DESCRIÇÃO

Utilização de tecnologia Java para a construção de sistemas interativos baseados em interfaces gráficas por meio de bibliotecas, como, por exemplo, AWT, Swing, SWT e JavaFX, mediante a adoção de padrões de desenvolvimento e de arquitetura MVC para os sistemas JavaFX.

PROPÓSITO

Utilizar as bibliotecas de componentes gráficos do Java, como AWT, Swing, SWT e JavaFX, para a construção de sistemas baseados em interfaces gráficas com uma interatividade baseada no uso de eventos, além da arquitetura MVC e de alguns padrões de desenvolvimento nos sistemas JavaFX para o alinhamento do processo de construção voltado para as exigências do atual mercado de desenvolvimento.

PREPARAÇÃO

Antes de iniciar a leitura deste conteúdo, é necessário configurar o ambiente com a instalação do JDK e do Apache NetBeans, definindo a plataforma de desenvolvimento a ser utilizada na codificação e na execução dos exemplos práticos. Também é necessário baixar a biblioteca SWT e instalar o editor JavaFX Scene Builder segundo procedimentos descritos ao longo do texto.

OBJETIVOS

MÓDULO 1

Descrever a construção de interfaces gráficas com base em AWT e Swing.

MÓDULO 2

Descrever a construção de interfaces gráficas com base em SWT e JavaFX.

MÓDULO 3

Aplicar a interatividade em sistemas gráficos por meio de eventos.

MÓDULO 4

Aplicar a arquitetura MVC e os padrões de desenvolvimento nos sistemas JavaFX.

INTRODUÇÃO

Neste conteúdo, analisaremos a construção de interfaces gráficas por meio de bibliotecas oferecidas pelo ambiente Java em um processo que inclui a composição das telas e a definição da interatividade com base em eventos.

Após compreendermos a utilização das principais bibliotecas disponíveis, construiremos um sistema baseado na tecnologia JavaFX. Para isso, seguiremos a arquitetura MVC, além de alguns padrões de desenvolvimento úteis para a arquitetura.

MÓDULO 1

- **Descrever a construção de interfaces gráficas com base em AWT e Swing.**

GUI (GRAPHICAL USER INTERFACE)

Os primeiros computadores pessoais trabalhavam no modo texto com comandos simples de entrada e saída para a interação com vídeo e teclado. Até mesmo os jogos eram feitos com caracteres ASCII, mas os pesquisadores da **Xerox**, em uma atitude pioneira, definiram conceitos que persistem até os dias atuais na criação de interfaces gráficas.

★ EXEMPLO

Janelas, menus, caixas de opção, caixas de seleção e ícones.

Seguindo as ideias definidas pela Xerox, a **Apple** lança o Macintosh em 1984, primeiro produto de sucesso a utilizar uma interface gráfica, enquanto a **Microsoft** contava com interfaces gráficas inicialmente invocadas a partir do MS-DOS. Recebendo o nome de Windows, elas só se popularizaram a partir de 1990 com a versão 3.0. Diversas outras plataformas também passaram a disponibilizar ambientes baseados em janelas.

★ EXEMPLO

XWindow, no UNIX; Linux; Workbench, no Commodore Amiga; e, por fim, GEM, nos computadores Atari, apenas para citar alguns exemplos que se tornaram populares.

Essas diversas plataformas apresentam um conjunto-padrão de componentes visuais denominado **WIMP** (Window, Icon, Menu, Pointer), ou seja, um ambiente composto de janelas, ícones, menus e ponteiros. Em ambientes WIMP, os gerenciadores de janelas facilitam a interação entre as representações gráficas dos componentes, as aplicações e o sistema de

gerenciamento de hardware, proporcionando um ambiente que passou a ser conhecido como **desktop**.

Os componentes definidos para o ambiente desktop também foram adotados para as interfaces gráficas dos **dispositivos móveis**. Apesar disso, deve-se guardar as devidas proporções, já que as telas de toque modificam a interatividade ao excluir a presença de um ponteiro, enquanto as janelas não podem ser exibidas simultaneamente.

BIBLIOTECAS AWT E SWING

O ambiente Java oferece, por padrão, duas bibliotecas voltadas para a construção de sistemas baseados em interface gráfica:

AWT (Abstract Window Toolkit), que trata da API original para GUI.

Swing, parte da JFC (Java Foundation Classes).

Nas bibliotecas Swing e AWT, observa-se a presença de dois grupos de componentes:

Componentes visuais: Voltados para a interação com o usuário, como botões, campos de texto, caixas de seleção e listas.

Contêineres: Responsáveis por agrupar os visuais.

O principal contêiner é a janela, definida a partir de **Frame**, **Dialog** ou **Applet**, no AWT, sobre o qual posicionamos os componentes visuais, incluindo os painéis definidos pela classe **Panel** (tipo de contêiner capaz de agrupar componentes visuais e outros painéis de forma recursiva).

Nós utilizaremos neste conteúdo a **biblioteca Swing**, pois o aspecto dos componentes visuais dessa biblioteca independe do ambiente de execução, enquanto a AWT sempre adota o padrão utilizado pela plataforma ou pelo sistema operacional. Em termos práticos, a programação é muito similar, mas os nomes de classes são precedidos da letra "**J**".

★ EXEMPLO

JFrame, JDialog, JPanel ou JButton.

Antes de começarmos a programar, precisamos compreender ainda o papel de mais um dos componentes estruturais: o **layout**. Ele serve para organizar a forma como os componentes visuais serão dispostos sobre o contêiner.

Esta tabela apresenta os principais descendentes de layout e demonstra como eles definem o posicionamento:

Layout	Funcionalidade
BorderLayout	Aceita até cinco componentes visuais, que devem ser posicionados em **North**, **South**, **East**, **West** e **Center**.
BoxLayout	Dispõe os componentes visuais em uma linha ou coluna simples, podendo ser alinhados.
CardLayout	Controlado por uma lista do tipo **drop-down**, permite exibir painéis de componentes de forma alternada.
FlowLayout	Apenas coloca os componentes sequencialmente em uma linha, mas serve como base para os painéis de **CardLayout**.
GridBagLayout	Extremamente sofisticado, divide o layout em um grid e, em seguida, permite a mescla de células e a adição de espaços ao posicionar os componentes.
GridLayout	Trabalha com um grid simples, em que os componentes vão preenchendo sequencialmente os espaços definidos.

Atenção! Para visualização completa da tabela utilize a rolagem horizontal

Quadro: Tipos de layout para Swing e AWT.

Elaborado por Denis Gonçalves Cople

A melhor forma de compreender a utilização da biblioteca Swing é por intermédio de um exemplo prático. Observemos o processo na listagem apresentada a seguir:

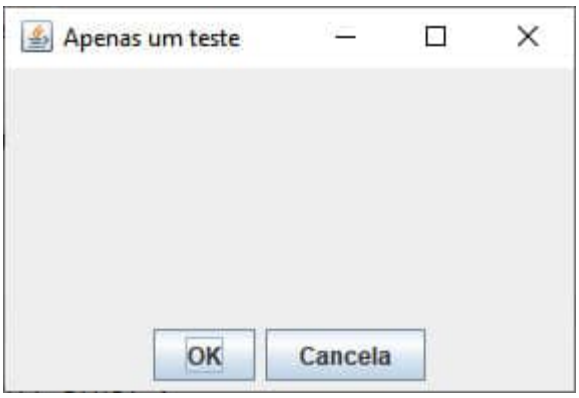
```
import java.awt.*;
import javax.Swing.*;

public class MinhaJanela extends JFrame {
	private final JButton b1 = new JButton("OK");
	private final JButton b2 = new JButton("Cancela");

	public MinhaJanela() throws HeadlessException {
		super("Apenas um teste");
		setLayout(new BorderLayout());
		setBounds(10, 10, 300, 200);
		JPanel jp = new JPanel(new FlowLayout());
		jp.add(b1);
		jp.add(b2);
		add(jp,"South");
	}

	public static void main(String[] args) {
		new MinhaJanela().setVisible(true);
	}
}
```

Nosso exemplo contém uma classe descendente de **JFrame**, a qual, na prática, constitui uma janela com dois atributos do tipo **JButton** e os textos "OK" e "Cancela".

No nível do construtor, temos a definição do título de nossa janela, com a chamada ao construtor do pai, e efetuamos algumas configurações, como o uso de **setLayout** para a escolha do tipo **BorderLayout** e a chamada para **setBounds**, definindo as coordenadas X e Y seguidas da largura e da altura da janela.

Ainda no construtor, podemos observar a criação de um **JPanel** com base em **FlowLayout** no qual são adicionados os botões, sendo adicionado o painel na posição inferior da janela

(**South**) ao final. Com a janela finalizada, basta instanciá-la e deixá-la visível no nível do método **main** para obter este resultado durante a execução:



📷 Execução do exemplo inicial de Swing.
Captura de tela da interface gráfica gerada com Java e bibliotecas Swing e AWT no NetBeans.

Embora tenhamos um exemplo simples, ele demonstra toda a dinâmica para a construção de janelas por meio das bibliotecas AWT e Swing. Mesmo trabalhando com janelas mais complexas, as diferenças estão apenas na quantidade de componentes, na composição de painéis e no uso de layout ou no tipo de janela de base, como a **JDialog**, para se trabalhar no formato **modal**, em que o foco é detido na janela.

💬 COMENTÁRIO

As bibliotecas AWT e Swing são bibliotecas gráficas nativas do Java.

COMPONENTES VISUAIS PARA AWT E SWING

A padronização de componentes nas diversas plataformas trouxe grandes vantagens para os usuários em termos de aprendizagem. Como as interfaces gráficas apresentam os mesmos conceitos visuais, a utilização de um novo sistema se torna intuitiva.

Podemos, no quadro seguinte, observar os componentes visuais mais comuns e as classes que os representam nas bibliotecas AWT e Swing:

Componente	AWT	Swing	Funcionalidade
Botão	Button	JButton	Interação com o usuário baseada no clique do mouse.
Caixa de texto	TextField TextArea	JTextField JTextArea	Digitação de valores no formato texto pelo usuário.
Elementos selecionáveis	CheckBox	JCheckBox JRadioButton	Opções que podem ser marcadas ou desmarcadas por meio do clique.
Listas e tabelas	List	JList JComboBox JTable	Componentes multivalorados que representam conjuntos de dados de um mesmo tipo.
Menus	Menu MeuItem	JMenu JMenuItem	Listas de ações agrupadas, acionadas por meio de clique, que normalmente ficam na parte superior da janela.
Etiquetas	Label	JLabel	Apenas um texto explicativo estático.

Atenção! Para visualização completa da tabela utilize a rolagem horizontal

Quadro: Componentes visuais com AWT e Swing.

Elaborado por Denis Gonçalves Cople

Como exemplo, se quisermos construir uma janela para o cadastro de **produtos** com base nos atributos **código**, **nome** e **quantidade**, poderemos trabalhar com caixas de texto para a edição dos valores e botões para inserir ou remover o registro na base de dados. Além disso, uma **tabela** pode ser utilizada para a exibição da listagem dos produtos, pois, embora ela não seja um componente padrão para GUI, atua da mesma forma que as **listas**, com a diferença da divisão em colunas, o que visualmente lembra uma planilha.

O código para a construção da janela pode ser observado a seguir. Como o NetBeans complementa o código por meio da combinação de teclas **CTRL + espaço**, adicionando as importações de forma automática, ambas foram omitidas.

```
public class JanelaProduto extends JDialog{
    private JTextField txtCodigo, txtNome, txtQuantidade;
    private JTable tblProdutos;
    private JButton btnIncluir, btnExcluir;

    public JanelaProduto() {
        JPanel edicao = new JPanel(new GridLayout(4, 2));
        edicao.add(new JLabel("Codigo"));
        edicao.add(txtCodigo = new JTextField(20));
        edicao.add(new JLabel("Nome"));
        edicao.add(txtNome = new JTextField(20));
        edicao.add(new JLabel("Quantidade"));
        edicao.add(txtQuantidade = new JTextField(20));
        edicao.add(btnIncluir = new JButton("Incluir"));
        edicao.add(btnExcluir = new JButton("Excluir"));

        JScrollPane listagem = new javax.Swing.JScrollPane();
        listagem.add(tblProdutos = new JTable(
        new Object [][] { {null, null, null}},
        new String [] { "Codigo", "Nome", "Quantidade" }));
        listagem.setViewportView(tblProdutos);

        this.setLayout(new GridLayout(2, 1));
        this.add(edicao);
        this.add(listagem);
    }
```

```
    public static void main(String[] args) {
        JanelaProduto j1 = new JanelaProduto();
        j1.setModal(true);
        j1.setBounds(0,0, 300, 300);
        j1.setVisible(true);
    }
    }
```

Para o exemplo, utilizamos **JDialog**, que pode ser apresentado de forma **modal**. Isso significa dizer que a janela domina o foco corrente, impedindo que o usuário troque para outra do mesmo programa.

Após a definição da classe, com sua descendência, temos a inclusão dos atributos dos tipos:

JTextField para as caixas de texto.
JButton para os botões de inclusão e exclusão.
JTable para exibição da lista de registros.

