo" + escolha);

    try{
      switch(escolha){
    case 0:
      t = 10 / escolha; // Gerando o erro de divisão por zero
      break;
    case 1:
      nums[4] = 4; // Gerando o erro de índice fora do escopo
      break;
    case 2:
      return; // Retornando ao bloco try
      }
    }
    catch (ArithmeticException exc){
      //Capturando a exceção
      System.out.println("Divisao por zero detectada");
      return; //Return do catch
    }
    catch (ArrayIndexOutOfBoundsException exc){
      //Capturando a exceção
      System.out.println("Erro de índice fora do escopo");
    }
    finally{
      System.out.println("Saindo do bloco try");
    }

  }
}


class FinallyDemo{
  public static void main(String args[]){
      for (int i=0; i < 3; i++){
    UseFinally.geraExcecao(i);
    System.out.println();
      }
  }
}
```

```
class FinallyDemo{
  public static void main(String args[]){
    for (int i=0; i < 3; i++){
    UseFinally.geraExcecao(i);
    System.out.println();
    }
  }
}
```

A saída desse programa seria o seguinte:

```
Recebendo 0

Divisão por zero detectada



Recebendo 1

Erro de índice fora do escopo



Recebendo 2

Saindo do bloco try

------------------------------------------------------------------------

BUILD SUCCESS

------------------------------------------------------------------------
```

Usando o throws

O **throws** é utilizado sempre que houver a necessidade de, ao executar um método, este (o método) deva tratar essa exceção.

```
public class JavaTester{
  public int division(int a, int b) throws ArithmeticException{
      int t = a/b;
      return t;
  }
  public static void main(String args[]){
      JavaTester obj = new JavaTester();
      try{
        System.out.println(obj.division(15,0));
      }
      catch(ArithmeticException e){
        System.out.println("Voce nao deveria tentar dividir por zero");
      }
  }
}
```

A saída desse programa seria a seguinte:

```
Você não deveria tentar dividir por zero

------------------------------------------------------------------------

BUILD SUCCESS

------------------------------------------------------------------------
```

Encerramento

Neste conteúdo, você pôde aprender que aplicações modernas e sistemas de informação são complicadas estruturas de dados, e que, para contornar e evitar problemas de execução em seu *software*, o tratamento de exceções é

uma prática indispensável.

Você aprendeu que, tratando esses dados, é possível contornar problemas tanto previsíveis quanto imprevisíveis, acelerar a solução do problema, descobrir a origem deste e, não menos importante, salvar evidências de que o problema ocorreu.

Você também viu que problemas de *performance* podem ocorrer devido a falhas não tratadas, além disso, quando essas exceções são devidamente tratadas, garante-se um *software* mais seguro.

Quando você tratar corretamente esses problemas com as ferramentas certas, garantirá a execução desejada e o fluxo correto do seu *software*.

Em suma, exceções fornecem os meios para separar os detalhes do que fazer quando algo fora do comum acontece a partir da lógica principal de um programa.