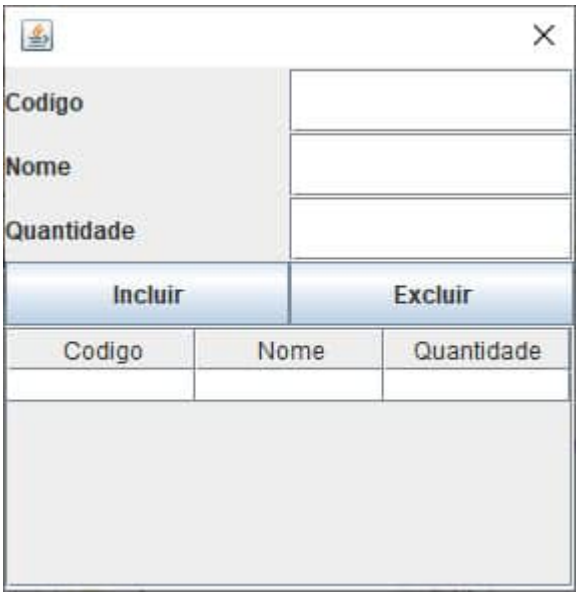
Atenção! Para visualização completa da tabela utilize a rolagem horizontal

No código interno do construtor, é possível observar um **JPanel** configurado a partir de um **GridLayout** com quatro linhas e duas colunas. Ao painel são adicionados, de forma sequencial, os componentes **JLabel** (com a denominação de cada campo) e **JTextField** para a entrada de dados, além dos botões de inclusão e exclusão.

Em seguida, criamos um **JScrollPane**, um tipo de contêiner que possibilita a rolagem de tela, com o nome listagem, posicionando o **JTable** sobre ele. Quanto à construção do JTable, temos uma matriz de objetos para os valores – nesse caso, com apenas uma linha de valores nulos – e um vetor de texto com os títulos das colunas.

Ao final do construtor, configuramos a disposição de componentes na área da janela por meio de um **GridLayout** com duas linhas e uma coluna, sendo adicionados o JPanel e o JScrollPane criados nos passos anteriores.

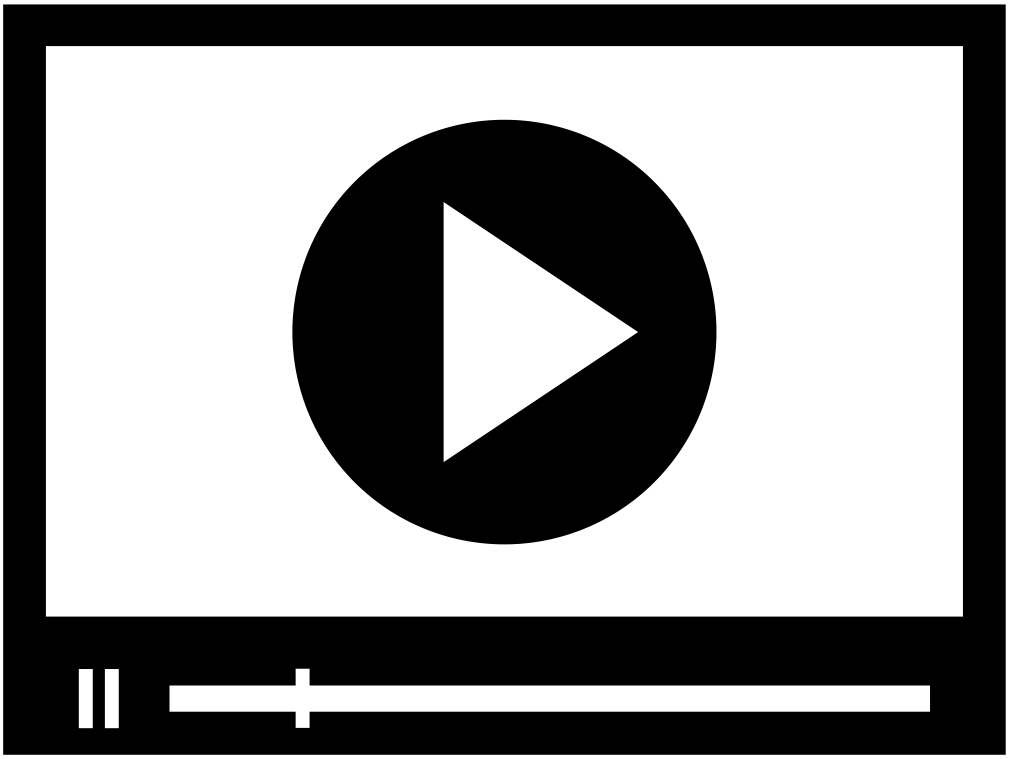
Quanto ao método **main**, temos apenas a criação de uma instância de **JanelaProduto**, nosso descendente de JDialog. Configurado para travar o foco por meio de **setModal** com valor **true**, ele é posicionado no canto superior esquerdo do monitor, com largura e altura de trezentos pixels, sendo deixado visível ao final.



📷 Execução da classe JanelaProduto.
Captura de tela da interface gráfica gerada com Java e bibliotecas Swing e AWT no NetBeans.

📢 ATENÇÃO

Algo que fica claro na criação de janelas é a utilização simultânea de herança e composição, pois a janela herda de JFrame ou JDialog, sendo composta de botões e outros elementos visuais.



COMPONENTES DO SWING

Assista ao vídeo a seguir. Nele, apresentamos os componentes oferecidos pelo Swing.

DIÁLOGOS PADRONIZADOS

Além das janelas criadas com JFrame e JDialog, é possível utilizar alguns diálogos padronizados com **JOptionPane**. Trata-se de janelas de diálogo voltadas para ações pontuais.

★ EXEMPLO

Situações em que fazemos uma pergunta ao usuário ou quando desejamos informá-lo acerca de algo ocorrido durante a execução do sistema.

Podemos utilizar JOptionPane até mesmo a partir da linha de comando, cumprindo boa parte das necessidades usuais de entrada de dados e exibição de informações. É possível observar isso pela descrição de seus métodos no quadro seguinte:

Método	Funcionalidade
showConfirmDialog	Diálogo de confirmação no estilo "sim/não/cancela".
showInputDialog	Solicitação de dados ao usuário.
showMessageDialog	Exibe uma mensagem para o usuário.
showOptionDialog	Unificação dos outros três tipos.

Atenção! Para visualização completa da tabela utilize a rolagem horizontal

Quadro: Métodos da classe JOptionPane.
Elaborado por Denis Gonçalves Cople

Observemos um pequeno exemplo de utilização do componente JOptionPane:

```
import javax.Swing.JOptionPane;
public class ExemploOptionPane {
	public static void main(String[] args) {
		int a = new Integer(
			JOptionPane.showInputDialog("Primeiro Numero:"));
		int b = new Integer(
			JOptionPane.showInputDialog("Segundo Numero:"));
		int c = a + b;
		JOptionPane.showMessageDialog(null, "A soma será: "+c,
				"Soma", JOptionPane.INFORMATION_MESSAGE);
```

```
    }
}
```

Em nosso exemplo, temos o uso do método **showInputDialog** para solicitar os valores – que devem ser convertidos por inteiro por terem sido recebidos como texto – ao usuário. Em seguida, efetuamos a soma dos números fornecidos e exibimos o resultado graças ao método **showMessageDialog**.

A execução do programa pode ser observada a seguir, na sequência de figuras geradas com Java e biblioteca Swing no NetBeans:

Execução do exemplo com JOptionPane.
Captura de tela da interface gráfica gerada com Java e biblioteca Swing no NetBeans.

Execução do exemplo com JOptionPane.
Captura de tela da interface gráfica gerada com Java e biblioteca Swing no NetBeans.

Execução do exemplo com JOptionPane.
Captura de tela da interface gráfica gerada com Java e biblioteca Swing no NetBeans.

Note que a opção **JOptionPane.INFORMATION_MESSAGE** fará com que apareça o ícone padrão de informação na caixa de diálogo. Outros ícones, como pode ser observado no quadro seguinte, ainda podem ser utilizados.

Valor do parâmetro	Tipo de ícone
JOptionPane.INFORMATION_MESSAGE	Informação comum.
JOptionPane.ERROR_MESSAGE	Indicativo de ocorrência de erro.
JOptionPane.QUESTION_MESSAGE	Interrogação para efetuar perguntas.
JOptionPane.WARNING_MESSAGE	Sinal de atenção ou alerta.

Atenção! Para visualização completa da tabela utilize a rolagem horizontal

Quadro: Métodos da classe JOptionPane.

Elaborado por Denis Gonçalves Cople

VERIFICANDO O APRENDIZADO

MÓDULO 2

⦿ **Descrever a construção de interfaces gráficas com base em SWT e JavaFX.**

BIBLIOTECA SWT (STANDARD WIDGET TOOLKIT)

A biblioteca AWT foi introduzida pela Sun Microsystems no JDK 1.0 como padrão para a criação de interfaces gráficas. Os componentes Swing foram utilizados na geração seguinte, sendo introduzidos pela Sun na Plataforma J2SE ou JDK 1.2, o que forneceu um conjunto visualmente mais rico de componentes.

Tendo surgido na década de 1990, o SWT foi criado pela Object Technology International (OTI) como um conjunto de widgets nativos multiplataforma voltados para o OTI Smalltalk, que se tornou o **IBM Smalltalk** em 1993. A camada Common Widget do IBM Smalltalk forneceu acesso rápido e nativo a vários componentes visuais dos diversos sistemas operacionais, ao mesmo tempo em que apresentou uma API com total portabilidade, algo que não era comum em outros kits de ferramentas para a construção de GUI.

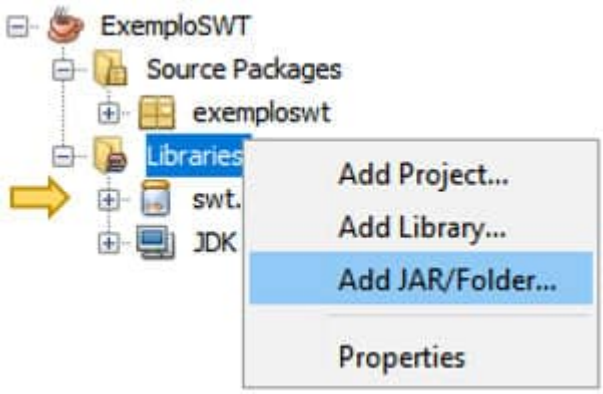
Com o desenvolvimento do **VisualAge**, uma IDE da IBM escrita em **Smalltalk**, ocorre a decisão de abrir o projeto. Isso levou ao surgimento do **Eclipse**, produto destinado a competir com ambientes proprietários, como o Microsoft Visual Studio.

O ambiente do Eclipse é escrito em **Java**. Os desenvolvedores da IBM precisavam de um kit de ferramentas GUI com as mesmas características do produto originalmente definido pela OTI, ou seja, que tivesse a aparência, o comportamento e o desempenho de componentes nativos. Isso levou à criação da biblioteca SWT para o ambiente Java em substituição aos componentes do Swing.

SAIBA MAIS

Embora originalmente fosse um produto diretamente relacionado ao uso do Eclipse, essa biblioteca foi separada das demais e disponibilizada como um arquivo jar ou zip1 disponível na seção de releases do endereço da Eclipse. Ali pode ser encontrado um link para a página de downloads, enquanto o zip específico para o sistema operacional está presente na seção SWT Binary and Source da versão escolhida.

Após baixar o arquivo da versão estável mais recente, precisamos descompactá-lo e encontrar o arquivo **swt.jar**, adicionando, em seguida, a biblioteca ao projeto por meio do clique com o botão direito em **Libraries** e a escolha da opção **Add JAR/Folder**. Isso pode ser verificado na figura ao lado:



Adição da biblioteca SWT ao projeto.
Captura de tela do NetBeans.

Agora o nosso projeto, que deverá ser do tipo Java padrão, pode utilizar os componentes do SWT diretamente. Basta, para tal, fazer as chamadas adequadas no método **main**, como será exemplificado no decorrer deste tópico.

A estrutura do SWT envolve **quatro** componentes principais:

Display	Representa a estação de trabalho, com monitor, teclado e mouse, sendo responsável pelo disparo de eventos.
Shell	Define um componente de janela, sendo a base de toda a hierarquia visual e dos contêineres utilizados.
Composite	Responsável pela organização da tela, ele é semelhante a um painel, sendo um contêiner para componentes visuais e outros contêineres.
Control	Encapsula um componente visual do sistema operacional, como Button, Label, Text ou Tree.

Atenção! Para visualização completa da tabela utilize a rolagem horizontal

Elementos como Shell, Composite e Control são descendentes de **Widget**, sendo uma nomenclatura comumente adotada para eles. No uso do SWT, temos um loop explícito de eventos que deve ser implementado na classe principal do aplicativo. Ele deve ler e despachar

todos os eventos de interface gráfica recebidos a partir do sistema operacional até que a aplicação seja finalizada – normalmente, com o fechamento da janela principal.

```
while (!shell.isDisposed()) {
	if (!display.readAndDispatch())
		display.sleep();
}
```

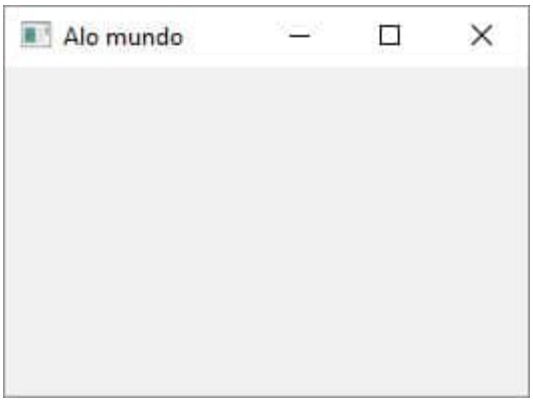
A criação de uma janela vazia, como pode ser observado no código de exemplo apresentado a seguir, é muito simples:

```
import org.eclipse.swt.widgets.Display;
import org.eclipse.swt.widgets.Shell;

public class JanelaTesteSWT {
	public static void main(String[] args) {
	Display = new Display();
	Shell shell = new Shell(display);
	shell.setText("Alo mundo");
	shell.open();
	while(!shell.isDisposed())
		if(!display.readAndDispatch())
		display.sleep();
	display.dispose();
	}
}
```

Inicialmente, é instanciado um objeto da classe **Display** para a interação com o ambiente e, em seguida, um objeto **Shell**, que está relacionado ao primeiro objeto e representa a janela em si. Nas linhas seguintes, podemos observar o uso de **setText** para a definição do título da janela e a abertura dela com **open**.

Na sequência, o programa ficará preso ao loop de eventos até o momento em que a janela for fechada, passando, assim, para a linha seguinte. Nela, o objeto display utiliza **dispose** para liberar quaisquer recursos pendentes relacionados ao sistema operacional.

Como resultado da execução, ocorre a apresentação da janela observada ao lado:



📷 Janela vazia com SWT.

Captura de tela da interface gráfica gerada com Java e biblioteca SWT no NetBeans.

A classe **SWT** contém todas as constantes da biblioteca, como **SWT.PUSH** e **SWT.RADIO**, além de alguns métodos de interesse, incluindo **getPlatform** e **getVersion**. Ela é muito importante para a definição de características gerais.

⭐ EXEMPLO

Tipo de botão e borda.

O construtor de um **Widget** normalmente envolve a indicação do contêiner, que pode ser um **Shell** ou um **Composite**, e um parâmetro de estilo.

💡 DICA

O componente do tipo Shell sempre está associado ao Display ou a outro Shell, ou seja, a janela está relacionada ao ambiente, no primeiro nível, ou a outra janela, em níveis subsequentes.

Analisemos agora uma janela com alguns componentes visuais:

```
public class JanelaCalculadoraSWT {
    public static void main(String[] args) {
        Display = new Display();
        Shell = new Shell(display);
        shell.setText("Calculadora");
```

```
        shell.setSize(250,80);
        Text t1 = new Text(shell, SWT.BORDER);
        t1.setBounds(10, 10, 50, 25);
        Label l1 = new Label(shell, SWT.NONE);
        l1.setText("+");
        l1.setBounds(65,15,10,15);
        Text t2 = new Text(shell, SWT.BORDER);
        t2.setBounds(80, 10, 50, 25);
        Button b1 = new Button(shell, SWT.PUSH);
        b1.setText("=");
        b1.setBounds(145,12,20,23);
        shell.open();
        while(!shell.isDisposed())
            if(!display.readAndDispatch())
                display.sleep();
        display.dispose();
    }
}
```

No exemplo anterior, vemos a utilização dos componentes Text, Label e Button. Para qualquer um deles, o posicionamento direto sobre a janela (ou Shell) precisa da definição dos limites (bounds) por intermédio de **setBounds**. Ele tem como parâmetros, respectivamente, as coordenadas **x** e **y**, **largura** e **altura**.

A definição do texto interno foi efetuada com o uso de **setText** para os componentes Label e Button. Já a janela teve seu tamanho determinado com o uso do método **setSize**, recebendo a largura e a altura como parâmetros.

Os estilos também foram utilizados na definição dos componentes, como o uso de **SWT.BORDER**, para a criação do contorno das caixas de texto, além da definição do tipo de botão, como o **SWT.PUSH**.

★ EXEMPLO

A utilização dos botões é muito interessante, pois o estilo SWT.RADIO transformaria o botão tradicional em um de rádio, modificando completamente, dessa forma, seu aspecto e

funcionalidade.

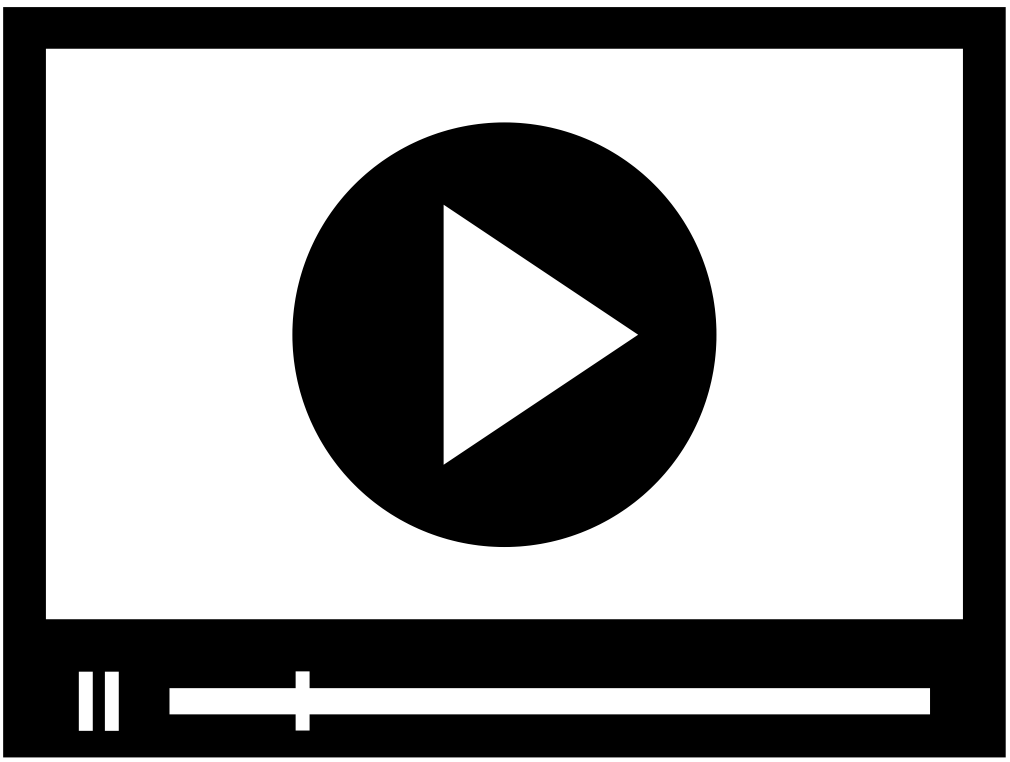
O resultado da execução pode ser observado na figura seguinte:

📷 Componentes visuais em janela SWT.

Captura de tela da interface gráfica gerada com Java e biblioteca SWT no NetBeans.

📢 ATENÇÃO

A arquitetura do SWT envolve a utilização de JNI (Java Native Interface) para efetuar chamadas à API de componentes gráficos do sistema operacional em que normalmente são utilizadas as linguagens C e C++ na implementação. Por conta disso, ocorre uma dependência em relação ao sistema operacional sobre o qual a JVM executa.



COMPONENTES DO SWT

No vídeo a seguir, abordamos os componentes visuais oferecidos pelo SWT.

COMPOSITE E LAYOUT

Embora seja possível criar uma janela simples com o uso de elementos posicionais, isso certamente não é o melhor procedimento, principalmente se considerarmos a grande diversidade de resoluções de tela dos dispositivos atuais. Assim como era feito no Swing, podemos organizar os contêineres visuais do SWT com o uso de objetos de layout.

Observe alguns exemplos desses objetivos no quadro a seguir:

Layout	Funcionalidade
FillLayout	Linha ou coluna com o preenchimento de todo o espaço.
RowLayout	Linhas ou colunas com um preenchimento simples, de forma sequencial.
GridLayout	Tabela organizada em linhas e colunas, o que permite a combinação de células por agrupamento.
FormLayout	Usado para a criação de formulários, o posicionamento é relativo, conectando as extremidades dos widgets.

StackLayout	Elementos empilhados que podem ser utilizados com galerias de fotos e pastas que se alternam.

Atenção! Para visualização completa da tabela utilize a rolagem horizontal

Quadro: Tipos de layout para SWT.
Elaborado por Denis Gonçalves Cople

O uso de um layout permite o controle da posição e do tamanho dos componentes dispostos sobre um **Composite**. Muitos dos layouts trazem os objetos de dados voltados para configurações específicas, os quais, a partir dos componentes, são usados por meio do método **setLayoutData**.

Com o uso de elementos do tipo Composite é possível combinar diversos layouts, produzindo janelas mais complexas, assim como era feito na utilização de componentes Panel nas janelas criadas com Swing.

Por intermédio de uma janela com dois botões e uma caixa de texto, exemplificaremos o uso de layouts como um diálogo de entrada de dados tradicional:

```
public class JanelaTesteLayout {
      public static void main(String[] args) {
       Display display = new Display();
       Shell shell = new Shell(display);
       shell.setText("Identificação");
       shell.setLayout(new FillLayout(SWT.VERTICAL));
       Composite superior = new Composite(shell,SWT.NONE);
       superior.setLayout(new FillLayout(SWT.HORIZONTAL));
       Label l1 = new Label(superior,SWT.NONE);
       l1.setText("Digite seu nome:");
       Text t1 = new Text(superior,SWT.SINGLE|SWT.BORDER);
       Composite inferior = new Composite(shell,SWT.NONE);
       inferior.setLayout(new FillLayout(SWT.HORIZONTAL));
       Button b1 = new Button(inferior, SWT.PUSH);
       b1.setText("Confirmar");
       Button b2 = new Button(inferior, SWT.PUSH);
       b2.setText("Cancelar");
       shell.setSize(300,100);
```

```
        shell.open();
        while(!shell.isDisposed())
          if(!display.readAndDispatch())
              display.sleep();
        display.dispose();
       }
 }
```

Os componentes são adicionados sequencialmente. O primeiro elemento é um Composite com o nome **superior**. São adicionados a ele um **Label** e um **Text**.

Em seguida, há a adição do Composite de nome **inferior** à janela, enquanto são adicionados a ele os dois componentes do tipo **Button**. Como inferior e superior adotam **FillLayout** no modo **SWT.HORIZONTAL**, os componentes são organizados horizontalmente, enquanto a janela adota o **FillLayout** no modo **SWT.VERTICAL**.

Podemos observar também a utilização de **ou binário** na configuração do **Text**. Trata-se da forma correta de combinar mais de um estilo para o **Widget**, pois as constantes utilizam sequências de bits que podem ser superpostas.

📷 Janela composta criada com SWT.
Captura de tela da interface gráfica gerada com Java e biblioteca SWT no NetBeans.

BIBLIOTECA JAVAFX

A tecnologia **JavaFX** foi desenvolvida pela Sun Microsystems com o objetivo de prover uma plataforma para desenvolvimento de aplicativos ricos para internet (**RIA**), sendo posteriormente expandida pela Oracle para as plataformas desktop e móveis.

O produto foi anunciado em maio de 2007 como JavaOne, tendo sua primeira versão lançada em dezembro de 2008. Embora a versão inicial utilizasse a linguagem JavaFX Script, a partir da segunda versão a tecnologia passou a ser uma biblioteca do Java. Nas versões atuais, por sua vez, ela se tornou parte da distribuição padrão do JDK.

As plataformas ARM ganharam suporte nas versões atuais, permitindo sua utilização em dispositivos móveis e sistemas embarcados, tornando-o, assim, uma boa solução para a criação de interfaces em ambientes de **IoT** (sigla de *internet of things*). Talvez o ponto mais interessante das bibliotecas JavaFX seja justamente sua capacidade de lidar com múltiplas plataformas, seguindo, para tal, o mesmo modelo de desenvolvimento.

A tecnologia JavaFX oferece amplo suporte a elementos multimídia, como sons, vídeos e gráficos 2D e 3D, além de permitir a utilização de componentes nativos, trazendo muito poder e versatilidade na construção de interfaces.

Atualmente, existem **duas possibilidades** para o desenvolvimento de GUI com o uso de JavaFX:

Modo programado

Simples utilização do Java.

[redacted]

[redacted]

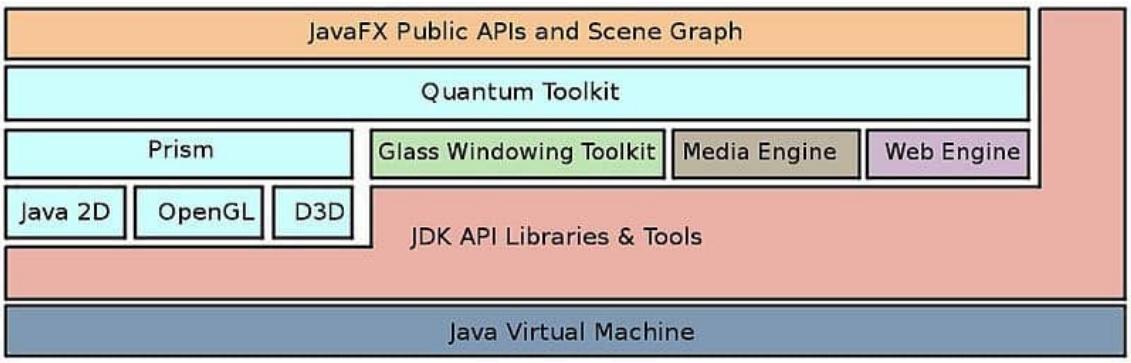
FXML

Linguagem baseada na sintaxe XML, ela é utilizada na estruturação visual dos componentes, o que permite a estilização com **CSS**.

Na arquitetura de componentes do JavaFX, temos elementos para a utilização de Java 2D, Open GL e D3D. Eles são agrupados sob o Prism para o controle das operações de desenho,

enquanto o Glass Windows Toolkit gerencia eventos e chamadas específicas feitas ao sistema operacional.

Os componentes são integrados no nível do Quantum Toolkit, o qual, por sua vez, é acionado a partir de chamadas efetuadas pelas APIs públicas do JavaFX e pelo Scene Graph, que representa as telas em termos de grafos. Também existem motores específicos para a utilização de elementos multimídia e componentes web.



Arquitetura do JavaFX.

Para a criação de um sistema JavaFX no modo programado, é necessário criar um descendente da classe **Application** e substituir o método **start**, que tem como parâmetro o palco principal (**Stage**) ao qual serão adicionadas as cenas, cada uma com os componentes visuais desejados. Ao trabalhar no ambiente do **NetBeans**, é necessário criar um projeto do tipo **JavaFX** com a opção **JavaFX Application**, na qual é possível definir esse descendente.

Observemos o código de exemplo gerado pelo NetBeans:

```
public class ExemploJavaFX extends Application {
    @Override
    public void start(Stage primaryStage) {
     Button btn = new Button();
     btn.setText("Say 'Hello World'");
     btn.setOnAction(new EventHandler() {
       @Override
       public void handle(ActionEvent event) {
           System.out.println("Hello World!");
       }
     });
     StackPane root = new StackPane();
     root.getChildren().add(btn);
     Scene scene = new Scene(root, 300, 250);
```

```
        primaryStage.setTitle("Hello World!");
        primaryStage.setScene(scene);
        primaryStage.show();
       }
       public static void main(String[] args) {
        launch(args);
       }
}
```

No exemplo, temos a definição de um botão e da ação de clique sobre ele em que será impressa a mensagem "Hello World!" no prompt. Para a programação da resposta, é gerado um **EventHandler** para **ActionEvent** e implementado o método **handle**.

💬 COMENTÁRIO

No entanto, mais detalhes acerca de tais eventos serão discutidos apenas no próximo módulo.

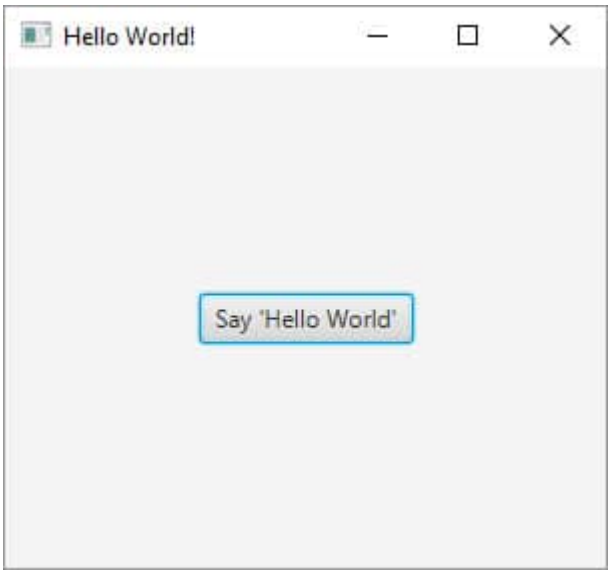
Em seguida, é definido um painel do tipo **StackPane**, ao qual é adicionado o botão com o uso de **add** a partir de **getChildren**. Na sequência, é criada uma cena (**Scene**) com o uso do painel com largura de 300 e altura de 250 pixels. Finalmente temos a configuração do palco, que representa a janela, com a definição do título, a associação com a cena por meio de **setScene** e a exibição com o uso de **show**.

Já o método **main** faz apenas uma chamada a **launch**, método estático de **Application** que já instancia o descendente, e chama **start**.

📢 ATENÇÃO

Cuidado com as importações ao trabalhar com o JavaFX, pois existem classes que apresentam o mesmo nome no AWT. Todas as importações devem ocorrer a partir do pacote javafx e de derivados.

A janela criada no exemplo acima pode ser observada a seguir:

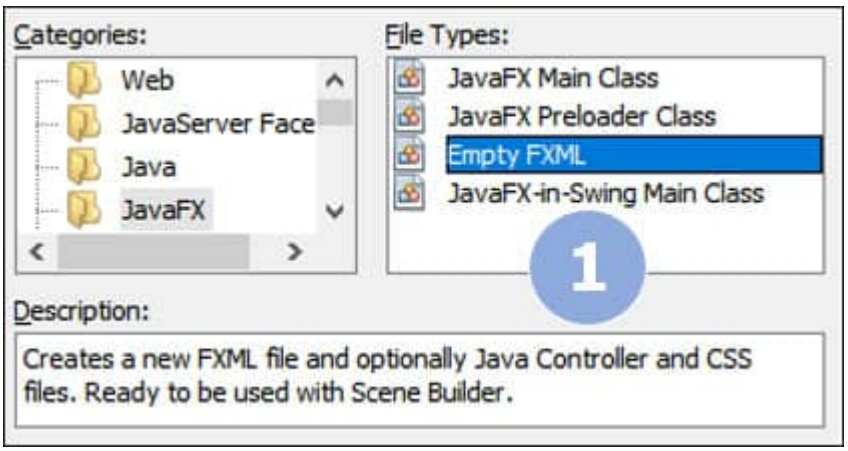


Janela gerada pelo código de exemplo para JavaFX.

Captura de tela da interface gráfica gerada com Java e JavaFX no NetBeans.

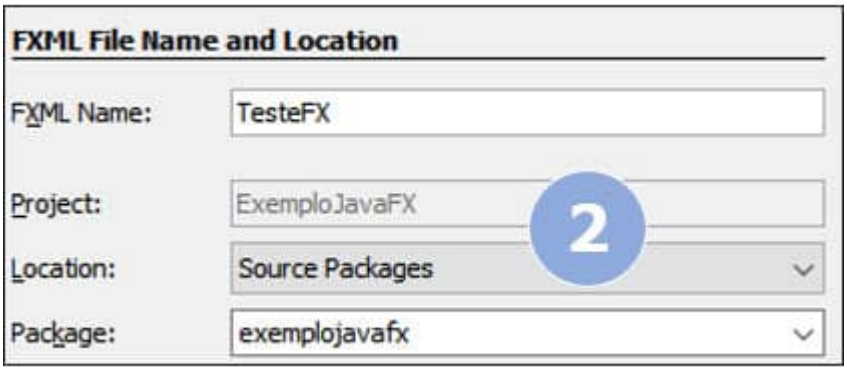
CRIAÇÃO COM FXML

O uso de FXML envolve a criação do arquivo **FXML** propriamente dito, um controlador **Java** e um arquivo de estilização **CSS** (Cascade Style Sheet), algo que é feito de forma simples com o uso do NetBeans, por meio da opção **New File**. Em seguida, ocorre a escolha dos itens JavaFX e Empty FXML, como pode ser observado na sequência de telas (capturas de tela do NetBeans) seguinte:



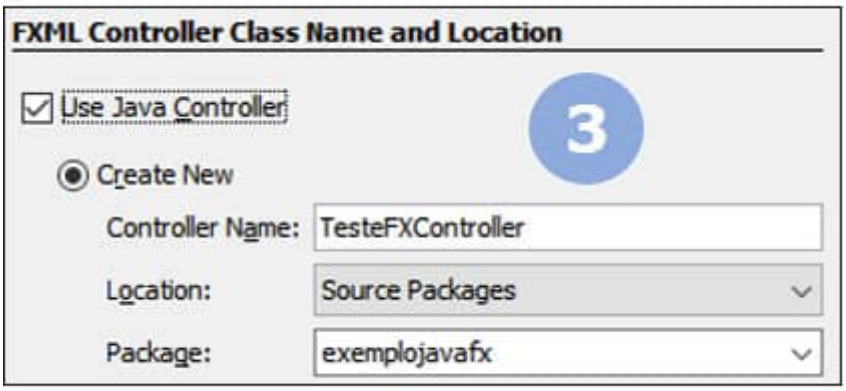
Criação de arquivo FXML vazio no NetBeans.

Captura de tela do NetBeans.



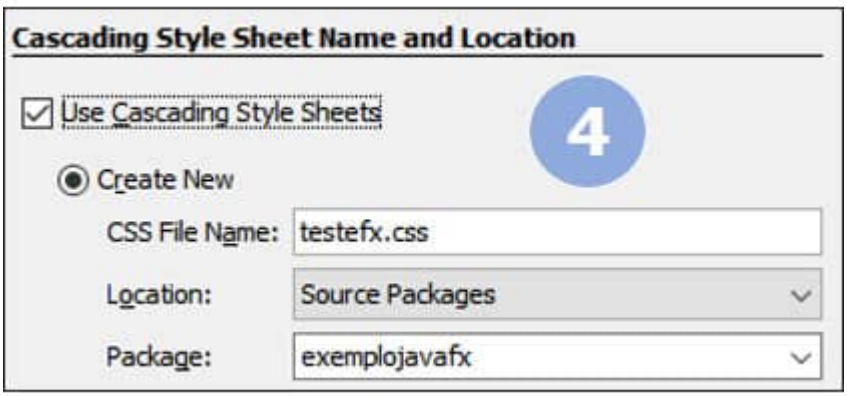
📷 Criação de arquivo FXML vazio no NetBeans.

Captura de tela do NetBeans.



📷 Criação de arquivo FXML vazio no NetBeans.

Captura de tela do NetBeans.



📷 Criação de arquivo FXML vazio no NetBeans.

Captura de tela do NetBeans.

Ao final, teremos três novos arquivos em nosso projeto e poderemos começar a analisar e modificar o arquivo **FXML** para a definição da interface visual:

```
<?xml version="1.0" encoding="UTF-8"?>
<?import java.lang.*?>
<?import java.net.*?>
<?import java.util.*?>
<?import javafx.scene.*?>
<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>
<AnchorPane id="AnchorPane" prefHeight="400.0" prefWidth="600.0"
      styleClass="mainFxmlClass" xmlns:fx="http://javafx.com/fxml/1"
      fx:controller="unesa.TesteFXController">
      <stylesheets> <URL value="@testefx.css"/> </stylesheets>
</AnchorPane>
```

No arquivo gerado, podemos observar várias importações de bibliotecas feitas com o uso da diretiva **import**. Uma importação que se mostra particularmente importante é a de controles do **JavaFX**, na qual estão os botões e demais componentes visuais.

A raiz da hierarquia visual é o **AnchorPane**, no qual são definidos namespace **fx**, altura e largura preferenciais (**prefHeight** e **prefWidth**), além da associação com a classe do controlador (**fx:controller**). O elemento AnchorPane é o painel principal ao qual serão adicionados diversos componentes de forma hierárquica, além da associação com a folha de estilo por meio da tag **stylesheets**.

Modifiquemos o interior de AnchorPane pela definição de nossa janela:

```
<AnchorPane id="AnchorPane" prefHeight="400.0" prefWidth="600.0"
      styleClass="mainFxmlClass" xmlns:fx="http://javafx.com/fxml/1"
      fx:controller="unesa.TesteFXController">
      <stylesheets>
      <URL value="@testefx.css"/>
      </stylesheets>
      <children>
      <TextField layoutX="15.0" layoutY="15.0" prefWidth="200"
            fx:id="txtNome"/>
          <Button layoutX="220" layoutY="15.0" text="Clique Aqui"/>
          <Label fx:id="lblMensagem" layoutX="15.0" layoutY="45"
            prefHeight="25" prefWidth="260" text="Mensagem"/>
```

```
    </children>
</AnchorPane>
```

Foi acrescentada a tag **children**, na qual serão agrupados os componentes; dentro dela, as tags de componentes do tipo **TextField**, **Button** e **Label**. Para os três componentes utilizados, temos os atributos de posicionamento **layoutX** e **layoutY**, além de largura e altura preferencias em alguns casos, ficando a unidade de medida em **pixels**.

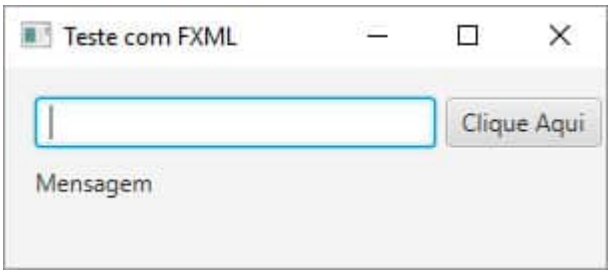
As tags **TextField** e **Label** apresentam o atributo **fx:id**, definindo o identificador que será utilizado pela classe controladora com o objetivo de localizar o componente na tela e associando – segundo o procedimento a ser detalhado nos módulos subsequentes deste conteúdo – ao código Java por intermédio de anotações. Também utilizamos o atributo **text** nas tags **Button** e **Label**, definindo os textos que serão exibidos.

Agora acrescentemos uma classe de **aplicativo** JavaFX por meio da opção **New File**, seguido de JavaFX na opção **JavaFX Main Class**, em que é adotado o nome **ExemploJavaFX2**:

```
public class ExemploJavaFX2 extends Application {
    @Override
    public void start(Stage primaryStage) throws IOException {
     URL arquivoFXML = getClass().getResource("./TesteFX.fxml");
     Parent fxmlParent = (Parent) FXMLLoader.load(arquivoFXML);
     primaryStage.setScene(new Scene(fxmlParent, 300, 100));
     primaryStage.setTitle("Teste com FXML");
     primaryStage.show();
    }
    public static void main(String[] args) {
     launch(args);
    }
}
```

Para a utilização do arquivo FXML, é necessário gerar uma **URL** com o caminho para encontrá-lo, seguido da carga dele por meio de **FXMLLoader**. Ao final, devemos repetir os passos utilizados anteriormente com a utilização do objeto associado ao **FXML** como painel de base para a **cena**.

Após mandar construir o projeto, por meio da opção **build**, precisamos apenas executar a classe de aplicativo. Ela pode ser observada na janela ao lado:



📷 Execução da janela baseada em FXML.

Captura de tela da interface gráfica gerada com Java, JavaFX e FXML no NetBeans.

VERIFICANDO O APRENDIZADO

MÓDULO 3

⦿ **Aplicar a interatividade em sistemas gráficos por meio de eventos.**

INTERATIVIDADE NAS INTERFACES GRÁFICAS

Agora que já conseguimos criar o design de nosso sistema, devemos nos preocupar com a **interatividade** dele. Nesse ponto, precisamos entender o conceito de **eventos**.

Ao observarmos o funcionamento dos processadores, percebemos a natureza **sequencial** da execução das instruções. Pode haver nesse processo desvios condicionais ou não, algo que se torna evidente na estrutura da sintaxe utilizada pela linguagem **Assembly**.

Não é por menos que a construção de **algoritmos**, sejam eles estruturados ou não, também apresenta um perfil de execução sequencial. O grande problema é que um computador não executa apenas uma função, devendo se alternar entre tarefas em períodos padronizados (chamados de **time slice**) ou a execução de operações de entrada e saída que não utilizam diretamente o processador.

Para tornar viável esse comportamento, utilizando-se, da melhor forma, o poder de processamento do computador, temos um mecanismo denominado **interrupção**. Trata-se de um sinal que é enviado para o processador solicitando a execução de uma rotina de tratamento específica antes de retornar ao fluxo original.

É por meio do mecanismo de interrupções que identificamos a digitação no teclado ou o clique de mouse, proporcionando o tratamento adequado, algo que funciona muito bem em sistemas de linha de comando. No entanto, em um sistema baseado em **janelas**, com múltiplos componentes visuais aptos a efetuar o tratamento da interrupção e fornecer uma resposta visual adequada, foi necessário um modelo **assíncrono**.

A resposta para as novas necessidades foi a concepção de um sistema de **mensagens**; iniciadas a partir da ocorrência de interrupções, elas são enviadas para os componentes visuais do ambiente, sendo interceptadas a partir de rotinas de **callback**.

★ EXEMPLO

Nas primeiras versões do Windows, a programação era feita basicamente em linguagem C, deixando evidente a interceptação das mensagens enviadas pelo sistema de janelas.

O modelo baseado em mensagens ainda é utilizado nos atuais sistemas de janelas, mas o estilo de programação demonstrou uma complexidade desnecessária, surgindo então, como alternativa, um modelo baseado em **eventos**, que podem ser definidos como ações predeterminadas que, ao ocorrer, permitem iniciar rotinas personalizadas. Em termos práticos, a programação interna dos componentes inclui rotinas de callback para as mensagens do sistema de janelas, porém, em vez de responder diretamente, ela desvia a execução para algum mecanismo que possa ser especializado pelo programador.

Cada ambiente apresenta um mecanismo específico para as respostas aos eventos, como o uso de ponteiros para as funções em Delphi ou por meio de objetos ouvintes no Swing, embora seu objetivo seja sempre o mesmo: garantir que o programador implemente a **interatividade** do sistema, respondendo a ações, como, por exemplo, clique do mouse, digitação ou abertura de uma janela, sem precisar se preocupar com a complexidade interna da comunicação com o sistema de janelas.

📢 ATENÇÃO

Atualmente, qualquer biblioteca de interfaces gráficas utiliza a interatividade baseada no modelo de eventos, não importando a plataforma utilizada.

EVENTOS PARA AWT E SWING

Os eventos permitem uma grande flexibilidade na definição da interatividade do sistema.

⭐ EXEMPLO

Ao clicar sobre o botão, abra a segunda janela ou, ao perder o foco da caixa de texto, verifique o formato do valor digitado.

Nos ambientes AWT e Swing, a resposta aos eventos deve ser programada por meio de interfaces "**ouvintes**", mas existem diversos tipos de eventos disponíveis. Devemos nos preocupar apenas com o ouvinte (ou **Listener**) adequado à ação que esperamos do usuário durante a utilização do sistema.

A tabela seguinte apresenta os principais tipos de interfaces Listener disponíveis, bem como os métodos de resposta:

Listener	Métodos	Parâmetro	Utilização
ActionListener	actionPerformed	ActionEvent	Clique do mouse sobre botões e menus.
FocusListener	focusGained focusLost	FocusEvent	Ganho e perda de foco pelos

			componentes.
KeyListener	keyPressed keyReleased keyTyped	KeyEvent	Acionamento de teclas ou liberação delas.
MouseListener	mouseClicked mouseEntered mouseExited mousePressed mouseReleased	MouseEvent	Eventos gerais para o mouse, como entrada e saída da região ou botão pressionado ou liberado.
MouseMotionListener	mouseDragged mouseMoved	MouseEvent	Movimento do mouse, incluindo arraste (**drag**) ou não.
TextListener	textValueChanged	TextEvent	Alterações no valor do texto.

Atenção! Para visualização completa da tabela utilize a rolagem horizontal

Quadro: Principais interfaces ouvintes para AWT e Swing.
Elaborado por Denis Gonçalves Cople

Observemos a implementação da resposta ao clique do mouse no exemplo seguinte:

```
public class JanelaEventos extends JDialog
    implements ActionListener{
    private JTextField t1 = new JTextField(10),
```

```
        t2 = new JTextField(10);
    private JButton b1 = new JButton("Somar");

    @Override
    public void actionPerformed(ActionEvent e) {
        if(e.getActionCommand().equals("Somar"))
            somar();
    }
    private void somar(){
        int a = Integer.parseInt(t1.getText());
        int b = Integer.parseInt(t2.getText());
        JOptionPane.showMessageDialog(this,"A soma será: "+(a+b));
    }
    public JanelaEventos() {
        setLayout(new FlowLayout());
        add(t1);
        add(t2);
        add(b1);
        b1.addActionListener(this);
    }
}
```

Aqui vemos uma janela de diálogo implementando a interface **ActionListener** com seu método **actionPerformed**, no qual verificamos o componente clicado, que pode ser um botão ou um item de menu, por meio de seu texto. Se o método **getActionCommand** retornar "Somar", o método somar será iniciado.

Antes da implementação do ouvinte, temos a definição dos componentes da janela (no caso, dois campos de texto e um botão) a serem adicionados no construtor da classe com base em um **FlowLayout**, no qual também vemos a associação do **botão** ao ouvinte definido pela janela por meio de **addActionListener**. Com a associação efetuada, o clique sobre o botão acionará o método **actionPerformed**, seu texto será reconhecido e o método **somar**, executado, capturando os valores da tela convertidos para o tipo inteiro e mostrando o resultado da soma em um diálogo padronizado.

📢 ATENÇÃO

Embora a abordagem adotada aqui seja válida, ela pode ser ineficiente, pois o aumento do número de botões exige uma grande quantidade de testes para se saber qual botão foi acionado. Por conta disso, uma prática mais interessante na resposta aos eventos seria instanciar um **objeto anônimo**, pois teríamos a associação direta com o botão ou o item de menu, eliminando a necessidade do teste com getActionCommand.

A modificação para o uso de objeto anônimo pode ser observada a seguir:

```
public class JanelaEventos extends JDialog {
	private JTextField t1 = new JTextField(10),
			t2 = new JTextField(10);
	private JButton b1 = new JButton("Somar");

	private void somar(){
		int a = Integer.parseInt(t1.getText());
		int b = Integer.parseInt(t2.getText());
		JOptionPane.showMessageDialog(this,"A soma será: "+(a+b));
	}
	public JanelaEventos() {
		setLayout(new FlowLayout());
		add(t1);
		add(t2);
		add(b1);
		b1.addActionListener(new ActionListener() {
		@Override
		public void actionPerformed(ActionEvent e) {
		somar();
		}
		});
	}
}
```

Na segunda versão, não temos a implementação do ouvinte pela janela. A associação do botão, por meio de **addActionListener**, recebe uma **instância** de ActionListener.

É claro que, por se tratar de um elemento abstrato, a implementação de seu único método, ou seja, **actionPerformed**, é obrigatória, sendo que agora não há a necessidade de testes para saber qual foi o botão clicado, podendo o método **somar** ser chamado diretamente.

Atualmente, pode-se simplificar ainda mais a escrita para diversos tipos de evento. Basta adotar a programação funcional com o uso de **lambda**.

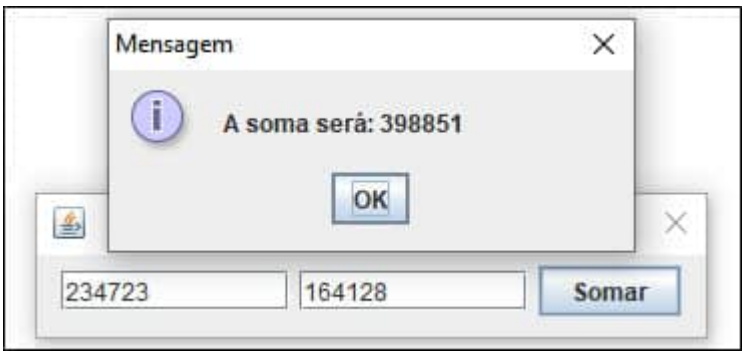
É possível ainda criar uma terceira versão de nosso código, com uma escrita bem mais concisa, efetuando, como é demonstrado a seguir, a substituição da associação do ouvinte:

```
 b1.addActionListener((ActionEvent e) -> {
   somar();
});
```

Para invocarmos nossa janela, precisamos de um método **main** como o deste código:

```
public static void main(String[] args) {
      JanelaEventos janela = new JanelaEventos();
      janela.setModal(true);
      janela.setBounds(50, 50, 350, 80);
      janela.setVisible(true);
}
```

Independentemente do modelo adotado, o resultado da execução será o mesmo. Ele pode ser observado na figura apresentada a seguir:



Exemplo com eventos no Swing.
Captura de tela de interface gráfica gerada com Java e biblioteca Swing e AWT no NetBeans.

EVENTOS NO SWT

Da mesma forma que no Swing, o modelo do SWT trabalha com interfaces ouvintes para os diversos eventos que podem ocorrer sobre os componentes. Cada **Widget** aceita determinados tipos de interfaces **Listener**, enquanto os dados acerca do evento são recuperados por meio de um elemento **Event**.

A prática mais comum na captura de eventos do SWT é a utilização das classes **Adapter**, pois elas já oferecem uma implementação básica de cada tipo de Listener, o que diminui a quantidade de métodos que precisariam ser codificados.

Vejamos um exemplo de resposta ao clique com a utilização de Adapter:

```
public class JanelaEventosSWT {
	private final Shell;
	private Text t1;
	private Button b1;
	public JanelaEventosSWT(Display display) {
		shell = new Shell(display);
		configurarJanela();
		configurarEvento();
	}
	private void configurarJanela() {
		shell.setSize(200, 80);
		t1 = new Text(shell, SWT.BORDER);
		t1.setBounds(10, 10, 100, 25);
		b1 = new Button(shell, SWT.PUSH);
		b1.setBounds(115, 10, 50, 25);
		b1.setText("CLIQUE");
	}
	private void configurarEvento() {
		SelectionListener listener = new SelectionAdapter() {
		@Override
		public void widgetSelected(SelectionEvent e) {
		System.out.println("Bem-vindo(a) " + t1.getText());
		}
		};
		b1.addSelectionListener(listener);
	}
	public Shell getShell(){
```

```
        return shell;
    }
}
```

No início de nossa classe, temos a definição de um campo de texto, um botão e uma referência para a janela (**shell**). O construtor é bem simples, instanciando **shell** a partir de **display** e chamando os métodos configurarJanela e configurarEvento.

No método **configurarJanela**, definimos o tamanho da janela e instanciamos a caixa de texto e o botão. Configuramos o posicionamento e o tamanho de cada um, além de definirmos o texto do botão.

💬 COMENTÁRIO

O processo é idêntico ao que foi demonstrado anteriormente neste conteúdo.

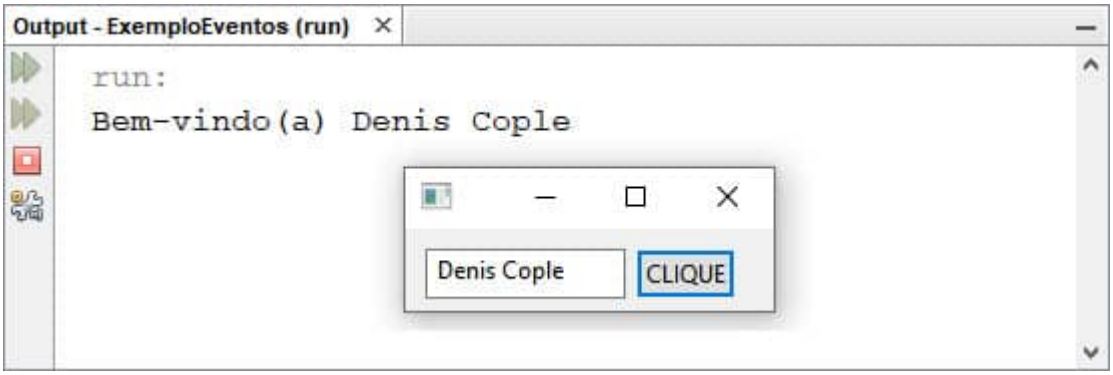
Já o método **configurarEvento** é iniciado com a criação do **SelectionListener** a partir de um **SelectionAdapter**, havendo a sobrescrita do método **widgetSelected** para que a mensagem seja impressa na linha de comando em resposta ao clique. Em seguida, o objeto ouvinte é associado ao botão, por meio do método **addSelectionListener**, com a passagem de nosso **listener**. Temos ainda um método **getShell**, que tem como objetivo retornar o **shell** interno já devidamente configurado.

Com o código de nossa janela pronto, será necessário um método **main** para testá-la:

```
public static void main(String[] args) {
    Display display = new Display();
    Shell shell = new JanelaEventosSWT(display).getShell();
    shell.open();
    while (!shell.isDisposed()) {
        if (!display.readAndDispatch()) {
        display.sleep();
        }
    }
    display.dispose();
}
```

O método é similar aos que foram apresentados neste conteúdo, porém a instância de **Shell** é obtida a partir de um objeto do tipo **JanelaEventosSWT**. Dessa forma, ela recebe a janela totalmente configurada.

O resultado da execução no ambiente do NetBeans pode ser visto a seguir:



Exemplo com eventos no SWT.

Captura de tela da interface gráfica gerada com Java e biblioteca SWT no NetBeans.

Assim como nas bibliotecas AWT e Swing, o SWT conta com ouvintes para diversos tipos de eventos:

Listener	Métodos	Parâmetro	Componentes
FocusListener	focusGained focusLost	FocusEvent	Control
KeyListener	keyPressed keyReleased	KeyEvent	Control
MouseListener	mouseDoubleClick mouseDown mouseUp	MouseEvent	Control

MouseMoveListener	mouseMove	MouseMoveEvent	Control
SelectionListener	widgetDefaultSelected widgetSelected	SelectionEvent	Button, MenuItem, ToolItem
ShellListener	shellActivated shellClosed shellDeactivated shellDeiconified shellIconified	ShellEvent	Shell

Atenção! Para visualização completa da tabela utilize a rolagem horizontal

Quadro: Algumas das interfaces ouvintes para SWT.
Elaborado por Denis Gonçalves Cople

EVENTOS NO JAVAFX

Para iniciarmos nossa análise acerca de eventos no JavaFX, observaremos um trecho de código apresentado no módulo anterior:

```
Button btn = new Button();
btn.setText("Say 'Hello World'");
btn.setOnAction(new EventHandler() {
 @Override
 public void handle(ActionEvent event) {
  System.out.println("Hello World!");
 }
});
```

O modelo de eventos utilizado no JavaFX não se diferencia, em termos funcionais, do que foi adotado para as demais bibliotecas, com a implementação de um objeto ouvinte e a associação dele ao componente que receberá o evento. Entretanto, o JavaFX utiliza amplamente a **modelagem comportamental**, como no exemplo apresentado, com o uso de **genéricos** para a programação da resposta ao clique.

Temos aqui a definição de um botão e da ação de clique sobre ele no qual será impresso a mensagem "Hello World!" no prompt. Para a programação da resposta, é gerado um **EventHandler** para **ActionEvent** e codificado o método **handle**, o qual, devido à tipagem proporcionada pelos elementos genéricos, tem um parâmetro do tipo ActionEvent.

A classe **Event** é a base da hierarquia de eventos, sendo as características específicas de cada tipo de evento, como tecla pressionada ou posição do mouse, capturadas graças a seus descendentes.

Este quadro contém os principais descendentes de Event:

Subclasse de Event	Utilização
ActionEvent	Tratamento das ações de clique sobre o componente.
MouseEvent	Detecção da atividade do mouse sobre o componente.
DragEvent	Tratamento das operações de arraste (**drag-and-drop**).
KeyEvent	Controle do pressionamento de teclas.
ScrollEvent	Rolagem de tela a partir do botão de rolagem do mouse, toque de tela, track pad ou outro dispositivo com o mesmo perfil.
TouchEvent	Indicação de atividades com toque de tela.

Atenção! Para visualização completa da tabela utilize a rolagem horizontal

Quadro: Subclasses de Event no JavaFX.
Elaborado por Denis Gonçalves Cople

Atualmente, é possível utilizar o paradigma funcional, com base no operador **lambda**, no tratamento de eventos do JavaFX. Poderíamos modificar o trecho de código anterior para adotar o novo modelo, simplificando o código:

```
Button btn = new Button();
btn.setText("Say 'Hello World'");
btn.setOnAction(event -> {
 System.out.println("Hello World!");
});
```

Devemos lembrar que o JavaFX não permite apenas o modo programado, pois o uso de **FXML** simplifica muito o tratamento de eventos. A resposta é implementada por meio de um método do **controlador** e associada ao componente na **tag** do arquivo FXML.

Completemos nosso exemplo do módulo anterior codificando o controlador gerado pelo NetBeans:

```
public class TesteFXController implements Initializable {
	@FXML
	Label lblMensagem;
	@FXML
	TextField txtNome;


	public void atualizar()
	{
		lblMensagem.setText("Bem-vindo "+txtNome.getText());
	}
	@Override
	public void initialize(URL location, ResourceBundle resources)
	{
	}
}
```

Ao anotarmos **lblMensagem** e **txtNome** com **@FXML**, estamos dizendo ao compilador para associar os atributos da classe às **tags** do arquivo FXML cujos valores de **fx:id** sejam equivalentes aos nomes dos atributos.

📢 ATENÇÃO

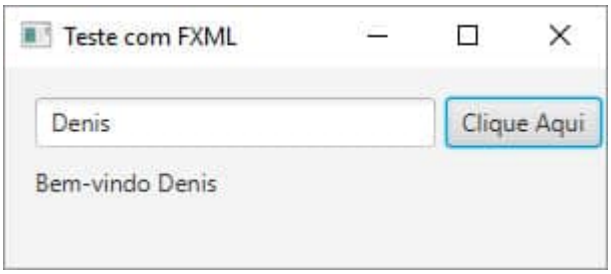
As tags devem ser do mesmo tipo das classes dos atributos, como ocorre em TextField, por exemplo, no caso de txtNome.

A programação da resposta foi feita com a implementação do método **atualizar**, em que **lblMensagem** tem o texto atualizado com a concatenação da expressão "**Bem-vindo**" e o nome digitado em **txtNome**. São utilizados os métodos **setText** para definir o novo texto, e **getText** para recuperar o atual.

Ao final da codificação do método de resposta ao clique, ele deve ser associado ao botão por meio do atributo **onAction** na tag **Button** de nosso arquivo FXML. Ainda precisamos, contudo, alterar o conteúdo de **AnchorPane** no arquivo **FXML** para efetuar as conexões descritas:

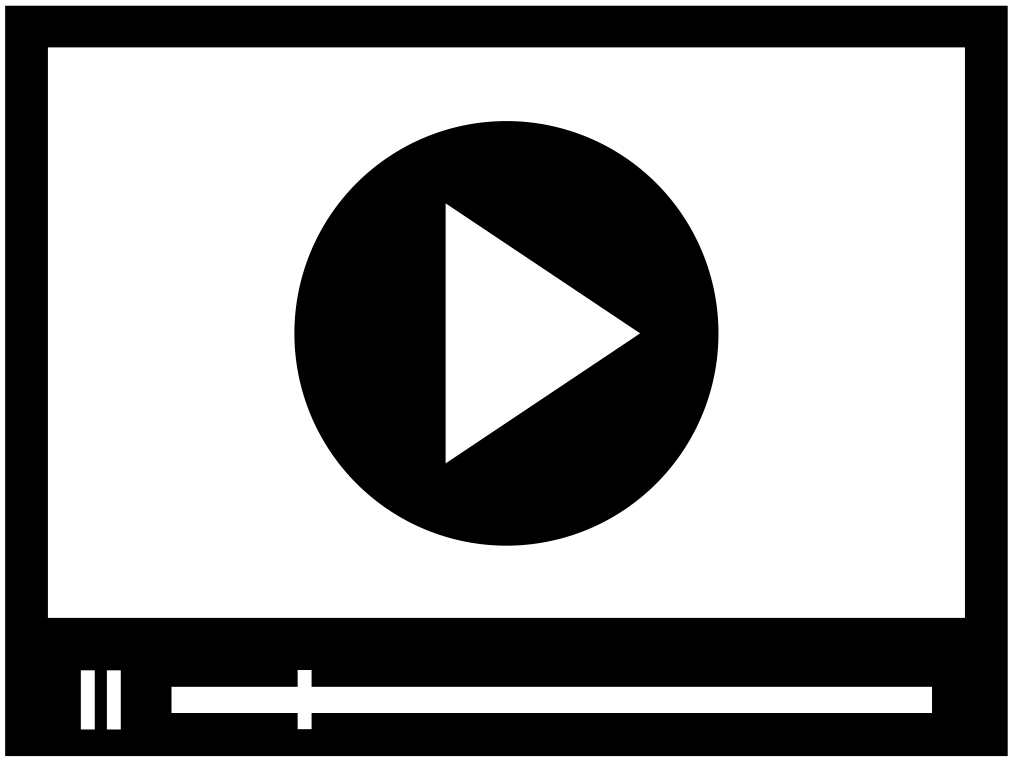
```
<AnchorPane id="AnchorPane" prefHeight="400.0" prefWidth="600.0"
 styleClass="mainFxmlClass" xmlns:fx="http://javafx.com/fxml/1"
 fx:controller="exemplojavafx.TesteFXController">
 <stylesheets>
      <URL value="@testefx.css"/>
 </stylesheets>
  <children>
      <TextField layoutX="15.0" layoutY="15.0" prefWidth="200"
      fx:id="txtNome"/>
      <Button layoutX="220" layoutY="15.0" text="Clique Aqui"
      onAction="#atualizar"/>
      <Label fx:id="lblMensagem" layoutX="15.0" layoutY="45"
      prefHeight="25" prefWidth="260" text="Mensagem"/>
  </children>
</AnchorPane>
```

Gerado de forma automática, o método **initialize** se refere apenas à implementação da interface **Initializable**, que é requerida pelo NetBeans. Agora o botão já responde ao clique conforme demonstra a figura ao lado:



📷 Exemplo com resposta ao clique no JavaFX.

Captura de tela de interface gráfica gerada com Java, JavaFX e FXML no NetBeans.



EVENTOS E INTERATIVIDADE

O conceito de eventos e a demonstração da implementação de resposta, com base em AWT/ Swing, SWT e JavaFX são apresentados no vídeo a seguir.

VERIFICANDO O APRENDIZADO

MÓDULO 4

- **Aplicar a arquitetura MVC e os padrões de desenvolvimento nos sistemas JavaFX.**

PADRÕES DE DESENVOLVIMENTO

Os padrões de desenvolvimento buscam o reuso do conhecimento, trazendo modelos padronizados de soluções para problemas já conhecidos no mercado. Com a utilização de padrões, temos uma visão mais estratégica para o projeto do sistema, facilitando a manutenção do código e evitando problemas já conhecidos devido a experiências anteriores de desenvolvimento.

Um exemplo simples de padrão com larga aceitação é o **DAO**. Referente à concentração das operações de acesso a banco de dados, ele evita a multiplicação de comandos SQL ao longo de todo o código. A implementação do padrão exige a criação de uma classe de entidade e outra de gerenciamento, as quais, em nosso exemplo, serão **Produto** e **ProdutoDAO**, sendo apresentadas nas listagens adiante.

Inicialmente, temos o código da classe **Produto** representando nossa **entidade**:

```
public class Produto {
    public int codigo;
    public String nome;
    public int quantidade;

    public Produto(){ }
    public Produto(int codigo, String nome, int quantidade) {
        this.codigo = codigo;
        this.nome = nome;
```

```
        this.quantidade = quantidade;
    }
}
```

A classe de entidade representa uma tabela de mesmo nome no banco de dados. Essa classe normalmente é constituída apenas de:

Atributos referentes aos campos da tabela.

Alguns construtores.

Encapsulamento por meio de getters e setters.

💬 COMENTÁRIO

Nosso exemplo não adota o encapsulamento padrão, pois desejamos obter um código menos extenso.

Com a entidade definida, devemos criar nossa classe **DAO** com o nome **ProdutoDAO**:

```
public class ProdutoDAO {
    private Connection getConnection() throws Exception {
        Class.forName("org.apache.derby.jdbc.ClientDriver");
        return DriverManager.getConnection(
            "jdbc:derby://localhost:1527/ExemploFX",
            "ExemploFX", "ExemploFX");
    }
    public List<Produto> obterTodos() {
        ArrayList<Produto> lista = new ArrayList<>();
        try (Connection c1 = getConnection()) {
        ResultSet r1 = c1.createStatement().
                executeQuery("SELECT * FROM PRODUTO");
        while (r1.next())
        lista.add(new Produto(r1.getInt("CODIGO"),
            r1.getString("NOME"),r1.getInt("QUANTIDADE")));
        } catch (Exception ex) { }
```

```
        return lista;
    }
    public void incluir(Produto p) {
        try (Connection c1 = getConnection()) {
        PreparedStatement ps = c1.prepareStatement(
                "INSERT INTO PRODUTO VALUES(?,?,?)");
        ps.setInt(1, p.codigo);
        ps.setString(2, p.nome);
        ps.setInt(3, p.quantidade);
        ps.executeUpdate();
        } catch (Exception e) { }
    }
}
```

Na programação de **ProdutoDAO**, temos um método utilitário interno para a obtenção da **conexão** com o banco de dados, além de um método de consulta e outro de inserção. Não seguimos o padrão DAO completo segundo o qual seria necessário implementar os métodos para a alteração e a exclusão de dados.

No entanto, obedecemos à seguinte **regra básica** do padrão DAO:

Os comandos SQL se concentram no interior dos métodos.
Toda comunicação com outros objetos envolve o uso de **entidades** e as coleções delas.

Atenção! Para visualização completa da tabela utilize a rolagem horizontal

A consulta ocorre no método **obterTodos**, que executa um comando **SELECT** na tabela **PRODUTO**, obtendo o resultado em um **ResultSet** e adicionando à **lista** de retorno um **objeto** com os dados do banco para cada **registro** da consulta. Enquanto isso, o método **incluir** utiliza um comando **INSERT** parametrizado cujos valores que complementam o comando são preenchidos com os dados fornecidos pela **entidade**.

Nos dois casos, temos a utilização de **try with resources**, que simplifica a escrita ao mesmo tempo em que garante o fechamento da conexão com o banco de dados ao final.

Para que nosso código seja funcional, precisamos criar um banco de dados **Java DB** com o nome **ExemploFX**, valor adotado também para usuário e senha. Já no banco, devemos criar a tabela **PRODUTO** por intermédio do comando **SQL** (a ser apresentado a seguir).

Também é necessário acrescentar a biblioteca **Java DB Driver** ao projeto **Java** utilizando o clique com o botão direito na divisão **Libraries** e a escolha da opção **Add Library**:

```
CREATE TABLE PRODUTO(CODIGO INT PRIMARY KEY,
        NOME VARCHAR(30), QUANTIDADE INT)
```

Obviamente, existem diversos outros padrões.

★ EXEMPLO

O Singleton garante uma única instância para a classe e pode ser utilizado em controles de acesso centralizados. Já o **Decorator** adiciona funcionalidades a um objeto existente, sendo muito utilizado com **Stream** no Java.

Na verdade, já vimos um exemplo de **Decorator** ao criarmos a classe **JanelaEventosSWT**, pois tivemos a obtenção de um **Shell** com diversos elementos associados, incluindo a resposta ao clique. No entanto, o loop do programa principal não precisou de qualquer alteração significativa. Em termos práticos, apenas "decoramos" uma janela e seguimos o fluxo de execução normal.

ARQUITETURA MVC (MODEL, VIEW E CONTROLLER)

Embora os padrões de desenvolvimento solucionem boa parte dos problemas internos do desenvolvimento de um sistema, devemos observar também que um software pode ter uma estratégia **arquitetural** que satisfaça a determinados **domínios**. Surge daí a ideia por trás dos **padrões arquiteturais**.

Uma arquitetura muito comum no mercado corporativo é a de **Broker**. Ela é utilizada para objetos distribuídos, como **CORBA** e **EJB**, definindo a presença de Stubs e Skeletons,

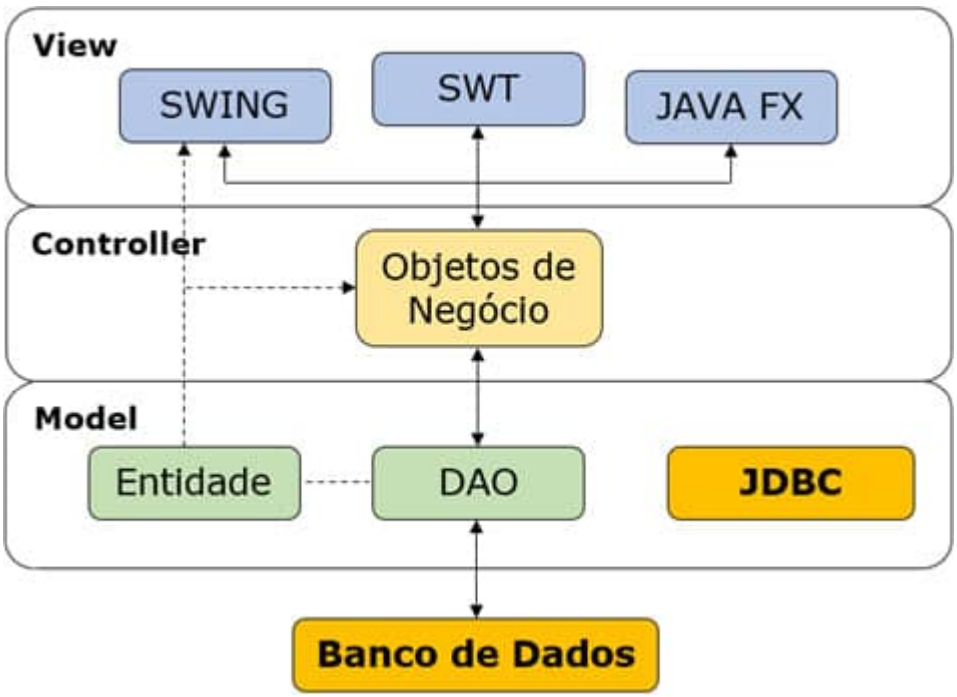
descritor de serviço e protocolo de comunicação, entre outros elementos, para viabilizar a distribuição do processamento.

O uso de um padrão arquitetural sugere a adoção de diversos padrões de desenvolvimento associados, como **Proxy** e **Fly Weight**, presentes respectivamente na comunicação e na gestão de um pool de objetos do Broker.

Entre os diversos padrões arquiteturais existentes, o **Model-View-Controller** (ou **MVC**) acabou dominando o mercado de desenvolvimento para aplicações cadastrais. O padrão define a divisão do sistema em três camadas:

Persistência de dados (**Model**).
Interação com o usuário (**View**).
Processos de negócios (**Controller**).

Atenção! Para visualização completa da tabela utilize a rolagem horizontal



📷 Representação da arquitetura MVC.

Nossa camada **Model** já está completa com a criação de **Produto** e **ProdutoDAO**. O próximo passo será a construção da **Controller**.

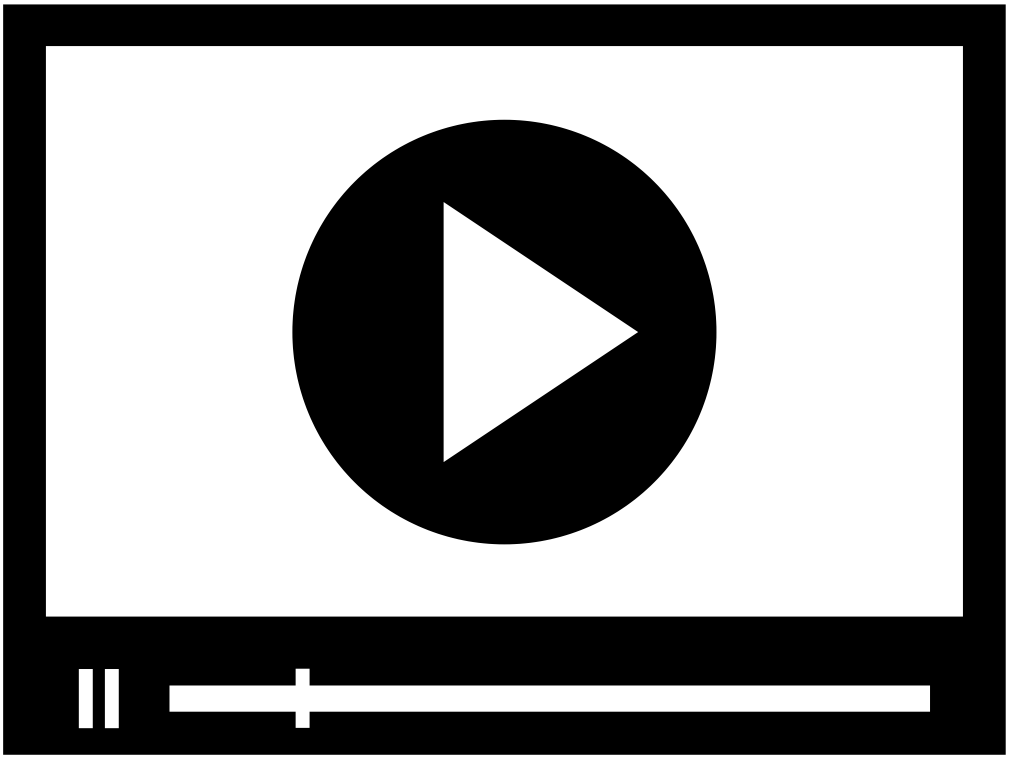
A principal característica dos objetos de negócios é a independência completa de ambiente, ou seja, o mesmo controlador deve poder ser utilizado em diversas interfaces gráficas, Servlets, web services ou qualquer outro elemento da camada View sem adaptações ou direcionamentos.

```
public class ProdutoController {
	private static ProdutoDAO dao = new ProdutoDAO();
	public static List<Produto> obterTodos(){
		return dao.obterTodos();
	}
	public static void incluir(Produto){
		dao.incluir(produto);
	}
}
```

Em nosso exemplo, um controlador de base **estática** segue o padrão **Facade**, no qual as chamadas são repassadas para o **DAO**, enquanto o retorno de consultas é emitido para o chamador sem regras de negócio adicionais. Não seria difícil acrescentar regras, como impedir a inclusão de um produto com quantidade negativa, o que seria resolvido com uma simples estrutura condicional.

```
public void incluir(Produto produto){
	if(produto.quantidade >= 0)
		dao.incluir(produto);
}
```

A próxima e última camada será a **View**. Ela será implementada utilizando uma tecnologia **JavaFX**. Contudo, para facilitarmos o processo de criação das telas, adotaremos uma ferramenta denominada **JavaFX Scene Builder** (a ser abordada em nosso próximo tópico).



ARQUITETURA MVC

Para finalizar esse módulo, apresentamos a descrição da arquitetura MVC, com o delineamento de componentes utilizados em cada camada, no vídeo a seguir.

JAVAFX SCENE BUILDER

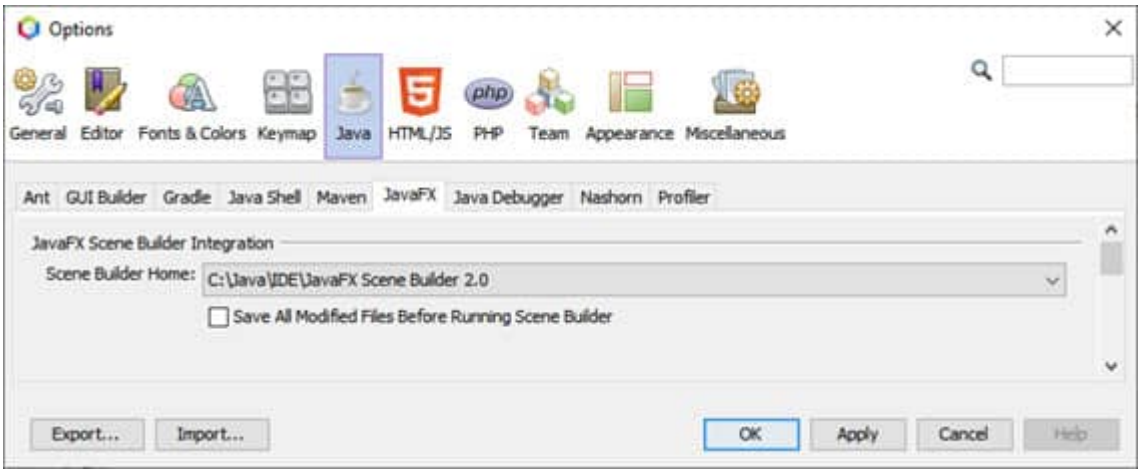
Como a sintaxe de um arquivo **FXML** é um pouco complexa, torna-se interessante adotar uma ferramenta para a garantia de produtividade na criação de telas **JavaFX** – no caso, o aplicativo da Oracle **JavaFX Scene Builder**.

SAIBA MAIS

O **JavaFX Scene Builder** está disponível na divisão JavaFX da página da Oracle, sendo utilizada aqui a versão 2.0 para o ambiente Windows.

Com o uso do **Scene Builder**, as janelas podem ser criadas de forma simples, com o arraste de componentes em um ambiente de criação visual. É possível ainda associar componentes de estilização **CSS** e classes de **efeito** JavaFX por meio de um painel de **propriedades** bastante organizado.

Para integrar a ferramenta com o **NetBeans**, basta apontar para o diretório de instalação na opção **JavaFX** (divisão **Java**) do painel acessado a partir do menu **Tools.Options**.



Integração do JavaFX Scene Builder ao NetBeans.
Captura de tela do NetBeans.

Após efetuar a configuração, o duplo clique sobre um arquivo **FXML** ou a escolha da opção **Open** no menu de contexto, se é levado para a tela do **Scene Builder**, enquanto a opção **Edit** levará para a visualização em modo texto no editor de código do **NetBeans**.

Criaremos mais um FXML vazio no projeto JavaFX com o nome de **CadastroProduto**, acrescentando tanto o controlador quanto o arquivo CSS. Com o duplo clique sobre o arquivo FXML, podemos abrir o **Scene Builder** e desenhar esta tela:

📷 Tela para cadastro de produtos com JavaFX.
Captura de tela de interface gráfica gerada com JavaFX Scene Builder e FXML no NetBeans.

Para construirmos nossa tela, precisaremos de três componentes **Label**, três do tipo **TextField**, um **Button** e um **TableView**. O componente TableView já traz duas colunas por padrão. Devemos adicionar então um terceiro componente **TableColumn** a ele.

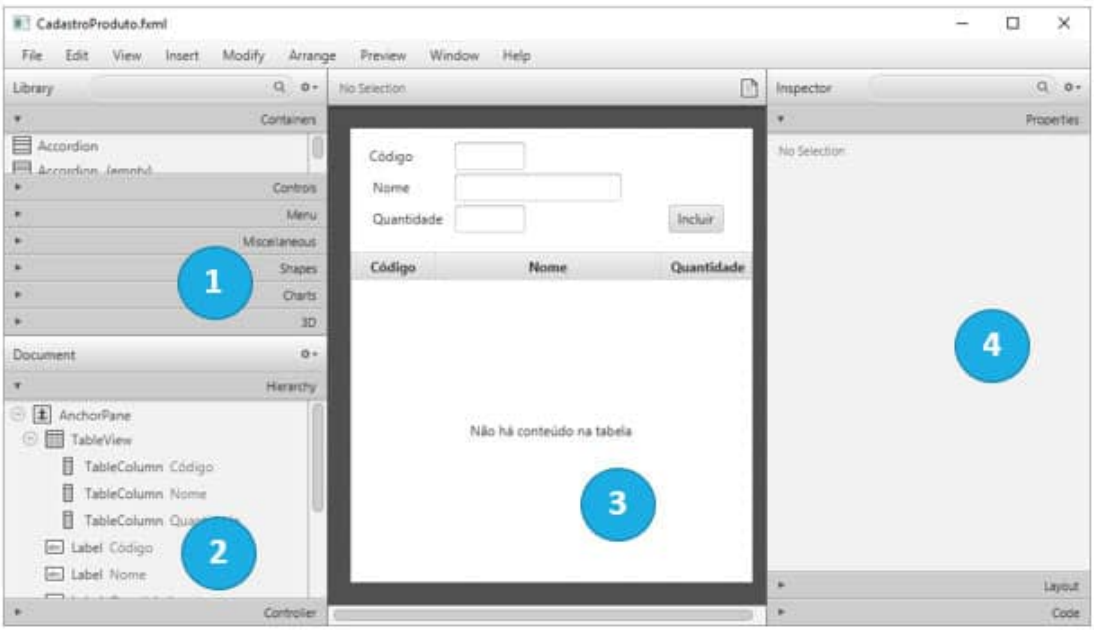
💡 DICA

Todos esses movimentos são feitos com operações de arraste.

O ambiente do JavaFX Scene Builder é dividido em **quatro** partes principais:

Paleta de componentes.
Navegador.
Editor visual.
Inspector (propriedades).

Atenção! Para visualização completa da tabela utilize a rolagem horizontal



Ambiente de criação do JavaFX Scene Builder.

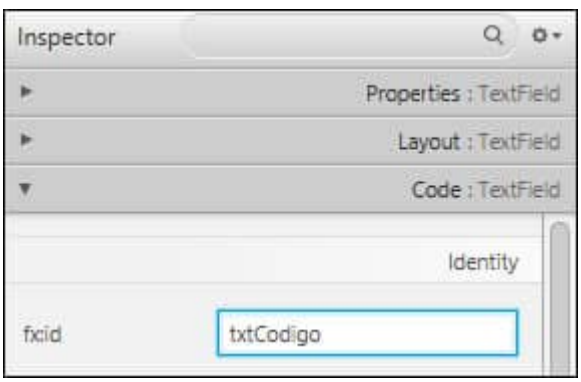
Captura de tela do JavaFX Scene Builder integrado ao NetBeans.

Os componentes são arrastados a partir da **paleta** de componentes para o **editor visual**, sendo acrescentados na hierarquia do **navegador**. Qualquer componente selecionado a partir do clique na área de edição é configurado no painel de **propriedades**.

Inicialmente, apenas a propriedade **Text** dos componentes será alterada com o uso de "Incluir" no **Button** e de "Código", "Nome" e "Quantidade" para os três componentes **Label** e os três do tipo **TableColumn**. Já os redimensionamentos e os posicionamentos podem ser feitos visualmente.

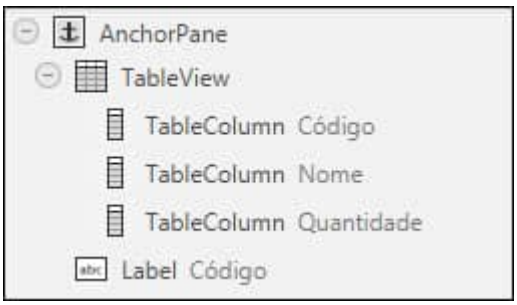
Agora definiremos o valor dos identificadores a serem utilizados na programação – no caso, para os componentes dos tipos **TextField** e **TableView**.

Devemos configurar o atributo **fx:id** por meio do **Inspector** (divisão **Code**) com os valores **txtCodigo**, **txtNome** e **txtQuantidade** para os componentes TextField e **tblProduto** para a TableView:



📷 Painel Inspector do JavaFX Scene Builder.
Captura de tela do JavaFX Scene Builder integrado ao NetBeans.

Os identificadores das colunas também precisam ser definidos, sendo adotados os nomes **cltCodigo**, **cltNome** e **cltQuantidade**. No entanto, a seleção de TableColumn é mais simples a partir da hierarquia do navegador.



📷 Painel de navegação do JavaFX Scene Builder.
Captura de tela do JavaFX Scene Builder integrado ao NetBeans.

SISTEMA MVC NO JAVAFX

Com a interface completa, incluindo os **identificadores**, podemos concluir nosso sistema na arquitetura **MVC**.

📢 ATENÇÃO

Utilizaremos as classes criadas nos tópicos anteriores, mas não existe uma compatibilidade direta entre a classe de entidade e os componentes JavaFX.

Como pretendemos acrescentar a compatibilidade ao ambiente para nossa entidade, adotaremos o padrão **Decorator** com a criação da classe **ProdutoFX**:

```
public class ProdutoFX {
    private final SimpleIntegerProperty codigo;
    private final SimpleStringProperty nome;
    private final SimpleIntegerProperty quantidade;
    public ProdutoFX(Produto produto){
        this.codigo = new SimpleIntegerProperty(produto.codigo);
```

```
            this.nome = new SimpleStringProperty(produto.nome);
            this.quantidade =
                  new SimpleIntegerProperty(produto.quantidade);
      }
      public int getCodigo(){ return codigo.get(); }
      public SimpleIntegerProperty codigoProperty() {
            return codigo;
      }
      public String getNome(){ return nome.get();  }
      public SimpleStringProperty nomeProperty() {
            return nome;
      }
      public int getQuantidade(){ return quantidade.get(); }
      public SimpleIntegerProperty quantidadeProperty() {
            return quantidade;
      }
}
```

Para a integração com o **TableView**, é necessário que os atributos da classe **ProdutoFX** sejam objetos de tipos, como, por exemplo, **SimpleIntegerProperty** ou **SimpleStringProperty**. Também deve haver métodos com a finalização **Property** para o acesso aos atributos, bem como getters e opcionalmente setters para os valores internos deles.

Observe que ProdutoFX recebe um objeto do tipo **Produto** em seu construtor, iniciando os atributos com os valores fornecidos pela entidade, o que denota a utilização do padrão **Decorator**. Também deve ser criada a classe **ListProdutoFX**, que tem de funcionar como um **Helper** para a listagem de produtos original (também no padrão Decorator):

```
public class ListProdutoFX extends ArrayList<ProdutoFX>{
      public ListProdutoFX(List<Produto> origem){
            origem.forEach((p) -> {
            add(new ProdutoFX(p));
            });
      }
}
```

A classe **ListProdutoFX** é uma coleção de **ProdutoFX** que tem o construtor alterado para a recepção de uma coleção de produtos e o uso de **forEach** para a conversão de cada entidade

em ProdutoFX, seguido da adição do elemento convertido.

Com as classes de suporte prontas, podemos começar a codificação do controlador Java, que tem o nome **CadastroProdutoController**:

```
public class CadastroProdutoController implements Initializable {
    @FXML TableView<ProdutoFX> tblProduto;
    @FXML TableColumn<ProdutoFX,Integer> cltCodigo;
    @FXML TableColumn<ProdutoFX,String> cltNome;
    @FXML TableColumn<ProdutoFX,Integer> cltQuantidade;
    @FXML TextField txtCodigo;
    @FXML TextField txtNome;
    @FXML TextField txtQuantidade;

    public void incluirClique(){
        Produto produto = new Produto();
        produto.codigo = new Integer(txtCodigo.getText());
        produto.nome = txtNome.getText();
        produto.quantidade = new Integer(txtQuantidade.getText());
        ProdutoController.incluir(produto);
        txtCodigo.setText("");
        txtNome.setText("");
        txtQuantidade.setText("");
        tblProduto.setItems(obterListaProduto());
    }
    @Override
    public void initialize(URL url, ResourceBundle rb) {
        cltCodigo.setCellValueFactory(
            new PropertyValueFactory<>("codigo"));
        cltNome.setCellValueFactory(
            new PropertyValueFactory<>("nome"));
        cltQuantidade.setCellValueFactory(
            new PropertyValueFactory<>("quantidade"));
        tblProduto.setItems(obterListaProduto());
    }
    private ObservableList<ProdutoFX> obterListaProduto() {
        return FXCollections.observableArrayList(
```

```
                new ListProdutoFX(ProdutoController.obterTodos()));
    }
}
```

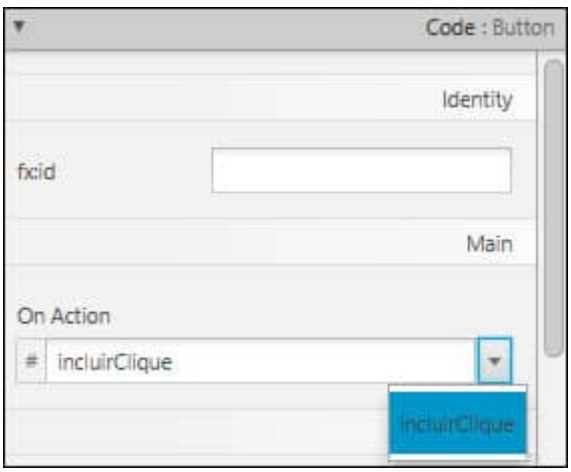
Inicialmente, é feita a associação de todos os elementos identificados por **fx:id** com o uso da anotação **FXML**, ao mesmo tempo em que as classes de preenchimento da tabela e do atributo de cada coluna são definidas por intermédio de **elementos genéricos**. Graças à configuração efetuada, definimos que o **tblProduto** utilizará como base a classe **ProdutoFX**, enquanto a coluna **cltCodigo** buscará um elemento do tipo **Integer** na classe **ProdutoFX**.

Na inicialização da janela, verificamos os relacionamentos das **colunas** da tabela com os atributos de **ProdutoFX** tendo como base **PropertyValueFactory**, o que justifica a nomenclatura utilizada nos **getters**, com a finalização **Property**.

Ao final, a **tabela** é preenchida com a invocação de **setItems**. Tendo como parâmetro a coleção do tipo **ObservableList**, ela é retornada pelo método **obterListaProduto**, no qual a lista correta pode ser gerada facilmente com o uso de **FXCollections.observableArrayList**, cujo parâmetro, por sua vez, é o de uma coleção de objetos do tipo **ProdutoFX**.

Há ainda o método **incluirClique**, que é associado ao clique do **botão**, devendo executar as ações relacionadas à inclusão de registro e atualização da TableView. Os dados da janela são capturados a partir das caixas de texto e convertidos nos tipos corretos para o preenchimento de um objeto do tipo **Produto**, o qual, por sua vez, é passado para o método **incluir** de **ProdutoController**, seguido da limpeza dos textos e da atualização da TableView com uma nova chamada para **setItems**.

Com o código finalizado, só falta associar o método **incluirClique** ao atributo **Action** do botão. Isso pode ser feito visualmente por intermédio do Inspector de **JavaFX Scene Builder** ou diretamente no arquivo **FXML**.



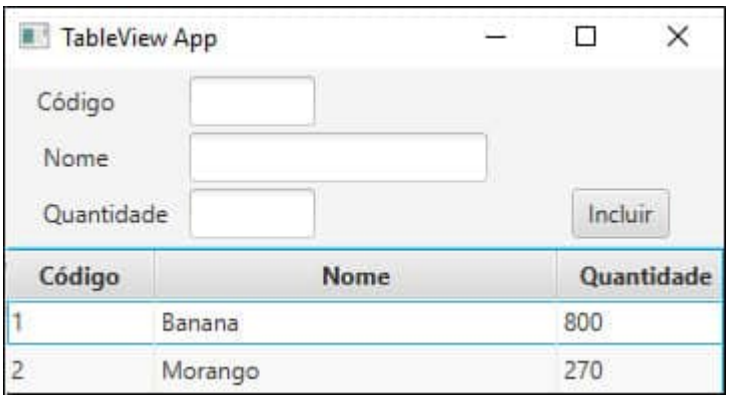
Associação de evento no JavaFX Scene Builder.

Captura de tela de interface gráfica gerada com JavaFX Scene Builder e FXML no NetBeans.

Agora só precisamos criar um **aplicativo** JavaFX por meio da opção **JavaFX Main Class** e adotar uma programação equivalente à que utilizamos anteriormente neste conteúdo:

```
public class Cadastro extends Application {
    @Override
    public void start(Stage primaryStage) throws IOException {
        Parent root = FXMLLoader.load(
            getClass().getResource("CadastroProduto.fxml"));
        Scene scene = new Scene(root, 360, 410);
        primaryStage.setScene(scene);
        primaryStage.setTitle("TableView App");
        primaryStage.show();
    }
    public static void main(String[] args) {
        launch(args);
    }
}
```

Efetuando uma compilação completa de nosso projeto e executando a classe **Cadastro**, poderemos observar nosso sistema **JavaFX** dentro da arquitetura **MVC**.



Versão final do sistema em execução.

Captura de tela de interface gráfica gerada com JavaFX Scene Builder e FXML no NetBeans.

VERIFICANDO O APRENDIZADO

CONCLUSÃO

CONSIDERAÇÕES FINAIS

Ao longo deste conteúdo, observamos diversas tecnologias para a criação de interfaces gráficas em Java. Para isso, apontamos suas diferenças e semelhanças, além do histórico de cada uma, o que permite uma compreensão melhor acerca de seus princípios funcionais.

Analisamos, em seguida, os fundamentos da interatividade nas interfaces gráficas. Após a definição do conceito de evento, vimos que, mesmo apresentando pequenas diferenças, todas as tecnologias estudadas capturam os eventos por meio de algum tipo de ouvinte.

Oferecemos ainda uma visão introdutória dos padrões de desenvolvimento e da arquitetura MVC. Por fim, realizamos a construção de um sistema cadastral com base na tecnologia JavaFX, seguindo, para tal, a arquitetura e os padrões sugeridos.

PODCAST

No podcast a seguir, apresentamos um resumo do conteúdo estudado.

AVALIAÇÃO DO TEMA:

REFERÊNCIAS

CORNELL, G; HORSTMANN, C. **Core Java**. 8. ed. São Paulo: Pearson, 2010.

DEITEL, P; DEITEL, H. **Java**: como programar. 8. ed. São Paulo: Pearson, 2010.

EBBERS, H. **Mastering JavaFX 8 controls**. 1. ed. New York: McGrall Hill, 2014.

GUOJIE, J. L. **Professional Java native interfaces with SWT/JFace**. 1. ed. New Jersey: John Wiley & Sons, 2005.

HECKLER, M. *et al*. **JavaFX 8** – introduction by example. 2. ed. New York: Springer Verlag, 2014.

SCARPINO, M; HOLDER, S; MIHALKOVIC, L. **SWT/JFace in action**. 1. ed. Shelter Island: Manning, 2005.

SCHILDT, H. **Introducing JavaFX 8 programming**. 1. ed. New York: McGraw Hill, 2015.

EXPLORE+

Pesquise na internet os seguintes temas:

A criação de janelas com NetBeans.

A edição visual de janelas SWT com Jigloo.

A apostila de SWT EclipseCon2005_Tutorial5.

A criação de CRUD com JavaFX.

A arquitetura do JavaFX.

CONTEUDISTA

Denis Gonçalves Cople

🔗 CURRÍCULO LATTES